PProg Exam Prep.

Tom Offermann

August 2023

1 Introduction

This is a collection of different definitions and useful corollary's from the **"Parallele Programmierung"** lecture held at **ETH Zurich** in **Spring 2023**.

* Note that the exam does not allow a Cheat-Sheet. This is just meant to be a collection of (hopefully) all the necessary information.
There is **no guarantee** that the information presented is correct or complete. Do not hesitate sending an email if you think something is wrong or missing.

2 Parallel Programming

T_1: Time on one processor
T_P: Time on P processors
T_∞: Time on "infinite" processors ($\lim_{P\to\infty} T_P$)

$S_P = \frac{T_1}{T_P}$: Speedup generated using P processors

• **Amdahls Law** assumes a fixed workload in variable time it states:

$S_P = \frac{T_1}{T_P} \leq \frac{1}{f+\frac{1-f}{P}}$

• **Gustafsons Law** assumes a fixed time window, but measures the speedup in workload terms:

$S_P \leq f + P(1-f)$

Pipelining:

• $Throughput \approx \frac{1}{max(\frac{pipeline\ stage\ time}{\#execution\ units})}$
• *Latency*: The time to perform all pipeline stages at a given execution time. Note that the latency can increase indefinitely (if the longest pipeline stage is not the first stage). If the latency remains constant, we say that the pipeline is balanced.

Task Graphs:

• T_1: $\sum val(node)$
• T_∞: $\sum_{node \in critical\ path} val(node)$
$T_1 - T_\infty$ gives us the sum of nodes that are not on the critical path.

3 Java Wait/ Notify

$wait()$ and $notify()$/ $notifyAll()$ are methods that can be called on a lock/ locked object:

• $wait()$: Can only be called once the object obtained the lock for the object, and will set the thread that called $wait()$ into a waiting state, and releases the lock (it can be woken up by $notify()$/ $notifyAll()$)

• $notify()$/ $notifyAll()$: These methods will wake up other threads, that are currently in a waiting state. Both methods can also only be called if the lock of the object is currently held. $notify()$ will choose a "random" thread T to wake up, continues its execution until the lock is released and moves T into the lock "acquire" state. $notifyAll()$ works the same, but it will wake up every thread currently in the waiting queue, after the thread that called $notifyAll()$ releases the lock.

Two "rules" for using $wait()$/ $notify()$:

```
// 1. Always enclose wait() in a while-loop:
synchronized(lock) {
    while(!condition) {
        lock.wait();
    }
    ...
}
// 2. Only call wait() and notify() on locked obj.:
synchronized(lock) {
    notifyAll();
    ...
}
// Conditions:
Condition C = lock.newCondition();
C.await();
C.signal();
C.signalAll();
```

4 Concurrent Programming

Different programming structures:

• **Lock:**
A lock has two methods: $acquire()$ and $release()$. When one thread acquires the lock, any other thread will fail to acquire the lock, until the first thread releases it, the second thread waits in that time.
Some lock implementations are at the end of this document.
• **Semaphore:**
A Semaphore S is essentially a parallel counting variable, usually initialized with some Integer value N, S = N or S(N). It also implements the two methods $acquire()$ and $release()$. If some thread acquires S, it will decrement

the counter by one S = S - 1, if S > 0. If S == 0 the thread will wait until S > 0 again and then continue the acquire. releasing S will increment the counter S = S + 1. Both of these methods are atomic (appear instantaneously). A simple Semaphore:

```
int count = N; // Init
synchronized void aquire() {
    while(count == 0) wait();
    count--;
}
synchronized void release() {
    count++;
    notify();
}
```

• **Barrier:**
A Barrier is used to synchronize progress among N threads/ processes. There are two variants of a barrier implementation at the end of this document.

When implementing a concurrent data structure there is a trade off between simplicity and speed. Analogously there are four types of lock-granularity that go from simplest & slowest to most complicated & fast:

• **Coarse Grained Locking:** Locks the entire data structure, makes the operation sequentially and release the lock after finishing.
• **Fine Grained Locking:** Locks only the parts of the data structure that are needed to perform the operation correctly. For a linked list this could be two locks that are locked in a hand-over-hand fashion.
• **Optimistic Locking:** Tries to do the operation sequentially. Once the operation should take place it checks if the state observed without locking is still correct (validation), then it locks the essential data and performs the operation. Implemented correctly this gives a big performance boost compared to the two methods above. The key to such an implementation is to come up with good in-variants.
• **Lazy Locking:** In case of a concurrent linked list lazy locking uses a marked bit, that logically removes a node from the list. This massively decreases the amount of locks on our data structure and thus also improves performance significantly. Lazy Locking is an optimization on top of optimistic locking.

