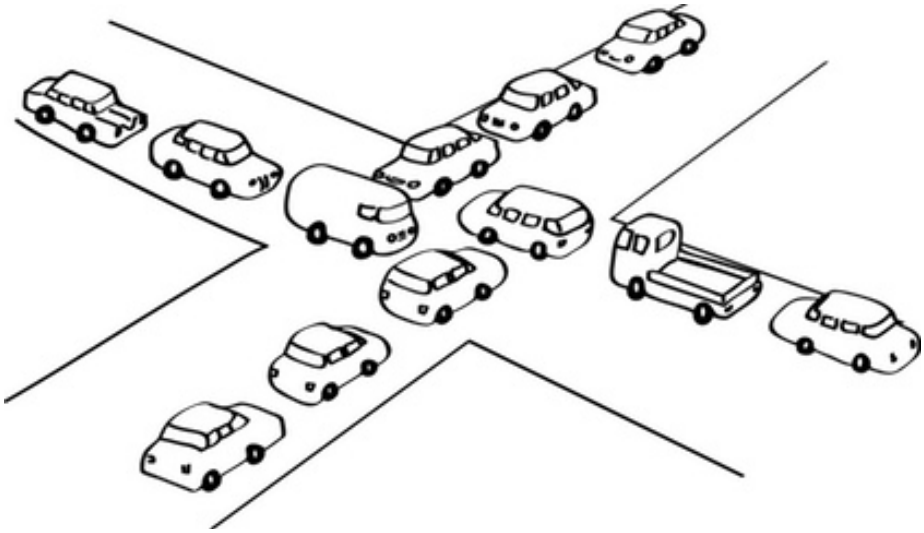


Abstract Parallel Programming

Roman Böhringer, June 2018

Table of contents

1 Introduction

1.1 Why Parallel Programming?

Moore's Law is still somehow true (Transistors keep getting smaller & faster), but heat and power have become the primary concern in modern computer architecture. Therefore we have smaller, more efficient processors and more processors (often in one package). Furthermore, CPUs are way faster than memory access and there are limits in inherent program's ILP so it is / was no longer affordable to increase sequential CPU performance.

1.2 Parallel vs. concurrent

The key concern of parallelism is to use extra resources to solve a problem faster. The key concern of concurrency is to correctly and efficiently manage access to shared resources. In practice, these terms are often used interchangeably.

(Parallel) Programming

2 Threads

Threads are independent sequences of execution. Multiple threads share the same address space, which is more vulnerable for programming mistakes because threads are not shielded from each other. But because of this, threads share resources and can communicate more easily. Furthermore, context switching between threads is efficient. No change of address space is needed, no automatic scheduling and therefore no saving / reloading of PCB (Process control block; data structure containing the information to manage the scheduling of a process) state. Multiple threads can share a single CPU or run on multiple CPUs.

In Java, there's the `java.lang.Thread` class for managing Threads. The most important methods are:

- `start`: method called to spawn a new thread (JVM calls run() on object)
- `interrupt`: freeze and throw exception to thread
- `join`: Wait for another thread (the target i.e. the thread we called `join` on) to terminate.
- `getId`: Gets the thread ID
- `getState`: Gets the thread state

There are two options for creating a thread in Java:

1. Instantiate a subclass of `java.lang.Thread` (must override `run()`):

```
class ConcurrWriter extends Thread { ...
      public void run() { ... }
```

```
}
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start(); // calls ConcurrWriter.run()
```

2. Implement a `java.lang.Runnable`, pass it to the thread object:

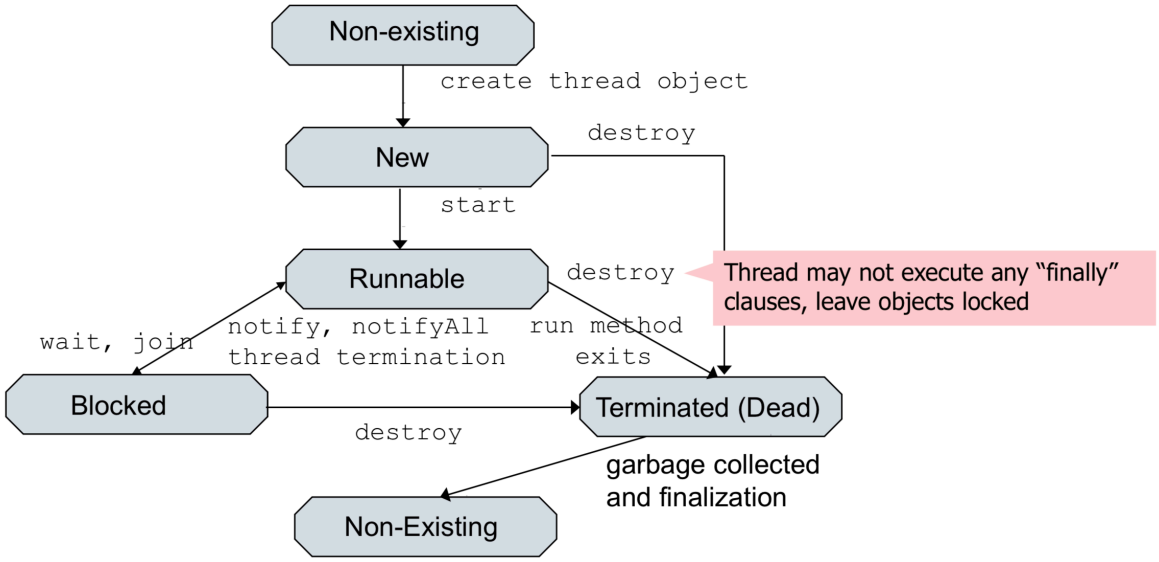
```
public class ConcurrReader implements Runnable { ...
      public void run() { ...
      ... code here executes concurrently with caller ... }
}
ConcurrReader readerThread = new ConcurrReader();
Thread t = new Thread(readerThread);
t.start(); // calls ConcurrReader.run() automatically
```

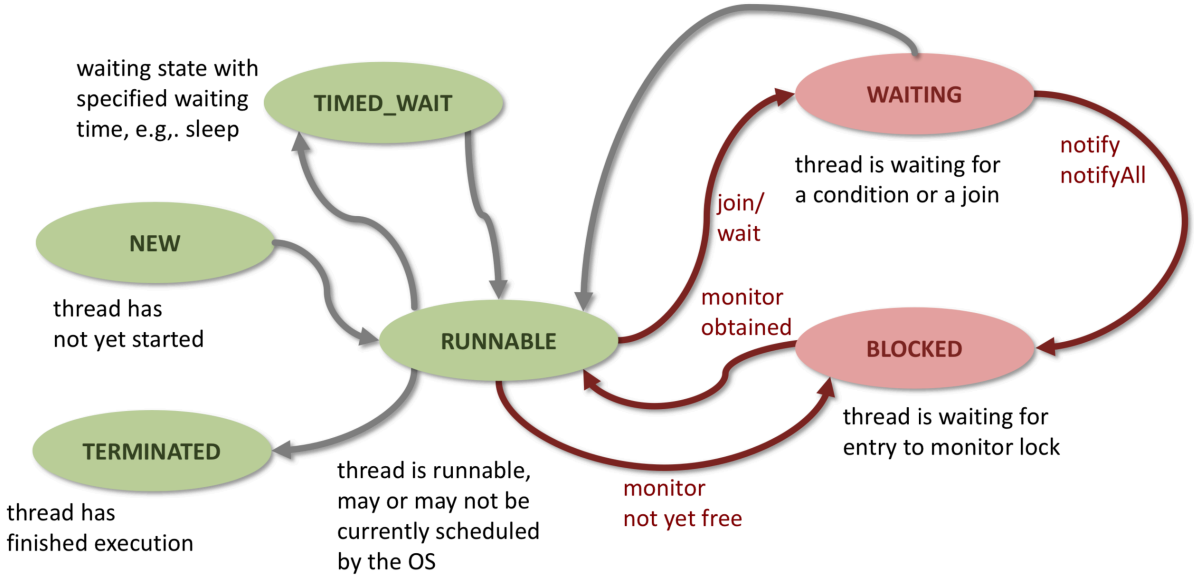
Every Java program has at least one execution thread (first execution thread calls the main function). Each call to start method of a Thread object creates a new thread, only creating the object doesn't start it. Calling `run()` doesn't start the thread either!

2.1 Thread safety

Informally, thread safety implies program correctness. It is the guarantee that nothing bad ever happens regardless of thread interleavings!

2.2 Thread states





3 Synchronization in Java

All Objects in Java have an internal lock (inherited from `java.lang.Object`). A synchronized operation locks the object (only with respect to locks on same object, other code that doesn't use locks can still modify the lock object or the protected object). The `synchronized` classifier in a method declaration is shorthand for a `synchronized (this) {…}` block surrounding the whole method. Java locks are reentrant, i.e. a thread can request to lock an object it has already locked. `synchronized` on a static method does lock the whole class, i.e. `public static synchronized getId() {…}` is the same as a `synchronized(getId.class) {…}` around the whole method.

3.1 Exceptions

When an exception inside a synchronized block is triggered, the lock is released and then the exception is caught and the exception handler is executed (if there is no exception handler, the exception is propagated back as usual).

3.2 Operations on the Monitor

Locks in Java are really Monitors. Therefore, they support the usual operations:

- `wait()`: Releases object lock, thread waits on internal queue
- `notify()`: Wakes the highest-priority thread closest to front of object's internal queue
- `notifyAll()`: Wakes up all waiting threads. The threads compete (non-deterministically for access to object, may not be fair)

These methods can only be called inside a `synchronized` block. `notify` / `notifyAll` don't release the lock, therefore the woken up thread can't run until the code which called `notify` releases its lock.

When we check a condition and `wait` if the condition is not fulfilled, we should always check the condition in a `while`-loop (not using an `if`-statement) because the thread can return from a `wait` call for various reasons (e.g. because he got interrupted) and we cannot be sure that the condition is fulfilled after he returned.

Furthermore, we have to take care that we do not hold another lock when calling wait (could lead to a deadlock because the other lock is still held).

3.3 Interface Lock

Javas intrinsic locks (used by `synchronized`) have some limitations. There is only one implicit lock per object, they are forced to be used in blocks and provide limited flexibility. For more flexibility, Java offers the Lock interface:

```
final Lock lock = new ReentrantLock();
```

Java Locks provide conditions that can be instantiated:

```
Condition notFull  = lock.newCondition();
```

They offer the `await()`, `signal()` and `signalAll()` methods.

There is also `java.util.concurrent.locks.ReentrantReadWriteLock` for a Reader / Writer lock.

3.4 Atomic operations

We have `java.util.concurrent.atomic.AtomicBoolean` (also for other data types) with the operations `set()`, `get()`, `compareAndSet(boolean expect, boolean update)` and `getAndSet(boolean newValue)`.

`compareAndSet(boolean expect, boolean update)` sets the Boolean to `update` when the current value is still `expect` (and returns true, false otherwise). `getAndSet(boolean newValue)` sets the Boolean to `newValue` and returns the current value.

The direct mapping to hardware of these operations is not guaranteed, meaning they are not guaranteed lock-free.

4 Executor Service

Because Java threads are quite heavyweight (in many implementations mapped to OS threads), using one thread per small task is highly inefficient. The alternative approach is to schedule tasks on threads in a thread pool. In Java, we use `ExecutorService` for that. The programmer submits a task to the ExecutorService (a `Runnable` that doesn't return a result or a `Callable` that returns a result) and gets back a `Future`.

`ExecutorService` isn't suited well for the divide and conquer approach. In a typical parallel divide and conquer program, a tasks spawns another tasks and waits for the result (the

spawned task spawns another one and so on…). In a short time, the thread pool is filled with tasks waiting for other tasks which can't run because the thread pool is filled.

5 ForkJoin framework

The ForkJoin framework is Javas implementation of the Fork / Join idiom. It is very similar to an Executor Service, but uses a work stealing scheduler. Because of this, it is well suited for divide and conquer types of programs. While threads in the pool are waiting for the results of their own spawned tasks, they can "steal" tasks from the other threads queues – therefore helping them progress and vice-versa.

To use the ForkJoin framework, we need to create a `ForkJoinPool` and then submit a `RecursiveTask<V>` to it which in turn spawns new `RecursiveTask<V>` by calling `fork()`. To wait for the task result, `join()` is called on the `RecursiveTask<V>`. In a `RecursiveTask<V>`, we need to override the `compute()` method.

6 Java Memory Model

The compiler and the hardware is allowed to make changes that do not affect the semantics of a sequentially executed program. Therefore, modern compilers do not give guarantees that a global ordering of memory accesses is provided. The memory model of a programming language provides certain (often minimal) guarantees for the effects of memory operations.