Non-Blocking Algorithms:

These algorithms heavily rely on atomic operations that are most of the time implemented by the underlying hardware (or their behaviour is guaranteed by java)

```
// Test And Set
bool TAS(bool* value) {
    if(*value == false) {
        *value = true;
        return true;
    }
    return false;
}

// Compare And Swap
bool CAS(boolean* value, int expected, int new) {
    oldValue = *value;
    if(oldValue == expected) {
        *value = new;
    }
    return oldValue;
}
```

ABA-Problem:
The ABA-Problem occurs, if we are reusing data elements, for example if we are using a node pool for our concurrent Linked-List. The ABA problem then occurs if we are checking two (or more) conditions, that should be met, (which should actually be checked atomically to ensure correct execution). If these to conditions cannot be rechecked atomically a second thread could potentially manipulate our data structure in a way that leads to undesired behavior.

We can try to fix the ABA problem by using:
• **Transactional Memory**
• **DCAS**
• **Pointer-Tagging (makes it unlikely)**
• **Hazard Pointers**
• **Garbage Collection**

5 Important Definitions

• **Thread-local:** The memory is copied into each thread and they do their work independently (share their results in the end (barrier, thread.join(), ...))
• **Immutable:** The data that is operated on is shared, but not changed (final keyword in java), which is always a safe option, but not always applicable.
• **Synchronized:** The shared memory access is controlled by some synchronization technique like locking.
• **Atomic operations:** Like with synchronization we control the memory access, but usually in a non-blocking manner (TAS, CAS, DCAS, LL/SC...) to increase performance.

• **Deadlock:** Is a program-state whithout outgoing edges, meaning it will not progress.
• **Livelock:** Is a cycle of program-states, where no thread is in the critical section.
• **Mutually exclusive:** There is no program state, in which more than one thread has entered the critical section

• **Deadlock-free:** At least one thread is guaranteed to proceed into the CS at some point in time.

• **Starvation-free:** All threads are guaranteed to proceed into the CS at some point in time.
• **Lock-free:** At least one thread always makes progress in a finite number of steps.
• **Wait-free:** All threads make progress in a finite number of time.

When there are no locks in your program it is **NOT** lock-free nor wait-free (spin-locks, ...)

• Note, that deadlock-freedom and starvation-freedom are essentially the same definitions as lock-freedom and wait-freedom respectively, but deadlock-freedom and starvation-freedom are assuming some program/progress conditions, that a Thread cannot just die while holding a lock for example. Which makes it a much stronger property

• Wait-free $\implies$ Lock-free
• Wait-free $\implies$ Starvation-free
• Lock-free $\implies$ Deadlock-free
• Starvation-free $\implies$ Deadlock-free
• Deadlock-free AND Fair $\implies$ Starvation-free

A CS has to be Deadlock-free and mutually exclusive!

• **Notion of Fairness (especially for locks):**
For a given lock-implementation we are defining the doorway and waiting section. Where the doorway always takes a finite amount of operations, but the waiting section may be unbounded.
We call a lock fair according to the first-come-first-serve principal, if whenever Thread A finishes its doorway section before Thread B, then Thread A will also finish the waiting section before Thread B.

Lock	**Deadlock free**	**Starvation free**	**Fair (first-come-first-serve)**
Peterson-Lock	✓	✓	✓
Filter-Lock	✓	✓	×
Bakery-Lock	✓	✓	✓
TAS-Lock (Spinlock)	✓	×	×

6 Sequential Consistency & Linearizability

• Histories A and B are called **equivalent**, if A and B's per-thread projections are identical
• A History H is called **complete**, if every invocation has a matching response (not necessarily immediately after the invocation).
• A History H is called **well-formed**, if its per-thread projections are all sequential. Histories that are not well formed usually do not make sense.
• A history H is called **legal**, if for every object x, the projections $H|x$ all behave like the sequential specification of the object x.
• A History H is **sequential**, if there are no overlapping methods (when every invocation is immediately followed by the matching response)

• A History is SC (**Sequentially Consistent**), if:
1. Every Thread projection is a sequential history.
2. Method calls appear to follow PO (Program Order), which allows for "reordering" of method-calls as long as they follow the ordering determined by the corresponding Thread-projection.

• For a program to be called SC (Sequentially Consistent), every possible execution history has to be SC.

• A History H is
linearizable, if there is an extension H' to H, which is equivalent (thread-projection-wise) to a legal sequential History S, where for all methods $m_x \rightarrow_H m_y \implies m_x \rightarrow_S m_y$. What linearizability means, is that the given parallel/ concurrent execution is equal to some sequential history, where a preceding method call shows effect before the later one, but overlapping method calls can be "reordered" as wished. The reordering means, that if m_1 and m_2 overlap (independent of which methods invocation or response was first/second) we can choose, if m_1 or m_2 shows its effect first.

7 MPI & Fork-Join Patterns

• **MPI (Message Passing Interface):**
MPI is an interface for parallel computing. It defines the syntax and semantics of many different library methods (like send, receive, reduce, map, ...) that are useful to write a program that runs on millions of processors.

In MPI we have multiple processes, that share data/ state via messages. Each process is associated with an integer value called the rank (unique ID), which ranges from $[0, \#processes - 1]$. Each message also has an ID, called the destination or source (from sender or receiver perspective). Processes communicate via a communicator, the default is COMM_WORLD, where all processes

communicate via the same "channel". When using MPI, we do not touch the implementation of the different methods, but rather use the interface in a language we like (C/C++, Java, Python, ...)

The two core methods of MPI are *Send*() and *Receive*(), which are defined as follows:

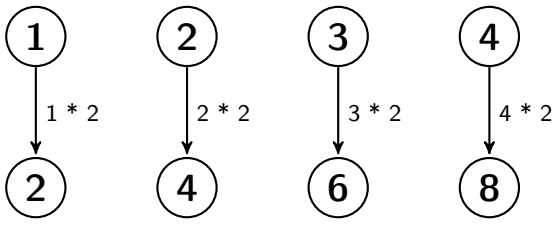
```
// Sends "<count>-many" elements of
// the <data> reference (of type: <datatype>)
// to the process with rank (<destination>)
// the <tag> identifies the message (id) and
// the <communicator> is a handle to distinguish
// between different communicators/ groups
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator
)

// Receives "<count>-many" elements of
// the <data> reference (of type: <datatype>)
// from the process with rank (<source>)
// the <tag> identifies the message (id) and
// the <communicator> is a handle to distinguish
// between different communicators/ groups
// <status> keeps track of the messages status :)
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status
)
```

• Parallel Algorithm Patterns:

• Map:

A map maps the elements of one set through a function onto a new set containing the modified results.

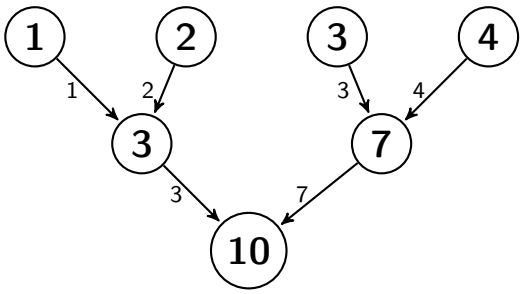
MAP([1,2,3,4], $x \mapsto 2 \cdot x$) → [2,4,6,8]:



• Reduce:

A reduce will perform an associative operation onto all elements in a set and return a single resulting value.