When we use `volatile` fields in Java, accesses do not count as data races. In detail, `volatile` in Java means:

- Every read / write to the field goes straight to main memory, there are no cache incoherences between different cores (for this field).
- There is a "happens-before" guarantee for volatile fields:
    - Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable. The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.
    - Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.

Parallelism

7 Approaches to parallelism

7.1 Implicit / explicit parallelism

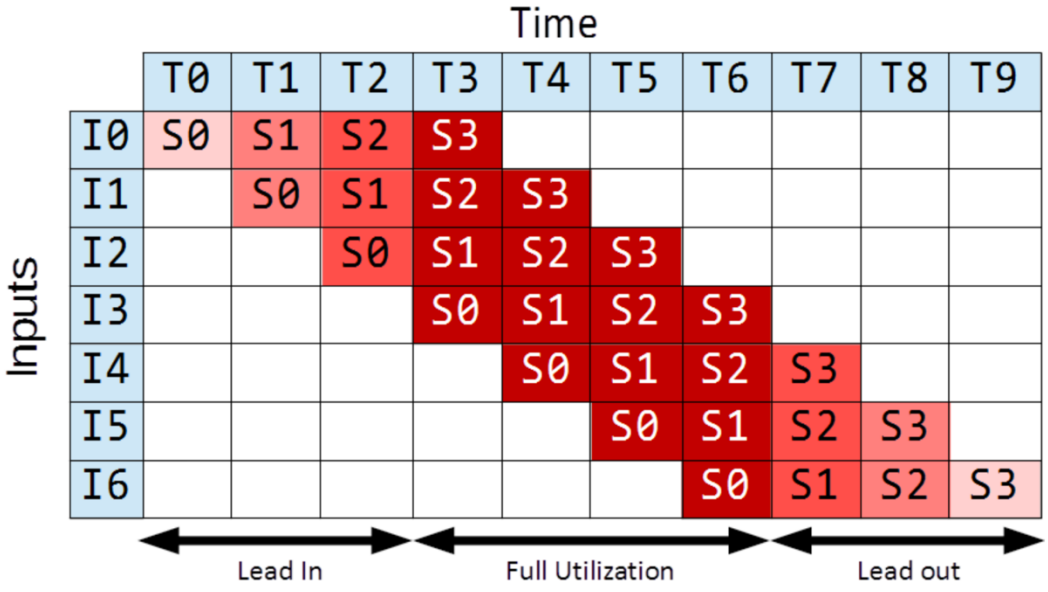
Implicit parallelism is when the programmer doesn't explicitly specify the parallelism. It is the job of the compiler or the runtime / OS to identify potential parallelism (e.g. independent memory access) and exploit it. Explicit parallelism is when the programmer explicitly specifies the parallelism, e.g. by creating a number of threads to solve a specific problem.

7.2 Vectorization

Vectorized operations apply an operation to an entire set of values. Whereas we normally load 1 value, modify it and store it back, in vectorized code we load multiple values (using special registers), apply the modification to all the values and store all values back in memory.

7.3 Pipelining

In a pipeline, we divide instructions in sequential steps and use multiple functional units to process these steps in parallel. A typical pipeline looks like this:



The **throughput** of the pipeline is the amount of work that can be done by a system in a given period of time (CPU: number of instructions per second). The throughput is bounded by the longest stage and we can approximate it (ignoring lead-in / lead-out time) by:

$$\frac{1}{\max\left(computationtime(stages)\right)}$$

The **latency** is the time to perform a computation (e.g. to execute a CPU instruction). The latency is only constant over time if the pipeline is balanced and is then the sum of execution times of each stage. In an unbalanced pipeline, the latency grows:

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

Pipelining often increases the latency, but significantly improves the throughput of a system.

7.4 Instruction Level Parallelism

ILP is a form of implicit parallelism. The CPU executes the code in parallel but to the programmer it appears that the code was executed sequentially. There are multiple forms of ILP:

- Pipelining
- Superscalar CPUs (multiple instructions per cycle)
- Out-of-Order (OoO) execution (reorder program)
- Speculative execution (predict results)

7.5 Architectures

7.5.1 Simultaneous Multithreading (SMT), Intel Hyperthreading

In SMT, there are multiple instruction streams (“virtual cores”) per core that are exposed to the OS. The processor schedules the instructions from the streams and can take advantage of the superscalar concept (instructions from different threads can get issued in the same cycle on the same core).

7.5.2 Multicores

A single processor has multiple cores with its own hardware units. The cores may share part of the cache hierarchy.

7.5.3 Symmetric Multi Processing (SMPs)

There are multiple CPUs on the same system and they share the memory. There has to be a cache coherence protocol to ensure that all the caches are coherent.

7.5.4 Non-Uniform Memory Access (NUMA)

The memory is distributed i.e. each processor has its own local memory but can access the memory of the other CPUs. The access time depends on if the data is in the local memory or in the memory of a different processor.

7.5.5 Distributed Memory

Distributed Memory is used by large clusters. There is an interconnect network between the machines and often time Message Passing (e.g. MPI) is used.

7.6 Expressing parallelism

7.6.1 Work partitioning / scheduling

Work partitioning to split up work of a single program into parallel tasks. It can be done explicitly / manually (task / thread parallelism) or automatically / implicit (data parallelism). Work partitioning is also called task / thread decomposition.

Scheduling is to assign tasks to processors. This is usually done by the system and the goal is full utilization (i.e. no processor is ever idle).

When the program is split up in many small tasks, we talk about fine granularity. On the other hand, if it is split up in a few big tasks, it is called coarse granularity. Fine granularity is more portable (can easily be executed in machines with more processors), better for scheduling but if the scheduling overhead is comparable to a single task, the overhead dominates. The guideline for granularity is to keep the tasks as small as possible but significantly bigger than scheduling overhead.

7.6.2 Scalability

Scalability means how well a system reacts to increased load. In parallel programming, when we talk about scalability we mean the speedup when we increase processors and we look what happens when processors go to infinity. If a program scales linearly, we have linear speedup.

When comparing parallel performance, we have a sequential execution time T_1 as a base measure. We then measure the execution time T_p on p CPUs and distinguish three cases:

- $T_p = T_1/p$ (perfection)
- $T_p > T_1/p$ (performance loss, normal case)
- $T_p < T_1/p$ (should only happen in special cases)

The (parallel) speedup S_p on p CPUs is defined as: $S_p = T_1/T_p$

We can distinguish the three cases again:

- $S_p = p$ → linear speedup (perfection)
- $S_p < p$ → sub-linear speedup (performance loss)
- $S_p = p$ → super-linear speedup (should only happen in special cases)

We get normally a sub-linear speedup because the programs may not contain enough parallelism, because of overheads introduced by parallelization (oftentimes associated with synchronization) or / and because of architectural limitations (e.g. memory contention).

7.6.2.1 Amdahl's Law

If we divide the execution time into two categories:

- W_{ser} = Time spent doing non-parallelizable serial work
- W_{par} = Time spent doing parallelizable work

We get that $T_1 = W_{ser} + W_{par}$ and get a lower bound for T_p: $T_p \geq W_{ser} + \frac{W_{par}}{W_{ser}}$

Therefore, we have this upper bound for the speedup:

$$S_p \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{p}}$$

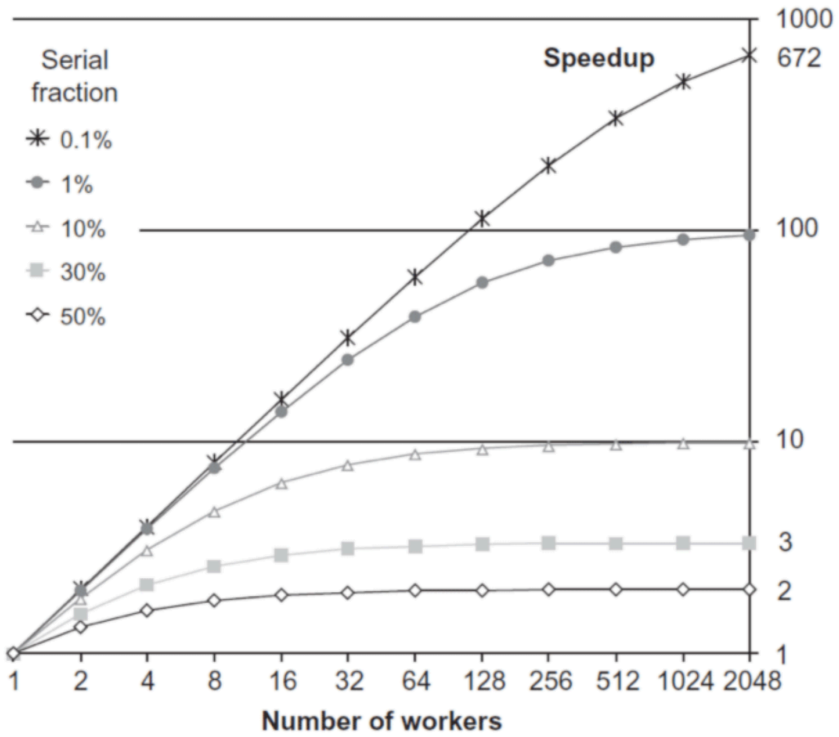
If f is the non-parallelizable serial fraction of the total work, we have: $W_{ser} = fT_1$ and $W_{par} = (1-f)T_1$. This gives:

$$S_p \leq \frac{1}{f + \frac{1-f}{p}}$$

Therefore, if we have infinite workers:

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

The following diagram illustrates the main takeaway from Amdahl's Law:



7.6.2.2 Gustafson's Law

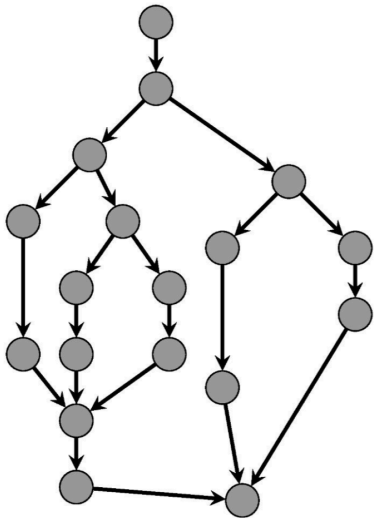
In Gustafson's Law, the run-time is constant but the problem size isn't. More processors allow to solve larger problem in the same time because the parallel part of a program scales with the problem size. If we have a task with a sequential part f, the speedup is:

$$S_p = f + p(1 - f)$$
$$= p - f(p - 1)$$

7.6.3 Parallel programming paradigms

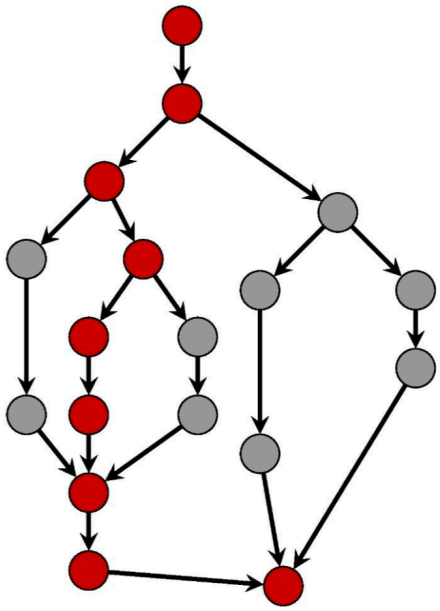
7.6.3.1 Task Parallel

In the task parallel paradigm, the programmer explicitly defines parallel tasks. In Cilk[1]-Style task parallel programming, tasks execute code, spawn other tasks and wait for results from other tasks (fork-join idiom). A task graph (directed acyclic graph) is formed, where an edge means that the head of the edge created the tail of the edge.