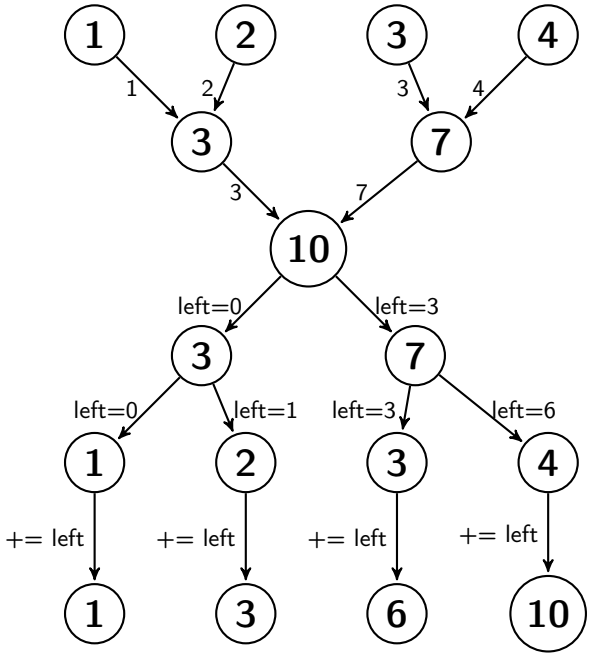
REDUCE([1,2,3,4], +) → [10]:



• Prefix Sum:

The prefix sum of a list of integers is calculated like this:

PREFIX_SUM([1,2,3,4]) → [1,1+2,1+2+3,1+2+3+4] = [1,3,6,10]:



• Pack/ Filter:

The pack or filter routine takes a set of elements and a predicate $p(x) \mapsto 0/1$. It filters the set for all elements x where $p(x) = 1$. The parallel algorithm works the following:

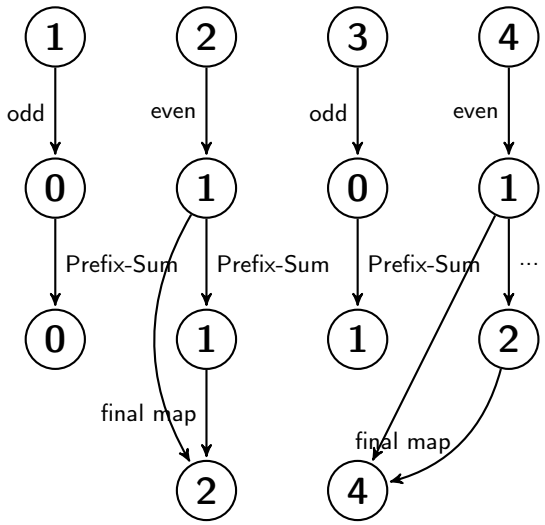
1. Perform a parallel map to compute the bit-vector for all true elements.

2. Compute the prefix-sum on the bit-vector

3. Perform another map, this time on the bit-vector, which picks an element x at index i, where $p(x) = 1$ and calculates its result index i_{new} by $prefix\_sum[i] - 1$. It maps this value into a result array ($result[i_{new}] = x$) with length $prefix\_sum[n-1]$, where n is the input array length.

PACK([1,2,3,4], $p(x) = 1 \iff x \bmod 2 = 0$) →

[2,4]:



7.1 MPI Methods

• **Gather:** Gather the data from different processes and collect them in one root process
• **AllGather:** Gather the data from different processes and collect them in all processes
• **Scatter:** Split the array of data in chunks and send them from one process to all other processes
• **Broadcast:** Send the whole array of data from one process to all other processes
• **Reduce:** Reduce data into one value (associative operator) and save the result in a root process
• **AllReduce:** Reduce data into one value (associative operator) and save the result in all processes

8 Merksachen für die Prüfung

• **Executorservice**

```
// Ohne Rückgabewert:
class Work1 extends Runnable {
    public Work1 (...) { ... }
    @Override
    public void run() {
        ...
    }
}
// Mit Rückgabewert:
public class Work2 extends Callable<T> {
    public Work2 (...) { ... }
    @Override
    public T call() {
        ...
        return <T>();
    }
}
```

• **ForkJoin**

```
// Ohne Rückgabewert:
public class Task1 extends RecursiveAction {
    public Task1 (...) { ... }
     @Override
    protected void compute() {
        ...
        Task1 l = new Task1(...);
        Task1 r = new Task1(...);
        l.fork();
        r.compute();
        l.join();
    }
}
// Mit Rückgabewert:
public class Task2 extends RecursiveTask<T> {
    public Task2 (...) { ... }
     @Override
    protected void compute() {
        ...
        Task2 l = new Task2(...);
        Task2 r = new Task2(...);
        l.fork();
        T a = r.compute();
        T b = l.join();
        return <T>combine(a,b);
    }
}
```