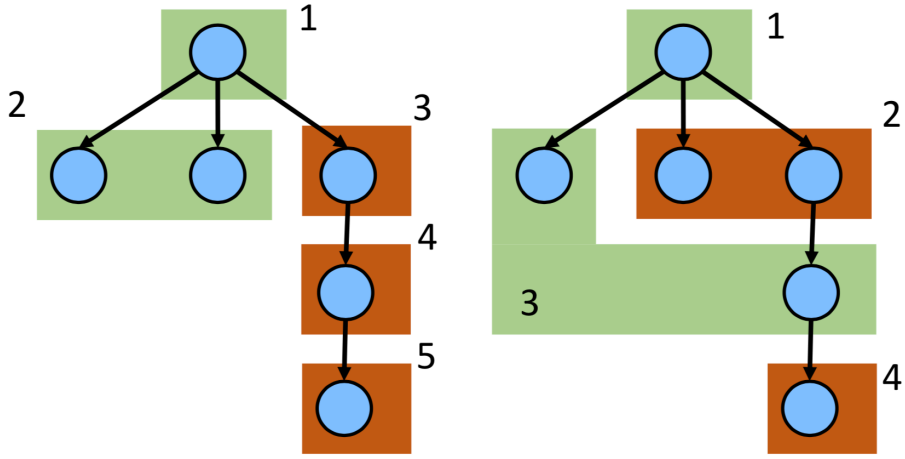


Tasks can be executed in parallel, but they don't have to. It is up to the scheduler to assign the tasks to CPUs / cores. The task graph is dynamic because it unfolds as execution proceeds Intuitively, a wider graph means more parallelism. T_∞ is the time it takes on infinite processors and is called span / critical path in this context. It is the longest path from root to sink in the task graph:

[1] A Programming language from MIT, designed for multithreaded parallel computing



T_p depends on the scheduler, whereas T_∞ (and T_1 and p) is fixed. In this example, we can see the difference of T_p based on the scheduling:



T_2 will be 5 with this scheduling

T_2 will be 4 with this scheduling

For a greedy scheduler (in a nutshell, a greedy scheduler is a scheduler in which no processor is idle if there is more work it can do,) we have:

$$T_p \leq \frac{T_1}{p} + T_\infty$$

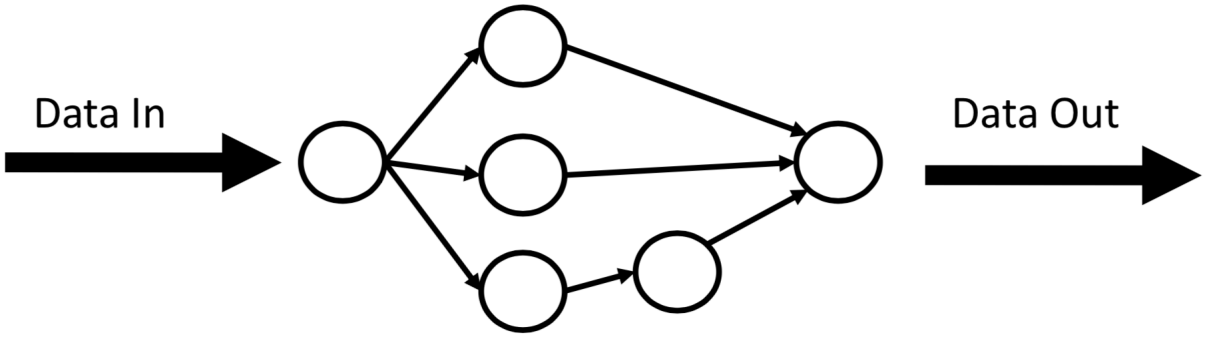
MIT's Cilk introduced the work stealing scheduler. In a work stealing scheduler, each processor in a computer system has a queue of work items (computational tasks, threads) to perform. Each work item consists of a series of instructions, to be executed sequentially, but in the course of its execution, a work item may also spawn new work items that can feasibly be

executed in parallel with its other work. These new items are initially put on the queue of the processor executing the work item. When a processor runs out of work, it looks at the queues of other processors and "steals" their work items. In effect, work stealing distributes the scheduling work over idle processors, and as long as all processors have work to do, no scheduling overhead occurs. For a work stealing scheduler, the lower bound of a greedy scheduler is usually achieved, i.e. we have:

$$T_p = \frac{T_1}{p} + O(T_\infty)$$

7.6.3.2 Data parallel (dataflow programming model)

In the dataflow programming model, the programmer defines what each task does and how the tasks are connected.



Dataflow programming is declarative because the programmer only describes what, but not how. The work partitioning is done by the system.

An important dataflow programming technique / library is MapReduce. A reduction is an operation that produces a single answer from a collection via an associative operator (e.g. max, count, leftmost, sum, product, …). A map operates on each element of a collection independently to create a new collection of the same size. In the MapReduce framework, the programmer just writes the mappers / reducers and the system takes care of distributing the data / managing fault tolerance.

8 Distributed memory / message passing

The basic idea of distributed memory / message passing is to have isolated mutable state i.e. state is mutable, but not shared (each thread has its private state). Tasks cooperate via message passing. With distributed memory, state is distributed and threads exchange amounts at coarse granularity. We differentiate between two message types:

- Synchronous messages: Sender blocks until message is received
- Asynchronous messages: Sender does not block (fire-and-forget), message is placed into a buffer for receiver to get

There are multiple programming models for concurrent message passing.

8.1 Programming models

8.1.1 CSP: Communicating Sequential Processes

CSP is a formal language defining a process algebra for concurrent systems. Functions can be linked with the operators seq (sequential) or par (parallel). Communication / Synchronization is done with message passing but naming is indirect (there are symbolic channels between sender and receiver). CSP was first implemented in Occam and the Go programming language was inspired by CSP. An example of a Go application is:

```
func main() {

        msgs := make(chan string)
        done := make(chan bool)

        go hello(msgs, done);

        msgs <- "Hello"
        msgs <- "bye"

        ok := <-done

        fmt.Println("Done:", ok);
}
```

Create two channels:
•msgs: for strings
•done: for boolean values

```
func hello(msgs chan string,
           done chan bool) {

        for {
                msg := <-msgs
                fmt.Println("Got:", msg)

                if msg == "bye" {
                        break
                }
        }

        done <- true;
}
```

8.1.2 Actor programming model

An actor is a computational agent that maps communication to:

- a finite set of communications sent to other actors (messages)
- a new behavior (state)
- a finite set of new actors created (dynamic recofigurability)

Global ordering is undefined and asynchronous message passing is used. The graph is dynamically configured during runtime and resources are dynamically allocated. Actor sends messages to other actors using “direct naming” (no indirection via e.g. ports / channels / queues / sockets, etc…).

The actor model is implemented in various languages such as Erlang, Scala, Ruby or Akka (Scala and Java). The model is sometimes also called event-driven programming model because often a program is written as a set of event handlers for events.

A sample erlang application looks like this:

```
start() ->
       Pid = spawn(fun() -> hello() end),
Pid ! hello, Pid ! bye.
hello() -> receive
hello ->
io:fwrite("Hello world\n"),
                hello();
```

```
          bye ->
io:fwrite("Bye cruel world\n"), ok
end.
```

8.2 Message passing libraries

In the 1980s, PVM (Parallel Virtual Machines) was created as a library for message passing. In the 1990s, MPI (Message Passing Interface) was created which is widely used today.

8.2.1 MPI (Message Passing Interface)

MPI processes are collected into groups. Each groups can have multiple colors (sometimes called the context). The group + the color is the communicator (like a name for the group). When an MPI application starts, the group of all processes is initially given the predefined name `MPI_COMM_WORLD`. Within each communicator, a process is identified by a unique number called rank. For two different communicators, the same process can have two different ranks. The communicators define the communication domain of a communication operation i.e. the set of processes that are allowed to communicate with each other. A communicator does not need to contain all the processes in the system.

MPI programs follow the SPMD (Single Program, Multiple Data) pattern. One program is compiled that works on multiple data. The most important MPI functions are:

- `MPI_INIT` – initialize the MPI library (must be the first routine called)
- `MPI_COMM_SIZE` - get the size of a communicator
- `MPI_COMM_RANK` – get the rank of the calling process in the communicator
- `MPI_SEND` – send a message to another process
- `MPI_RECV` – receive a message from another process
- `MPI_FINALIZE` – clean up all MPI state (must be the last MPI function called by a process)

When sending messages, message tags can be used to differentiate between different messages being sent. Furthermore, a receiver can “filter” messages so that he only receives from the source he specified and the sender can specifiy the destination in addition to the communicator.

Synchronous send (Ssend) waits until complete message can be accepted by receiving process before completing the send whereas asynchronous send (send) does not wait for actions to complete before returning and therefore needs a buffer. Receive is always synchronous. Send and receive are blocking by default. Blocking / Nonblocking is about local handling of data to be sent / received, a blocking call can only return after local actions are complete (though the message transfer may not have been completed; depending on synchronous / asynchronous) whereas nonblocking calls return immediately.

The following code is called “unsafe”:

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

It depends on the availability of system buffers in which to store the data sent until it can be received and can therefore lead to a deadlock. It can be solved by reordering the messages, explicitly supplying a buffer, sending / receiving at the same time or by using non-blocking operations and calling `waitall` afterwards.

A sample application to compute pi looks like this:

```
MPI.Init(args);
… // declare and initialize variables (sum=0 etc.)
int size = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();

for(int i=rank; i<numSteps; i=i+size) {
   double x=(i + 0.5) * h;
   sum += 4.0/(1.0 + x*x);
}

if (rank != 0) {
   double [] sendBuf = new double []{sum};
   // 1-element array containing sum
   MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 10);
}
else { // rank == 0
   double [] recvBuf = new double [1] ;
   for (int src=1 ; src<P; src++) {
      MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 10);
      sum += recvBuf[0];
   }
}
double pi = h * sum; // output pi at rank 0 only!
MPI.Finalize();
```

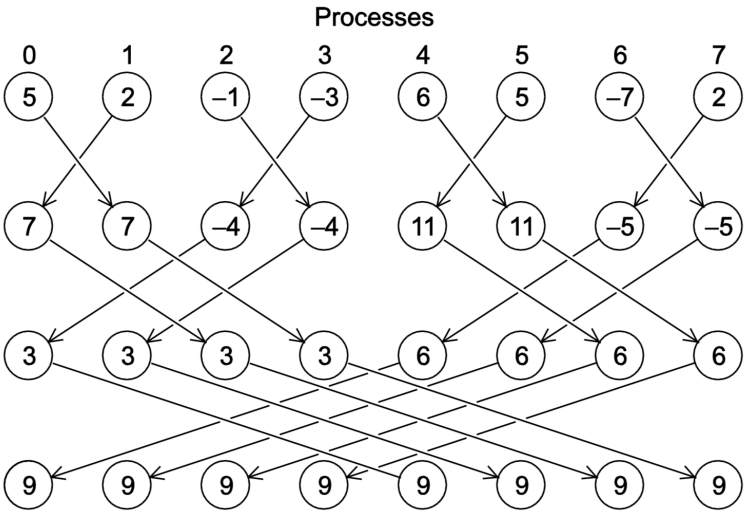
As we can see, each process calculates depending on its rank and the total number of processes (size). All processes except the one with rank 0 send the result, whereas the process with rank 0 receives all results and sums them together.

MPI applications are started using `mpiexec`, e.g. `mpiexec -np 16 ./test`.

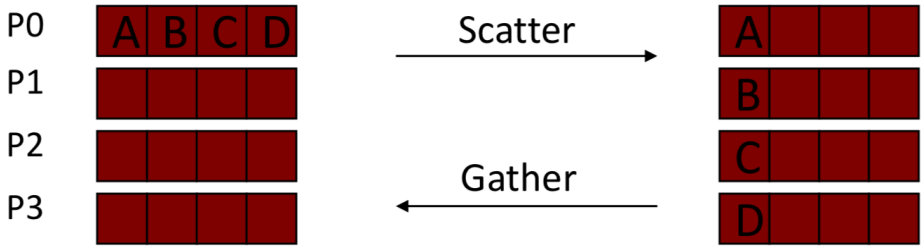
8.2.1.1 Collective communication

Besides point-to-point communication, MPI also supports communications among groups of processors. The most important operations are:

- `Reduce`: Implemented in a tree-structure. Process 1 sends to 0, 3 to 2, … Then process 2 sends their reduced value to 0, 6 to 4, … Finally process 4 sends its newest value to process 0.
- `Broadcast`: Implemented in a tree-structure as well. Process 0 sends to 4, which starts broadcasting to other nodes as soon as the value arrives…
- `Allreduce`: Allreduce is when not only one process needs the reduced value (e.g. the sum) but rather all need it. Allreduce is implemented in a "butterfly structure":



- `Scatter / Gather`: Scatter is the operation of splitting a vector and sending the components to different processes, whereas gather collects the components and forms a fector:



Concurrency

9 Managing state

Managing state is the main challenge for parallel programs. There are three approaches:

- Immutability (data does not change, best option when possible)
- Isolated mutability (data can change, but only one thread / task can access them)
- Mutable / shared data (data can change and all threads / tasks can access them)

Mutable / shared data is present in shared memory architectures, but concurrent accesses may lead to inconsistencies. Therefore we need to protect state by allowing only one task / thread to access it at a time (in general). Intermediate inconsistent states should not be observed. The methods to ensure mutual exclusion are locks and transactional memory (see later chapters).

9.1 Mutual exclusion

A critical section is a piece of code that may be executed by at most one process at a time with the following conditions:

- Mutual exclusion (statements from critical sections of two or more processes must not be interleaved)
- Freedom from deadlock: If some processes are trying to enter a critical section, one of them must eventually succeed
- Freedom from starvation[2]: If any process tries to enter its critical section, then that process must eventually succeed

Mutual exclusion is an algorithm to implement a critical section. The required properties for mutual exclusion are:

- Safety property: At most one process executes the critical section code
- Liveness: acquire_mutex must terminate in finite time when no process executes in the critical section

9.2 Atomicity

An operation is atomic if no other thread can see it partly executed (“appears indivisible”).

9.2.1 Atomic operations

Most modern hardware supports atomic instructions. We differentiate TAS (Test-and-set) and CAS (Compare-and-set). The semantics of them are:

[2] Starvation is the repeated but unsuccessful attempt of a recently unblocked process to continue its execution.

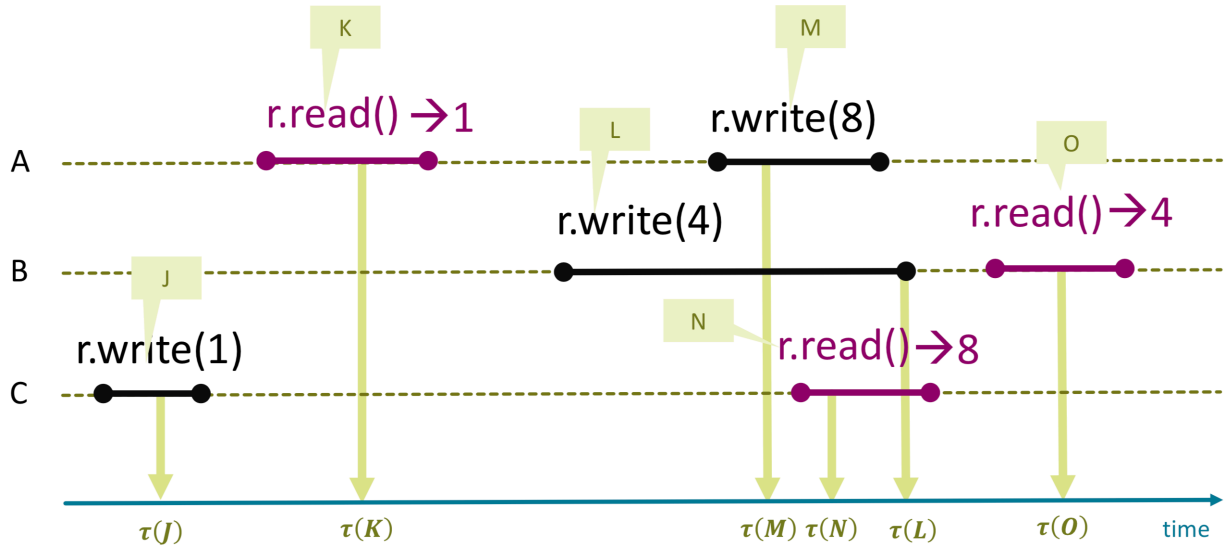
```
boolean TAS(memref s)
atomic
    if (mem[s] == 0) {
        mem[s] = 1;
        return true;
    } else
        return false;
```

```
int CAS (memref a, int old, int new)
atomic
    oldval = mem[a];
    if (old == oldval)
        mem[a] = new;
    return oldval;
```

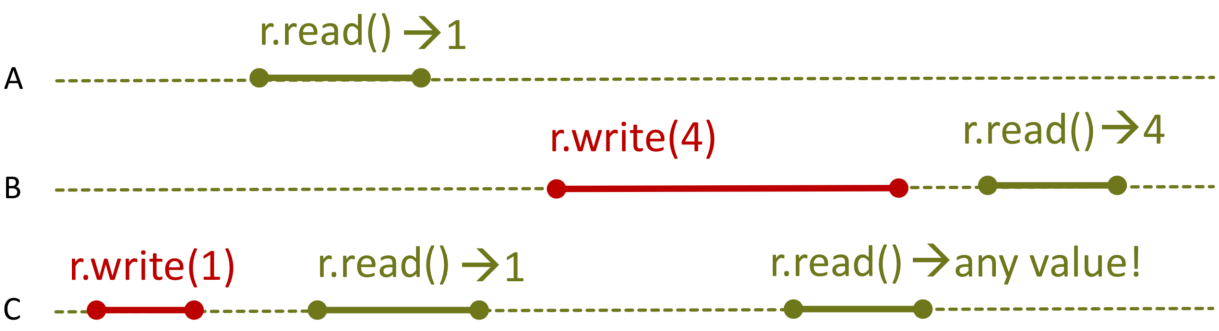
9.3 Types of registers

We differentiate between atomic, safe and regular registers:

- **Atomic register:** An invocation of read / write takes effect at a single point in time. This point lies between start and end of the operation. Two operations on the same register always have a different effect time. An invocation of read returns the value written by the invocation of write with closest preceding time:



- **Safe Register:** A Safe SWMR (Single Writer Multiple Reader) allows only one concurrent write but multiple concurrent reads. Any read not concurrent with a write returns the current value of the register. Any read concurrent with a write can return any value of the domain of r.



- **Regular Register:** Like a Safe Register, but any read concurrent with a write will either read the new or the old value (but not consistent, can sometimes return the old and then the new value).

10 Races

10.1 Data races

A data race is an erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. simultaneous read/write or write/write of the same memory location.

As a guideline, you should never allow two threads to read/write or write/write the same location at the same time. Do not make any assumptions on the orders of reads or writes.

10.2 Interleavings

If a second call starts before the first ends, we say the calls interleave (can also happen with one processor when a thread gest preempted / a context switch happens).

A bad interleaving is an erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources.

11 Locks

Locks are primitives with atomic operations:

- new: Make a new lock, initially “not held”
- acquire: Blocks if this lock is already currently “held”
- release: Makes this lock “not held” (if more than 1 threads are blocked on the lock, exactly one will acquire it)

A re-entrant lock (recursive lock) stores the thread that currently holds it and a count. If the current holder calls acquire, it does not block but increments the count. On release, the count is decremented and if the count is 0, the lock becomes not held.

“Locking” / mutual exclusion is very easy with a single core. The only way to get bad interleavings is when the scheduler decides to schedule another thread while a thread is in its critical section. To do that, an interrupt request (IRQ) is emitted. It is possible to switch off IRQs before entering the critical section and to switch them back on after leaving the critical section. Then it isn’t possible to interrupt the thread while in its critical section.

In the following sections, some algorithms to implement mutual exclusion with multiple threads are described. Decker’s Algorithm / Petersons’s Algorithm and the Filter lock require an atomic register, whereas the bakery locks also works with an SWMR register.

In practice, mutual exclusion is not implemented like this. There’s a theorem that states that a mutual exclusion protocol must have at least as many variables as processes. Therefore,

implementing mutual exclusion like this would require much storage and would be inefficient. Because of this, atomic hardware operations are used for implementing mutual exclusion.

11.1 Decker's Algorithm

Decker's Algorithm is an algorithm to ensure mutual exclusion with two processes. It uses two flags (indicating that the other process wants to enter the critical section) and a variable turn to indicate which thread is allowed to enter the critical section:

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn!=1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

and try again

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

Only using a turn variable wouldn't work because we haven't made any assumptions about progress outside of the critical section i.e. process 1 (when the turn variable is initialized to 0) could wait forever because process 0 isn't doing any progress and therefore not arriving at the critical section.

11.2 Peterson Lock

Peterson Lock is another way to ensure 2 process mutual exclusion. We again have two flags and a turn or victim variable

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

```
Process P (1)
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

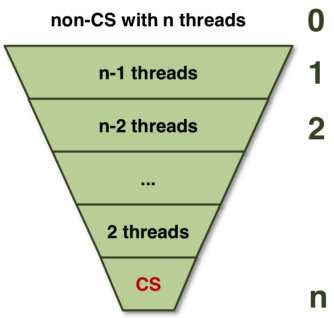
```
Process Q (2)
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

11.3 Filter Lock

The Filter Lock is an extension of Peterson's Lock to n processes. The idea is that every thread knows his level in the filter `level[t]`. In order to enter the critical section, a thread has to elevate all levels. For each level, we use Petersons's mechanism to filter at most one thread if other threads are at higher level. For every level, there is one `victim[l]` that has to let other pass in case of conflicts.

The Filter Lock

```
int[] level(#threads), int[] victim(#threads)


lock(me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```



Other threads are at same or higher level

And I have to wait

The Filter Lock is not fair, it is first-come-first-serve.

11.4 Bakery algorithm

The bakery algorithm works like the numbering system in a postal office. Every thread has a label indicating when he is allowed to enter the critical section (when he has the lowest label). It is possible that multiple processes have the same label, in which case the thread id gets compared.

```
integer array[0..n-1] label = [0,...,0]
boolean array[0..n-1] flag =  [false, ..., false]
```

SWMR «ticket number»

SWMR «I want the lock»

```
lock(me):
   flag[me] = true;
   label[me] = max(label[0], ... , label[n-1]) + 1;
   while (∃k ≠ me: flag[k] && (k,label[k]) <_l (me,label[me])) {};

unlock(me):
   flag[me] = false;
```

$(k, l_k) <_l (j, l_j) \Leftrightarrow l_k < l_j \text{ or } (l_k = l_j \text{ and } k < j)$

11.5 Spinlock

It is very easy to implement a spinlock using TAS:

```
Init(lock):
      lock = 0;
Acquire(lock):
      while !TAS(lock); // wait
Release(lock):
      lock = 0;
```

If we have a `CAS(memref a, int old, int new)` operation, it is also very easy to implement a spinlock:

```
Init(lock):
      lock = 0;
Acquire(lock):
      while (CAS(lock, 0, 1) != 0);
Release(lock):
      CAS(lock, 1, 0); // result gets ignored
```

11.5.1 Test-and-Test-and-Set (TATAS) Lock

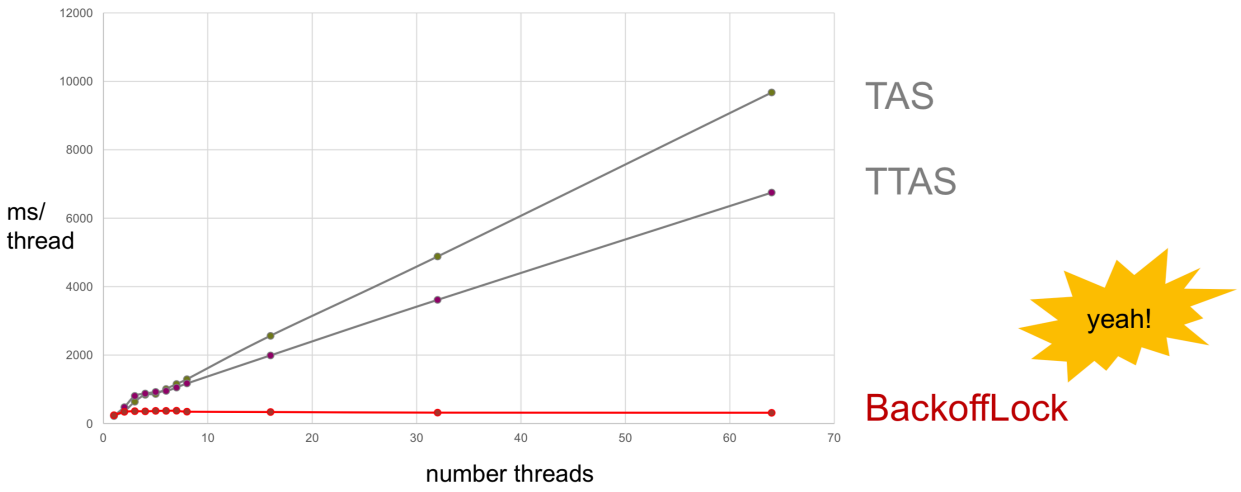
The performance of the spinlock described above is quite poor for many threads, because all threads fight for the bus (memory bus) during the call of getAndSet() and the cache coherency protocol needs to invalidate cached copies of the lock on other processors. Therefore, we can improve performance by only calling getAndSet / compareAndSet when we noticed that the lock is available (i.e. testing first):