• **Barrier Implementations**

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

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

    public void await() {
        synchronized(this) {
            count++;
            if(count == threshold) {
                barrier2.aquire();
                barrier1.release();
            }
        }
        barrier1.aquire();
        barrier1.release();
        synchronized(this) {
            count--;
            if(count == 0) {
                barrier1.aquire();
```

```
                barrier2.release();
            }
        }
        barrier2.aquire();
        barrier2.release();
    }
}

class BarrierWaitNotify {
    private int threads;
    private int waiting = 0;
    private boolean doorOpen = true;

    public BarrierWaitNotify(int threads) {
        this.threads = threads;
    }

    public synchronized void await() {
        while(!doorOpen) wait();

        waiting++;
        while(doorOpen && waiting < threads)
            wait();

        if(doorOpen && waiting == threads) {
            doorOpen = false;
            notifyAll();
        }

        waiting--;

        if(waiting == 0){
            door_is_open = true;
            notifyAll();
        }
    }

}
```

• **Lock**

A lock has to be **deadlock-free** (in itself, not in the sense that it can lead to deadlocks), **starvation-free** and **mutually exclusive**. This definition implies however, that all spin-locks from the lecture are not correct lock-implementations.

*** Note** that in the following lock implementations whenever we write $volatileT[]$ of size n, we actually mean n independent volatile values. This could also be achieved with AtomicBooleanArray, ...

• **Peterson-Lock**

```
volatile boolean[] flag = new boolean[2];
volatile int victim;
void lock(int id) {
    flag[id] = true;
    victim = id;
    while(flag[1-id] && victim == id) {/*wait*/}
}
void unlock(int id) {
    flag[id] = false;
}
```

• **Deckers-Algorithm**

```
volatile boolean want0;
volatile boolean want1;
volatile int turn;
void lock(int id) {
    want<id> = true;
    while(want<1-id>) {
        if(turn == 1-id){
            want<id> = false;
            while(turn == 1-id){};
            want<id> = true;
        }
    }
}
void unlock(int id) {
    turn = 1-id;
    want<id> = false;
}
```

• **Filter-Lock**

```
volatile int[] level;
volatile int[] victim;
volatile int n;

// k: k != me: level[k] >= i
boolean Others(int me, int lev) {
    for (int k = 0; k < n; ++k)
        if (k != me && level.get(k) >= lev)
            return true;
    return false;
}

public void lock(int me) {
    for (int lev = 1; lev < n; ++lev) {
        level[me] = lev;
        victim[lev] = me;
        while(me == victim[lev]
            && Others(me,lev));
    }
}

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

• **Bakery-Lock**

```
volatile boolean[] flag;
volatile int[] label;
volatile int n;

// k: k != me: level[k] >= i
```

```
boolean Conflict(me) {
    for(int i = 0; i < n; i++) {
        if(i != me && flag[i]) {
            int diff = label[i] - label[me];
            if(diff < 0 || diff == 0 && i < me)
                return true;
        }
    }
    return false;
}

public void lock(int me) {
    flag[me] = true;
    label[me] = max(labels) + 1;
    while(Conflict(me));
}

public void unlock(int me) {
    flag[me] = false;
}
```

• **TAS-Lock**

```
volatile inCS = false;
public void lock() {
    while(inCS.getAndSet(false)){};
}

public void unlock() {
    inCS = false;
}
```

• **TATAS-Lock**

```
volatile inCS = false;
public void lock() {
    do{
        while(inCS.get()){}
    }
    while(inCS.getAndSet(false));
}

public void unlock() {
    inCS = false;
}
```

• **CAS-Lock**

```
volatile inCS = false;
public void lock() {
    while(!inCS.compareAndSet(false,true));
}

public void unlock() {
    inCS = false;
}
```

• **Consensus**

An implementation of a consensus protocol implements a method decide(v), which returns a value t. The value t has to be the same across all calls to decide (for every thread). And has to be some threads input value to decide (v). The decide method also has to be implemented wait-free!

If we are solving binary consensus, there are 3 different types of states:

• univalent: State, were the output is settled on either 0 or 1
• bivalent: both outputs 0 and 1 are still possible
• critical: bivalent & the following state-transition ends in two univalent states