```
public void lock()
{
    do
        while(state.get()) {}
    while (!state.compareAndSet(false, true));
}
```

11.5.2 TATAS with backoff

In a TATAS lock, there are still too many threads fighting for access to the same resource. Therefore we set a thread to sleep for a random duration when the acquisition of the lock fails. With an exponential backoff, we double the duration every time the acquisition fails.

This leads to a heavy improvement in performance:



11.6 Locks with waiting / scheduling

Spinlocks have some problems. They aren't fair (missing FIFO behavior), computing resources are wasted which degrades performance (especially for long-lived contention) and there is no notification mechanism. Scheduled locks are locks that suspend the execution of threads while they wait and solve these problems. Semaphores, mutexes and monitors (normally hybrid, while threads are in the "waiting entry" queue a spinlock is used to let only one thread in to the critical section, while they are in the "waiting condition" queue they are suspended) are typically implemented using a scheduled lock.

Scheduled locks require support from the runtime system (OS / scheduler) and the data structures for them need to be protected against concurrent access (using spinlocks or lock-free). They often have a higher wakeup latency and there are hybrid solutions (try access with spinlock for a certain duration before rescheduling).

11.7 Reader / Writer locks

Multiple concurrent reads of the same memory location are not a problem, whereas multiple concurrent writes and multiple concurrent read & writes of the same memory location are. Furthermore, there are many applications that have many simultaneous read operations and only a few write operations. Therefore, there are reader / writer locks. This lock can have three states:

- not held
- held for writing: By one thread
- held for reading: By 1 or more thread

It can never be held for reading / writing at the same time. Therefore instead of the `acquire` / `release` operations, we have `acquire_write` (block if currently held for reading or held for writing, else set state to held for writing) / `release_write` (set state to not held) and `acquire_read` (block if currently held for writing, else set state to held for reading and increment readers count) / `release_read` (decrement readers count, if 0 set state to not held).

A simple monitor-based implementation looks like this:

```
class RWLock {
 int writers = 0;
 int readers = 0;

synchronized void acquire_read() {
 while (writers > 0)
  try { wait(); }
  catch (InterruptedException e) {}
 readers++;
}

synchronized void release_read() {
 readers--;
 notifyAll();
}
```

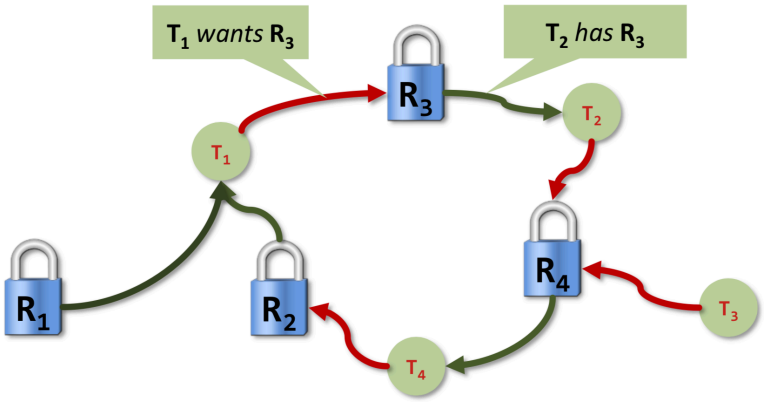
```
 synchronized void acquire_write() {
   while (writers > 0 || readers > 0)
    try { wait(); }
    catch (InterruptedException e) {}
   writers++;
 }

 synchronized void release_write() {
  writers--;
  notifyAll();
 }
}
```

This lock isn't fair because it gives priority to readers (they can just enter when other readers are reading). A "fairer" model would be to let a number k of currently waiting readers pass. When the k readers have passed, the next writer may enter (if any), otherwise further readers may enter until the next writer enters (who has to wait until the current readers finish).

11.8 Deadlocks

A deadlock occurs when two or more processes are mutually blocked because each process waits for another of these processes to proceed. More formally stated:

A deadlock for threads $T_1 \dots T_n$ occurs when the directed graph describing the relation of $T_1 \dots T_n$ and resources $R_1 \dots R_m$ contains a cycle.



Deadlock detection is implemented by finding cycles in the dependency graph. To avoid deadlocks, we can do two things:

- Two phase locking with retry (done when we can abort transaction without consequence, usually in databases)
- Resource ordering (usually in parallel programming). The whole program has to obey this order to avoid cycles.

11.8.1 Livelock

A livelock occurs when two or more processes can't make progress because they constantly switch between the same states (in contrast to the deadlock, where they are blocked).

11.9 Lock granularity

There are different approaches to lock granularity. Most of these concepts translate somehow to other data types / programs, but will be described based on a linked list.

11.9.1 Coarse grained locking

Coarse grained locking is very simple, but performance is poor. We just lock the whole object on all operations.

11.9.2 Fine grained locking

In the fine grained locking approach, we split the object into pieces with separate locks. Therefore we have no mutual exclusion for algorithms on disjoint pieces, which improves performance. When traversing a linked list, "hand over hand locking" is used in the fine grained locking approach. In hand over hand locking, we first lock the head and the predecessor of the head. Then the head is unlocked, the third node is locked, the predecessor is unlocked, the fourth node is locked and so on… With this method, in the end the predecessor and the node to be modified / deleted is locked and it is prevented that another thread changes the next pointer of the predecessor. The disadvantages of this method are that it can lead to a potentially long sequence of locking / unlocking before the intended action can take place and one slow thread locking "early nodes" can block another thread wanting to modify "late nodes" (→ no passing possible).

11.9.3 Optimistic synchronization

In optimistic synchronization search is performed without acquiring any locks at all. If the method finds the sought-after component, it locks that component, and then checks that the component has not changed in the interval between when it was inspected and when it was locked. This technique is only worthwhile when it succeeds more often than not, why it is called optimistic.

In the case of linked lists, it could happen that the sought-after node isn't reachable anymore (got deleted) or has a new successor. Therefore we need to validate that both isn't the case.

```
private Boolean validate(Node pred, Node curr) {
  Node node = head;
  while (node.key <= pred.key) { // reachable?
     if (node == pred)
           return pred.next == curr; // correct?
     node = node.next;
  }
  return false;
}
```

With optimistic synchronization, traversals are wait-free and there are less lock acquisitions. But we need to traverse the list twice and the `contains` method needs to acquire locks. Furthermore, optimistic synchronization is not starvation free (if new nodes are always added / removed, a thread can be delayed infinitely because validation always fails).

11.9.4 Lazy synchronization

In lazy synchronization, the task of removing a component from a data structure is split into two phases: The component is logically removed by setting a tag bit and later physically removed by unlinking it from the rest of the data structure.

In the case of linked lists, we use deleted markers and have the invariant that every unmarked node is reachable. When removing a node, we scan the list and lock predecessor and current (same as with the optimistic synchronization approach). Then we mark the current node as removed (logical deletion) and afterwards we redirect predecessors next (physical deletion).

Our remove routine now looks like this (without finding node & locking):

```
if (!pred.marked && !curr.marked &&
     pred.next == curr) {
   if (curr.key != key)
      return false;
   else {
      curr.marked = true;    // logically remove
      pred.next = curr.next; // physically remove
      return true;
   }
}
```

`contains` is very simple and wait-free:

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

12 Semaphores

Sometimes we need more than locks. Locks provide means to enforce atomicity via mutual exclusion but they lack the means for threads to communicate about changes (e.g. changes in the state).

A semaphore is an integer-valued abstract data type S with some initial value $s \geq 0$ and the following operations:

```
acquire(S)
atomic {
  wait until S > 0
  dec(S)
}
```

```
release(S)
atomic {
  inc(S)
}
```

It is very easy to build a lock with a semaphore. We just initiate it with value 1. If more than 1 thread should be let into the “critical section”, we initiate it with this value.

12.1 Rendezvous

It is easy to implement a Rendezvous (point in the program where two threads meet, i.e. wait for each other) using Semaphores:

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`release(Q_Arrived)` `acquire(P_Arrived)`
post	`...`	`..`

12.2 Implementation

A semaphore can be implemented without spinning by using a process list:

Consider a process list Q_S associated with semaphore S

```
acquire(S)
atomic {if S > 0 then
            dec(S)
        else
            put(Qs, self)
            block(self)
        end }

release(S)
atomic {if Qs == ∅ then
            inc(S)
        else
            get(Qs, p)
            unblock(p)
        end }
```

13 Barriers

Barriers are used when we want to synchronize a number of processes at one point in the program. An implementation of a reusable barrier using semaphores is provided below:

init

```
mutex=1; barrier1=0; barrier2=1; count=0
```

barrier

```
acquire(mutex)
  count++;
  if (count==n)
     acquire(barrier2); release(barrier1)
release(mutex)

acquire(barrier1); release(barrier1);
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
acquire(mutex)
  count--;
  if (count==0)
      acquire(barrier1); release(barrier2)
signal(mutex)

acquire(barrier2); release(barrier2)
// barrier2 = 1 for all processes, barrier1 = 0 for all processes
```

The two semaphores are needed because otherwise, processes could pass other processes and increment count while others are still “in” the barrier. Like this, the process has to wait until count is 0 and `release(barrier2)` is called.

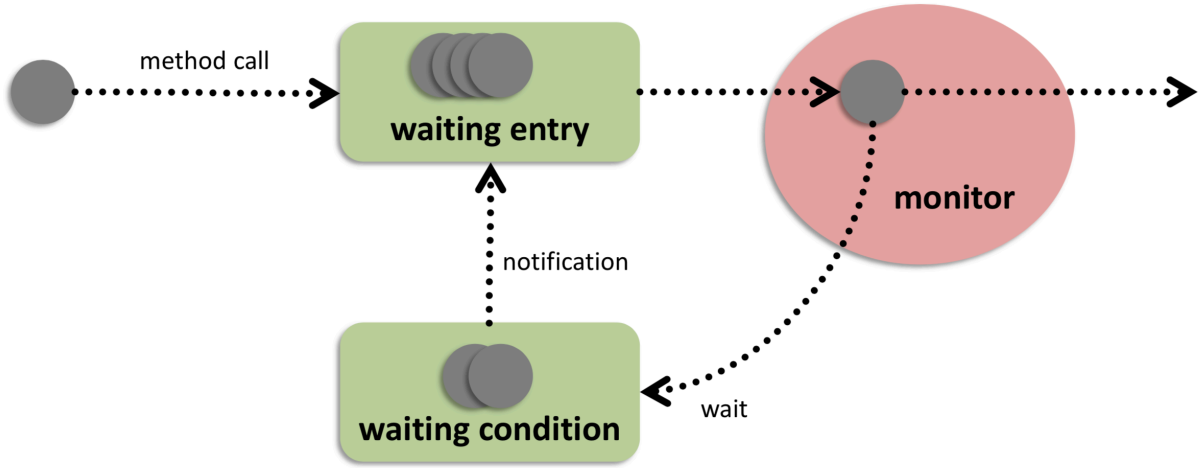
14 Monitors

Monitors provide, in addition to mutual exclusion, a mechanism to check conditions with the following semantics:

If a condition does not hold:

- Release the monitor lock
- Wait for the condition to become true
- Signaling mechanism to avoid busy-loops

A monitor has an associated monitor queue (similar to semaphores). The threads in the queue are separated in “waiting entry” (they are just waiting for the lock, but not on a condition) and “waiting condition” (waiting for a signal):



There are different monitor semantics:

- Signal and wait: Signaling process exits the monitor (goes to waiting entry queue) and passes the monitor lock to the signaled process.
- Signal and continue: Signaling process continues running and the signaled process is moved to waiting entry queue. (implemented by Javas Monitors)

15 Lock-free programming

Locks have several disadvantages. They are pessimistic by design (assume the worst and enforce mutual exclusion), have performance issues (overhead for each lock taken even in unconteded case, contended case leads to significant performance degradation) and they are blocking. If a thread is delayed in a critical section, all threads have to wait. If a thread dies in the critical section, it's even worse and they are prone to deadlocks. Lock-free programming tries to overcome these issues. We define the following progress conditions:

- Lock-freedom: At least one thread always makes progress even if other threads run concurrently. Implies system wide progress but not freedom from starvation.
- Wait-freedom: All threads eventually make progress. Implies freedom from starvation (and implies lock-freedom).

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone make progress	Lock-free	Deadlock-free

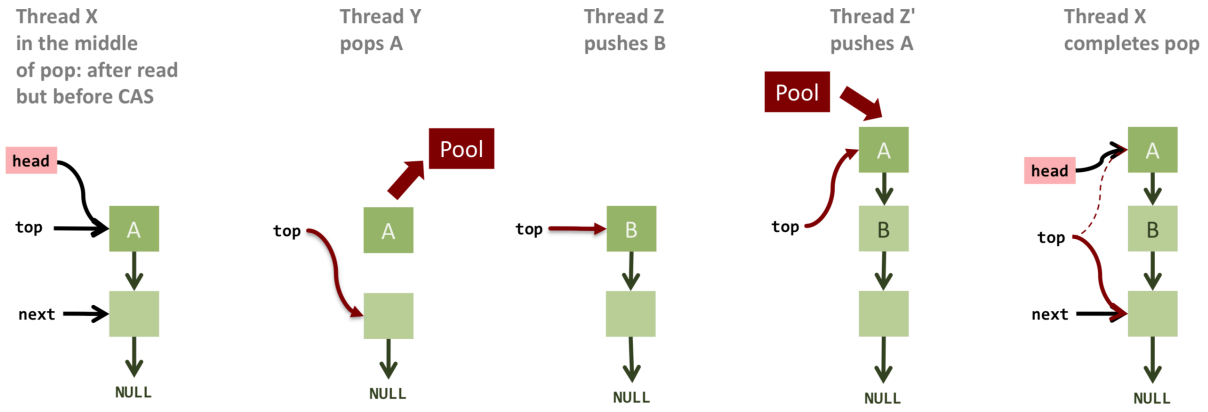
The mechanism to check for exclusive access in lock-free programming is CAS. A positive result of CAS suggests that no other thread has written (not always true because of the ABA problem).

15.1 Memory reuse / ABA problem

When we reuse objects e.g. by a ObjectPool (and don't rely on garbage collection), the ABA problem can occur. The definition is:

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.

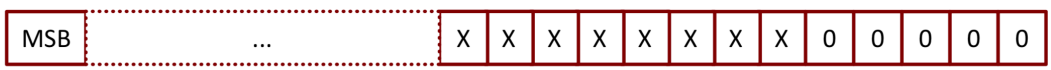
An example with a concurrent stack is outlined below. Thread X reads the value A, then comes Threads Y and pops A (and puts it in the NodePool), then comes Thread Z that pushes B and Thread Z' that pushes A again. Thread X wants to complete the pop now by `calling top.compareAndSet(head, next)` (where `head` is his local copy of A and `next` still the previous successor of A). This call succeeds because A is (again) the top element but the previously inserted Element B is lost because Thread X uses the previous successor of A in his CAS call.



15.1.1 Solutions

- **DCAS:** If we could check that the successor of A is still the same in the CAS in the previous example, the problem wouldn't occur. But DCAS is not available on most platforms.
- **Garbage collection:** Slow, especially for things like a runtime kernel
- **Pointer tagging:** Some bits of the addresses are made available for a tag. Each time a pointer is stored in a data structure, the tag is increased by one. If we have 5 bits for pointer tagging, the ABA problem is much less probable because 32 versions of each pointer exists:

Example: pointer aligned modulo 32 → 5 bits available for tagging



- **Hazard Pointers:** When a process reads a pointer, it marks it hazarduous by entering it in one of the n (=number of threads) slots of an array associated with the data structure. When finished (after the CAS), the process removes the pointer from the array. Before a process tries to reuse a pointer, it has to check if it is hazarduous (by checking all entries of the hazard array).

16 Transactional memory

The goal of transactional memory is to remove the burden of synchronization from the programmer and place it in the system (hardware / software). Furthermore, it solves the issue of locks that they aren't composable (combining thread-safe operations is hard).

In transactional memory, the programmer explicitly defines atomic code sections. The programmer only defines that the operations should be atomic but not how (→ declarative approach). The benefits of transactional memory is simpler and less error-prone code, higher-level semantics (what but not how), composability and they are optimistic by design. Changes made by transactions are made visible atomically i.e. other threads preserve either the initial or the final state, but not any intermediate states. Furthermore, transactions run in isolation. This means that while a transaction is running, effects from other transactions are not observed (as if the transaction takes a snapshot of the global state and operates on this snapshot). Therefore, the transactions appear serialized.

Transactional Memory is heavily inspired by database transactions with the ACID properties:

- Atomicity
- Consistency (database remains in a consistent state)
- Isolation (no mutual corruption of data)
- Durability (e.g. transaction effects will survive power loss, not important in transactional memory)

To implement transactional memory, the system keeps track of operations performed by each transaction and ensures atomicity / isolation properties. When a conflict occurs (e.g. read value was changed by another transaction during the run) the transaction gets aborted / rollbacked.

The consistency guarantee is usually implemented by a snapshot at the beginning or early abort (abort if a value has changed).

There are two different isolation levels:

- Strong isolation: When shared state is accessed outside of a transaction, the transactional guarantees are still maintained.
- Weak isolation: When shared state is accessed outside of a transaction, the transactional guarantees aren't maintained (i.e. we can have inconsistencies).

For nested transactions, there are also different design choices:

- Flat nesting: Nested transactions are flattened to one transaction i.e. an abort of an inner transaction causes the abort of the outer transaction and an inner commit is only visible if the outer transaction commits.
- Closed nesting: Similar to flat nesting, but an abort of an inner transaction does not result in an abort of an outer transaction. Furthermore, when the inner transaction commits the changes are visible to the outer transactions but they are only visible to other transaction when the outer transaction commits.

Some Transactional Memory implementations (e.g. Scala STM) provide a retry method which causes the transaction to abort and retry when any of the variables that were read change.

16.1 Implementation

In a clock-based STM system, we have a global clock that is read by transactions at their creation and all objects have a timestamp. When a transaction commits, the clock is increased. Each transaction uses a local read-set and a local write-set holding all locally read and written objects. When the transaction calls read, it checks if the object is in the write set in which case this version is returned. Otherwise it is checked if the object time stamp is bigger than the timestamp of the transaction, in which case the transaction aborts. If the time stamp is smaller, the value is retrieved from memory and added to the read set. When the transaction calls write, the object is copied and added to the write set.

When the transaction commits, all objects of the read- and write-set are locked and the system checks that all objects in the read set have a time stamp smaller than the timestamp of the transaction. If this is not true, the transaction aborts. Otherwise the clock is incremented and all elements of the write set are copied back to global memory (with the new timestamp).

17 Concurrency Theory

A method call is the interval that starts with an invocation and ends with a response. The method call is pending between invocation / response. In concurrent settings, method calls can overlap and an object might never be between method calls (when there is no call on the object, we call it periods of quiescence).

17.1 Linearizability

Each method should appear to take effect instantaneously between invocation and response events. An object for which this is true for all possible executions is called linearizable. The linearization points can often be specified but they may depend on the execution (e.g. if a queue is empty, a dequeue may fail while it does not fail with a non-empty queue, leading to different linearization points).

More formal, we have a History H that is a sequence of invocations and response. Invocations and response match, if thread names agree and object names agree. An invocation is pending if it has no matching response. A subhistory is complete when it has no pending responses.

Object projections are all invocations / responses on an object in a history whereas thread projections are all invocations / responses on a thread in a history:

Object projections

H|q =

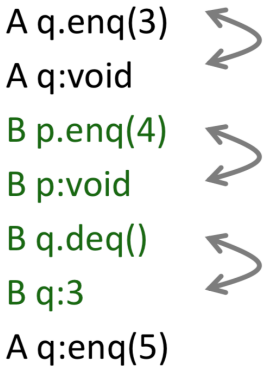
A q.enq(3)
A q:void
A q.enq(5)
B q.deq()
B q:3

Thread projections

H|B =

B p.enq(4)
B p:void
B q.deq()
B q:3

In **sequential** histories, method calls of different threads do not interleave. A final pending invocation is ok. An example of a sequential history is:



In **well-formed** histories, all thread projections are sequential. An example of a well-formed history is:

H= A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3

H|A = A q.enq(3)
A q:void

H|B = B p.enq(4)
B p:void
B q.deq()
B q:3

Histories are **equivalent**, if the thread projections are identical. An example of equivalent histories is:

H=		H|A =	
	A q.enq(3)		A q.enq(3)
	B p.enq(4)		A q:void
	B p:void	H|B =	B p.enq(4)
	B q.deq()		B p:void
	A q:void		B q.deq()
	B q:3		B q:3

A history is **legal** if for every object x, H | x adheres to the sequential specification of x.

A method call precedes another method call if the response event precedes the invocation event, otherwise they overlap. We denote $m_0 \rightarrow_H m_1$ if method execution m_0 precedes method execution m_1.

A history H is linearizable if it can be extended to a history G by

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take effect

such that G is equivalent to a legal sequential history S with

$$\rightarrow_G \subset \rightarrow_S$$

$\rightarrow_G \subset \rightarrow_S$ means that the "happens-before order" (method call a precedes method call b, …) of G is a subset of the "happens-before order" of S (in other words: S respects the real-time order of G).

The composability theorem states that history H is linearizable if and only if for every object x, H|x is linearizable. The consequence of this is modularity (linearizability of objects can be proven in isolation).

Atomic registers are linearizable with a single linearization point (they are sequentially consistent, every read operation yields most recently written value and for non-overlapping operations, the real-time order is respected).

When we use locking, the linearization points are when locks are released.

17.2 Sequential consistency

A history H is sequentially consistent if it can be extended to a history G by

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take effect

such that G is equivalent to a legal sequential history S.

No order across threads is required, sequential consistency is therefore weaker than linearizability. Only operations done by one thread need to respect program order, no need to preserve

real-time order (cannot re-order operations done by the same thread but can re-order non-overlapping operations done by different threads).

In most hardware / memory models, sequential consistency is violated by default for performance reasons but can be requested explicitly (e.g. with volatile).

17.3 Quiescent consistency

Programs should respect real-time order of algorithms separated by periods of quiescence.

18 Consensus

Consider an object with the method `decide(T value)`. A number of threads call `c.decide(v)`. A consensus protocol is a protocol with the following requirements:

- wait-free: consensus returns in finite time for each thread
- consistent: all threads decide the same value
- valid: the common decision value is some thread's input

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 C is the largest n such that C solves n-thread consensus.

Consensus is important because it can be used to proof that certain data structures cannot be implemented with certain operations / registers. E.g. atomic registers have consensus number 1. We can't implement a FIFO queue using atomic registers because we can implement 2-thread consensus with a FIFO queue by inserting one "red ball" and "black ball" and having an additional array where the threads write their proposed value. The one with the red ball wins and decides the value, the one with the black ball takes the value of the other thread. Therefore, a FIFO queue has consensus number 2 and can't be implemented by atomic registers.

The consensus hierarchy is as follows:

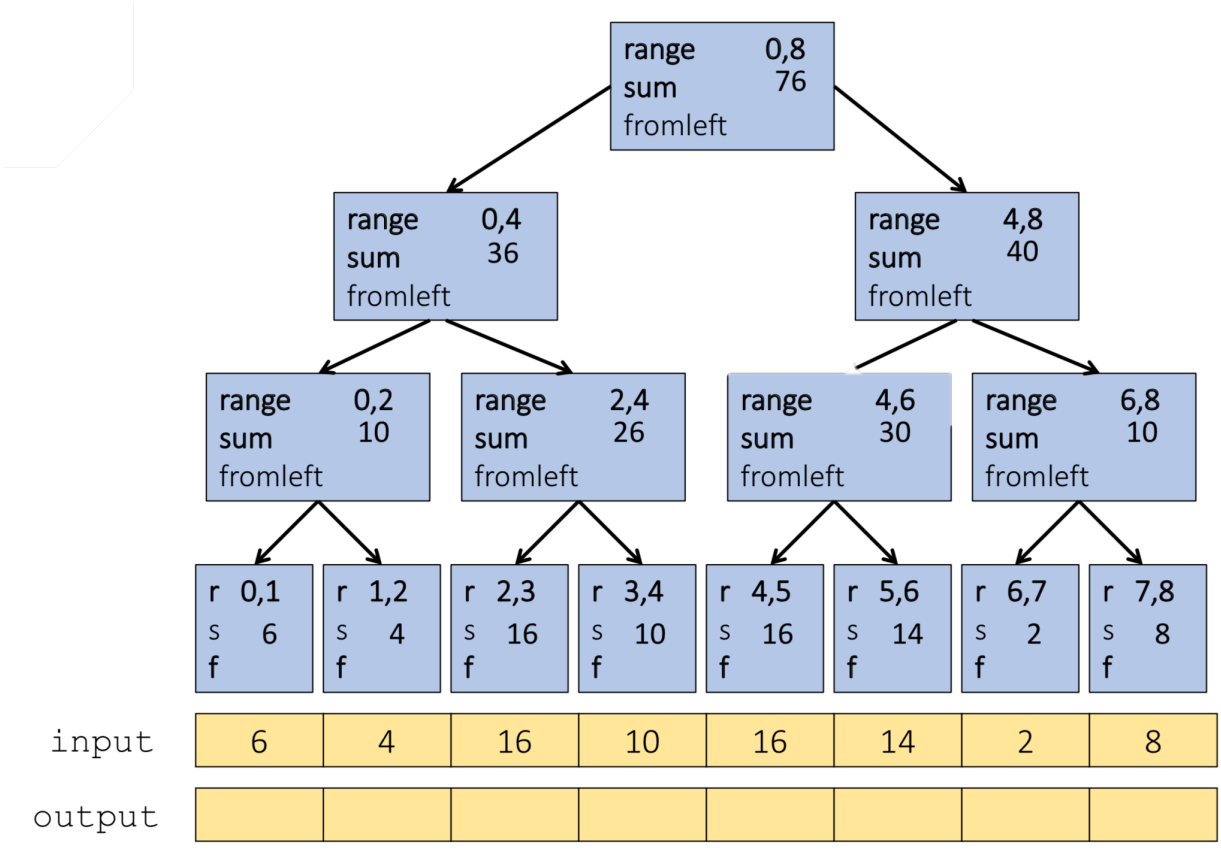
1	Read/Write Registers	
2	getAndSet, getAndIncrement, …	FIFO Queue LIFO Stack
. .		
∞	CompareAndSet, …	Multiple Assignment

Parallel algorithms / data structures

19 Parallel prefix-sum

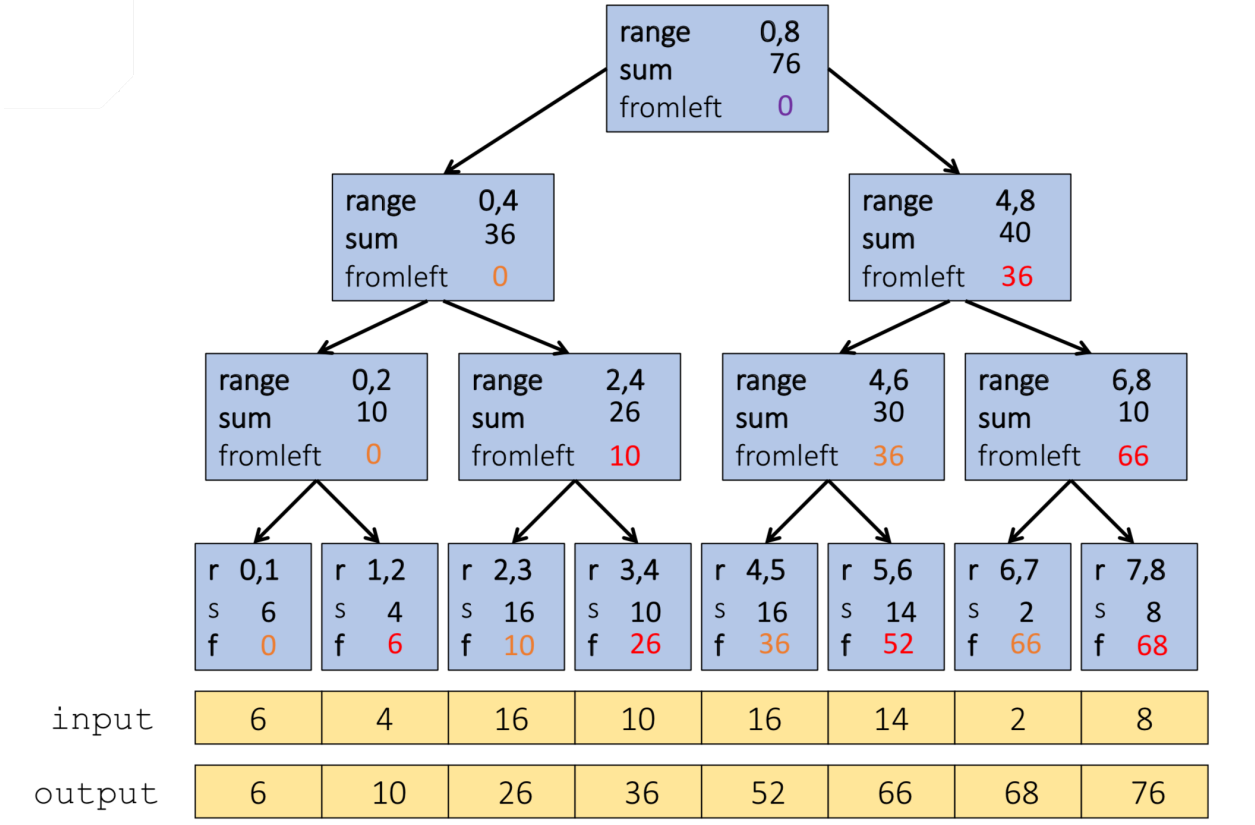
The prefix-sum problem is defined like this: Given an input array of n elements, produce an output array of n elements where the i-th element is the sum of elements 0 to i, i.e. `output[i] = input[0]+input[1]+...+input[i]`. The sequential algorithm doesn't seem parallelizable, but we can write another algorithm. The algorithm does two passes:

In the first pass (bottom-up), a binary tree is built where the sum under each node is noted:



In the second pass (top-down), each node takes its fromLeft value and passes its left child the same fromLeft value and its right child its fromLeft plus its left child's sum. This starts at the root node and the root node is given a fromLeft value of 0. The output at position i is then:

`output[i]=fromLeft+input[i]`



Both passes have $O(n)$ work and $O(\log n)$ span, therefore the total work is $O(n)$ and the total span is $O(\log n)$.

19.1 Parallel pack

Given an array input, produce an array output containing only elements such that $f(elt)$ is true. With parallel prefix, this can be parallelized:

1. Parallel map to compute a bit-vector for true elements:

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector:

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output (because of the parallel-prefix sum on the bit-vector, we have the positions of true elements in the `bitsum` array):

```
output [17, 11, 13, 19, 24]
```

19.2 Quicksort

If we just do the two recursive calls in parallel, our span is $O(n)$. But we can use our parallel pack method to do the partitioning by packing elements less than pivot into the left side of an

array and the elements greater than pivot into the right side of an array. Because parallel pack has a span of $O(\log n)$, we get a total span of $T(n) = O(\log n) + 1T\left(\frac{n}{2}\right) = O(\log^2 n)$

20 Producer / Consumer Pattern

In the producer / consumer pattern, there is a producer (thread) and a consumer (thread) (or multiple threads). We need a synchronized mechanism to pass objects between the producer and the consumer. The producer / consumer pattern can be used to build data-flow parallel programs, e.g. pipelines.

To implement the producer / consumer, a circular buffer can be used as a queue. Such a buffer can be implemented with a semaphore, but is much easier to implement using a monitor:

```
synchronized void enqueue(long x) {
      while (isFull())
            try {
                wait();
           } catch (InterruptedException e) { }
          doEnqueue(x);
          notifyAll();
}

synchronized long dequeue() {
      long x;
      while (isEmpty())
            try {
            wait();
            } catch (InterruptedException e) { }
    x = doDequeue();
    notifyAll();
    return x;
}
```

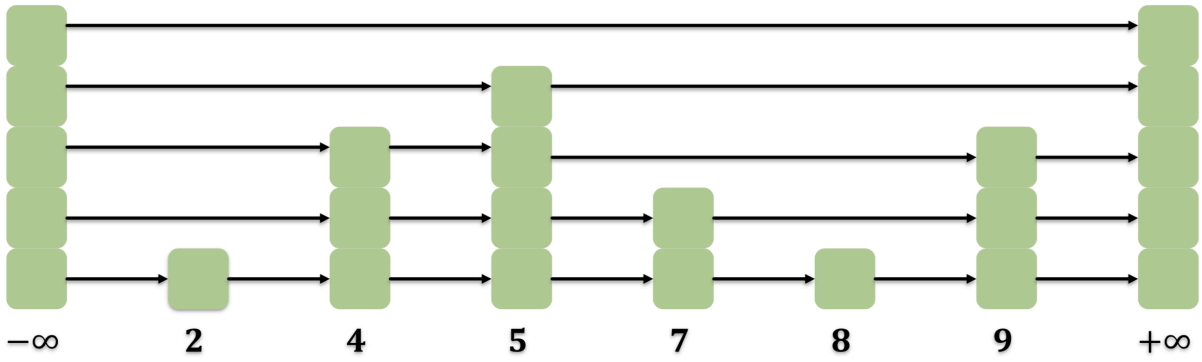
As usual, we have to check the condition in a `while`-loop because we can't be sure it is fulfilled after we got signaled.

We can also use Javas Lock interface with its condition to implement the buffer. To avoid that the signal is sent when no threads are waiting, we can work with two additional variables that indicate how many threads are waiting (for enqueuing / dequeuing).

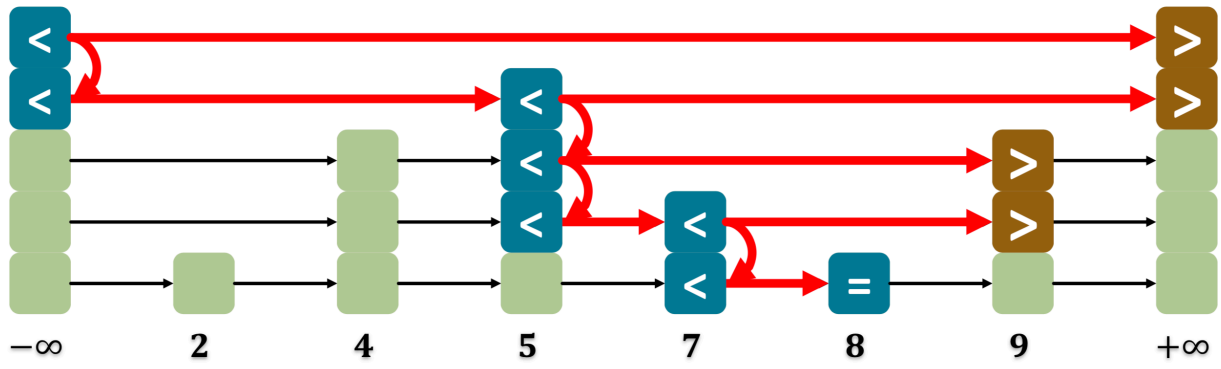
21 Lazy Skiplist

A skiplist is a collection of elements (without duplicates) with the methods `add`, `remove` and `find`. We assume that there are many calls to `find`, fewer to `add` and much fewer to `remove`.

The skiplist is a sorted multi-level list where the height of a node is probabilistic (e.g. $\Pr[height = n] = 0.5^n$) and we have two sentinels that are in the maximal level. If a node is in a level i, he is automatically in all levels $0 \dots i-1$ as well. All nodes are in level 0:

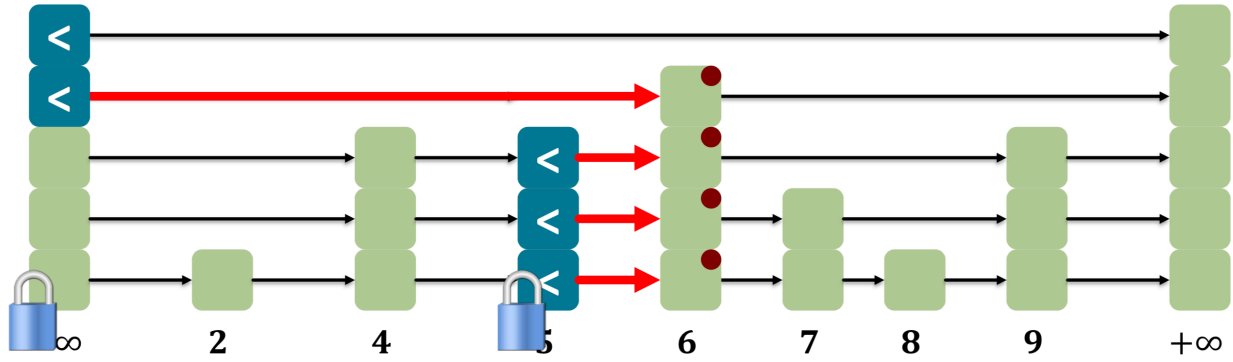


When searching for a value, we start at the sentinel $-\infty$ and check if the predecessor of the highest level is smaller than our value. If this is true, we skip all nodes between the sentinel and the predecessor and continue at the predecessor (because the predecessor is smaller and therefore all nodes between are too). If this is not true, we go a level down and check again if the predecessor is smaller. This gets repeated until we find the node (or we are in the lowest level and haven't found the node, i.e. he isn't in the list). The search for 8 is illustrated below:



The expected runtime for contains is logarithmic (with high probability) and it is wait-free.

When we add a new node, we need to find the correct destination, lock all predecessors, validate and finally insert the node. E.g. when we add 6 with height 4:



When we remove a node, we find the predecessors, lock the victim, logically remove the victim (mark it), lock the predecessors and finally physically remove the node (change pointers).

22 Lock-free stack

A stack is quite easy to implement lock-free, when popping / pushing we just need to make sure that the top element is still the element we retrieved:

```
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 newi = new Node(item);
     Node head;

     do {
          head = top.get();
          newi.next = head;
     } while (!top.compareAndSet(head, newi));
}
```

23 Lock-free list set

Implementing a list set (set based on a linked list) is more difficult because we want to atomatically establish consistency of two things: The next pointer and the mark bit. Java provides some sort of DCAS with AtomicMarkableReferences. One bit of the reference is reserved as a mark bit that can be checked in the compareAndSet as well. With these references, when removing a node, we first try to set the mark of the next pointer (logically deleting it). Then we do the "DCAS" on the predecessor which only succeeds when pred.next wasn't marked / updated:

```
public boolean remove(T item) {
   Boolean snip;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key != key) {
         return false;
      } else {
         Node succ = curr.next.getReference();
         snip = curr.next.attemptMark(succ, true);
         if (!snip) continue;
         pred.next.compareAndSet(curr, succ, false, false);
         return true;
      }
   }
}
```

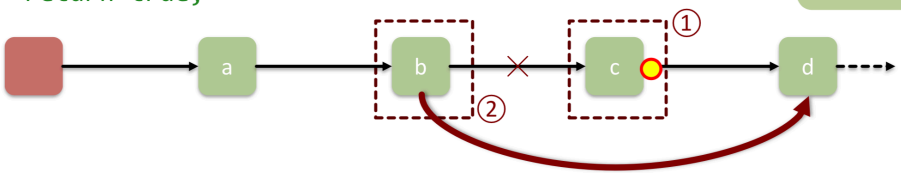
Find element and prev element from key

If no such element -> return false

Otherwise try to logically delete (set mark bit). ①

If no success, restart from the very beginning

Try to physically delete the element, ignore result ②



The physical deletion can fail, therefore other threads “clean up” when they observe a logically but not physically deleted node while traversing (if other threads would have had to wait for one thread to clean up the inconsistency, the approach would not have been lock-free).

When we add a new node, we keep retrying until we succeed:

```
public boolean add(T item) {
   boolean splice;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = new AtomicMarkableRef(curr, false);
         if (pred.next.compareAndSet(curr, node, false, false))
            return true;
      }
   }
}
```

Find element and prev element from key

If element already exists, return false

Otherwise create new node, set next / mark bit of the element to be inserted

and try to insert. If insertion fails (next set by other thread or mark bit set), retry

24 Lock-free unbounded queue

An unbounded queue is a queue with “unlimited” (not restricted) capacity. When we want to implement a queue lock-free, we run into problems because we sometimes need to simultaneously update head / tail / tail.next. We introduce a sentinel node at the start and the head pointer always points at this node. Now we only need to update tail / tail.next. The enqueue method appends the node and tries to set the tail pointer to the appended node. This can fail (other thread observed in the meantime that tail did not point to the last element and changed it)

but this doesn't matter because in these cases it was changed by another thread and the overall state is still consistent.

```
public void enqueue(T item) {
   Node node = new Node(item);
   while(true) {
      Node last = tail.get();
      Node next = last.next.get();
      if (next == null) {
         if (last.next.compareAndSet(null, node)) {
            tail.compareAndSet(last, node);
            return;
         }
      }
      else
         tail.compareAndSet(last, next);
   }
}
```

Create the new node

Read current tail as last and last.next as next

Try to set last.next from null to node, if success then try to set tail

Ensure progress by advancing tail pointer if required and retry

Help other threads to make progress !

When we dequeue an element, the current next element becomes the new sentinel (i.e. is "deleted). Before advancing head, we must make sure that tail is not left referring to the sentinel node (that gets deleted). Therefore we check if `first == last` and the the sentinel has a successor (`next != null`). In this case, we need to update tail to the successor of the sentinel:

```
public T dequeue() {
   while (true) {
      Node first = head.get();
      Node last = tail.get();
      Node next = first.next.get();
      if (first == last) {
         if (next == null) return null;
         else tail.compareAndSet(last, next);
      }
      else {
         T value = next.item;
         if (head.compareAndSet(first, next))
            return value;
      }
   }
}
```

Read head as first, tail as last and first.next as next

Check if queue looks empty
(1) really empty: return
(2) next available: advance last pointer

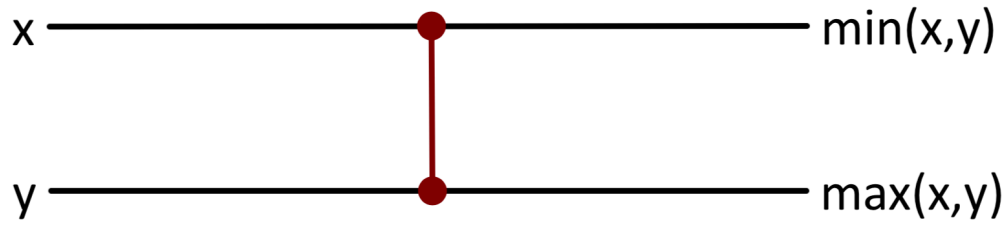
If queue is not empty, memorize value on next element and try to remove current sentinel

Retry if removal was unsuccessful

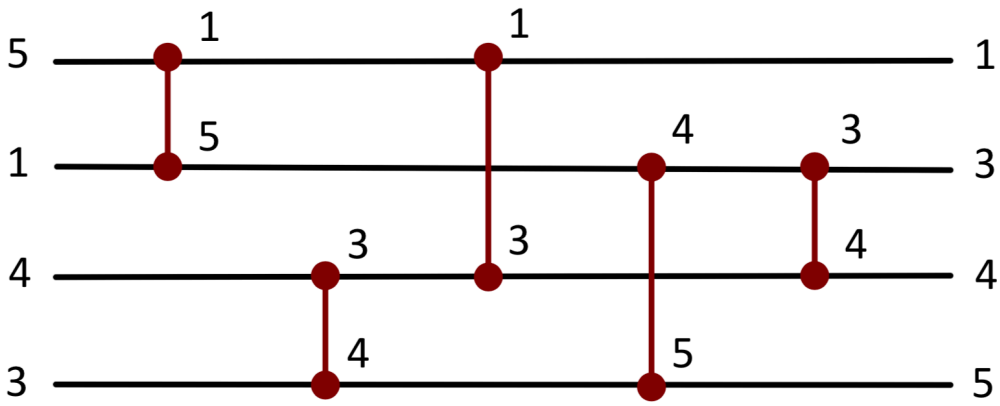
Help other threads to make progress !

25 Sorting networks

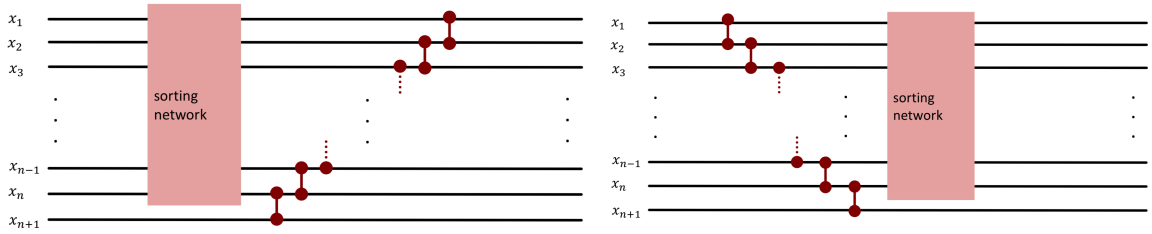
Sorting networks are abstract devices that are very efficient for parallel execution (and implementation in hardware). They are built from comparators that take two values and return the min / max of the two:



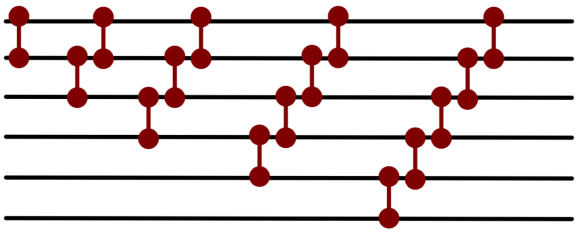
A sorting network consists of many comparators so that each possible input sequence gets sorted:



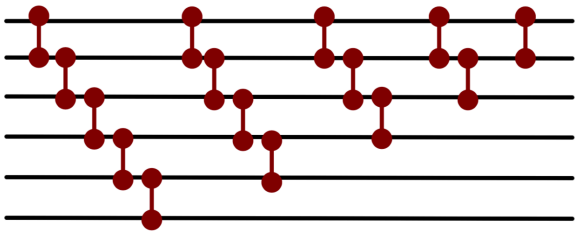
A sorting network can be constructed recursively quite easy. Given a network that sorts the numbers $x_1, \dots, x_n$ we just can add n comparators in front or behind the network to construct one that sorts the numbers $x_1, \dots, x_{n+1}$.



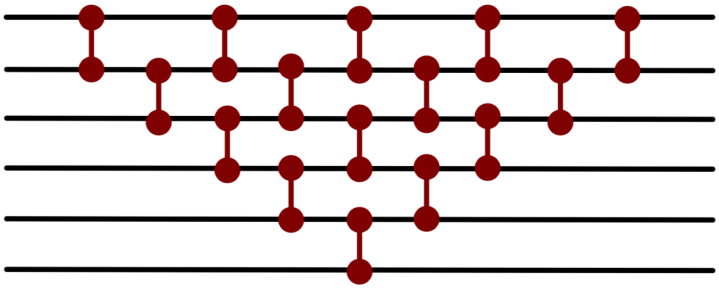
Applied recursively, this gives us a insertion sorting network and a bubble sorting network, which are the same with parallelism:



insertion sort



bubble sort



with parallelism: insertion sort = bubble sort !

Using these sorting networks, a computer with infinite numbers of processors requires $2n-3$ steps to sort n numbers (longest sequence of comparators). The networks are well suited for parallel processing because the comparators on the same vertical line can be executed in parallel. These sorting networks are not optimal, but finding the optimal sorting network is a hard task (testing whether a candidate network is a sorting network is co-NP complete i.e. the class of problems for which there is a polynomial-time algorithm that can verify counterexamples).

The zero-one-principle states that if a network with n input lines sorts all 2^n sequences of 0s and 1s into nondecreasing order, it will sort any arbitrary sequence of n numbers in nondecreasing order. The proof is quite simple, if x is sorted by a network N, then also any monotonic function of x. If x is not sorted, then there is a monotonic function $f(x)$ that maps x to 0s and 1s and is not sorted as well. Such a function is $f(x) = 0 \text{ if } x < y, 1 \text{ otherwise}$ (for some non-minimal y). Such a function fulfills $f(x) \leq f(y)$ whenever $x \leq y$.

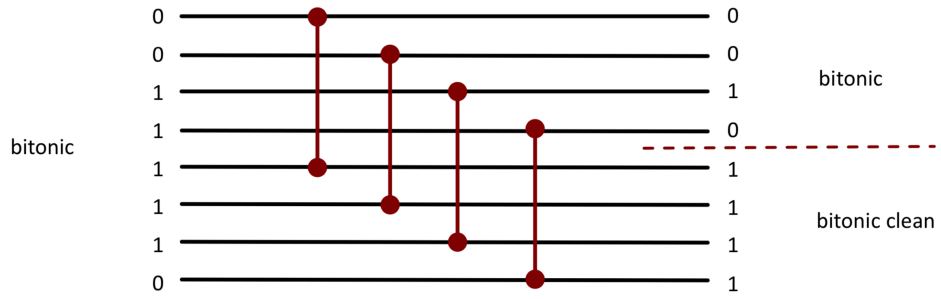
25.1 Bitonic sort

Bitonic sort is a parallel algorithm for sorting with a time complexity of $O(n\log^2 n)$ when he is executed sequentially and $O(\log^2 n)$ for the parallel case (worst case = average case = best case).

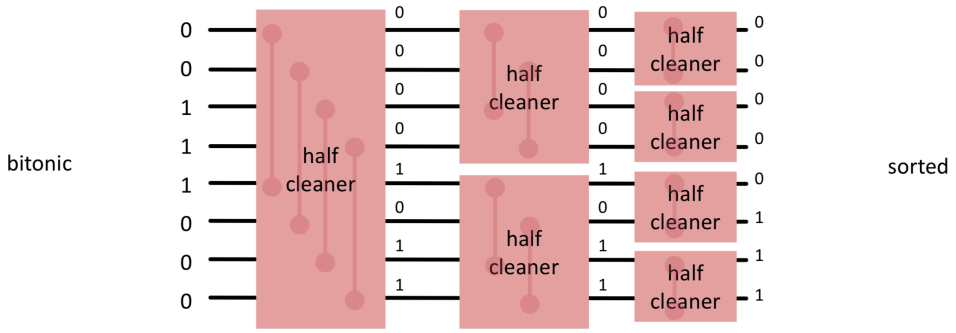
A bitonic set is a set in which the sign of the gradient changes once at most, a bitonic sequence is defined as a list with no more than one local maximum / minimum (→ can be wrapped around to a bitonic set):

(a) Single maximum (b) Single maximum and single minimum

A half-cleaner is a sorting network that splits a binary bitonic sequence into a bitonic and bitonic clean (all 0s or 1s). Every number in the upper half is greater or equal than in the lower half:



A bitonic sorting network takes a bitonic sequence as an input and sorts them with multiple half cleaners:



To get the bitonic sequences, “mergers” consisting of half cleaners and bi-mergers (acts like a half-cleaner, but on two sorted sequences). So the final sorting network to sort any input looks like this: