Parallele Programmierung - Essentials

Simon Sure
https://simonsure.com

June 25, 2023

Bei Anmerkungen/Verbesserungsvorschlägen etc. gerne jederzeit bei mir melden. Ich gebe KEINE Garantie bzgl. Korrektheit oder Vollständigkeit für diese Inhalte. Der Inhalt dieser Datei ist stark durch den Kurs beeinflusst und enthält z. B. Bilder von diesem. Das Skript ist jedoch nicht mit dem Kurs affiliiert.

Contents

Parallele Programmierung thematisiert Computing auf mehreren Ausführungseinheiten (CPUs, CPU-Kerne, Computer, ...). Wir betrachten hier meist mehrere CPU-Kerne, auf welchen wir zeitgleich Berechnung ausführen. So möchten wir Performance-Steigerungen erreichen, ohne von Hardware-Verbesserungen der einzelnen Kerne 'abhängig' zu sein.

Es ist wichtig zu wissen, dass viele Begriffe in der Parallelen Programmierung nicht zu 100 % standardisiert sind. Begriffe wie Data races, Interleavings, Deadlocks, Prozess, Thread, ... werden je nach Kontext leicht unterschiedlich verstanden.

Eine Schwierigkeit bei der Parallelen Programmierung ist, dass wir uns bereits an sequentielles Programmieren gewöhnt haben und auch meist sequentiell über Programme nachdenken. Dies führt zu "unabsichtlicher Blindheit" (inattentional blindness) bzgl. Problemen von parallelen Programmen.

1 Java Recap

Java ist eine Cross-Plattform Programmiersprache. Java Compiler übersetzen den Quellcode in Byte-Code, welcher für alle Plattformen identisch ist. Eine kompilierte Datei kann mit 'javap -c filename' betrachtet werden. Der Byte-Code wird auf einem Computer dann von einer plattformspezifischen JVM (Java Virtual Machine) ausgeführt.

1.1 JVM - Java Virtual Machine

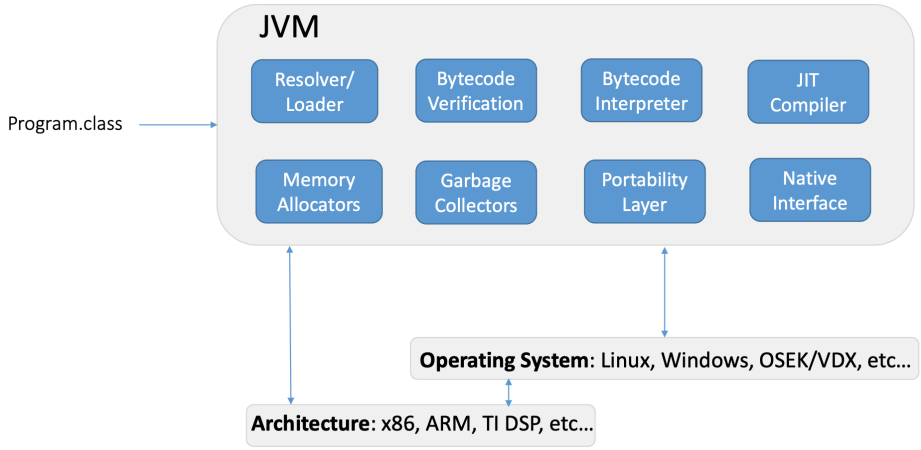


Figure 1: wichtige Bestandteile der JVM

- Resolver/Loader

  Hier werden die Klassen-Dateien geladen und der Speicher wird konfiguriert. Es gibt zwei JVM Typen: Eager (Referenzen etc. werden beim Laden der '.class' Datei geladen) und Lazy (Referenzen etc. werden erst bei einer Instanziierung erstellt). Statische Initialisierung kann parallel und vor Benutzung der Klasse erfolgen.

- Bytecode Verification

  Diese Komponente verifiziert grundlegende Anforderungen an Byte-Code für eine korrekte Ausführung. Dies geschieht nachdem eine Klasse geladen wurde, jedoch vor der statischen Initialisierung. Die Checks beinhalten: Typen-Checks, Casts, Konvertierungen, Sichtbarkeit, ...

- Bytecode Interpreter

  Der Bytecode Interpreter "führt aus"/interpretiert den Byte Code mittels eines Stacks/einer Menge von Registern. Dies ist typischerweise langsam, wird jedoch vom JIT Compiler beschleunigt.

- JIT Compiler

  Um die Ausführung zu Beschleunigen wird Byte-Code während der Ausführung dynamisch zu Maschinen-Code kompiliert. Da dies aufwendig ist, geschieht es nur für häufig genutzte Code-Segmente, wie häufig aufgerufene Methoden.

- Memory Allocators

  Wenn Objekte erstellt werden, erhält der Memory Allocator Speicher vom OS. Dieser Speicher wird dann weiter intern verwaltet.

- Garbage Collectors (GC)

  JVMs verwenden verschiedene Garbage Collector Algorithmen, welche häufig parallel ausgeführt werden. Der Garbage Collector wird regelmäßig aufgerufen und gibt Speicher, welcher vom Programm nicht mehr benötigt wird, wieder frei. Die 'finalize()' Methode eines Objektes wird aufgerufen, bevor der GC es löscht.

  Der GC war lange Zeit eine der fehlerhaftesten Komponenten von Java bzgl. parallelen Programmen. Die korrekte Implementation eines GCs für parallele Progarmme ist sehr schwer.

- Portability Layer

  Dies ist notwendig, da bestimmte Java-Strukturen nicht direkt auf eine Plattform abgebildet werden können. Dies liegt u. A. daran, dass unterschiedliche Plattformen unterschiedliche Konzepte unterstützen. Die Portability Layer implementiert die Java-Konzepte mittels auf der Plattform zur Verfügung stehen Instruktionen.

- Native Interface

  Das Native Interface ermöglicht es, Plattform-spezifische/-native Methoden und Module zu nutzen. Diese Komponente übernimmt die Konvertierung von Daten und Parametern, um die lokalen Binaries korrekt aufrufen zu können.

2 Threads (in Java)

Ein Prozess entspricht einem Programm, welches in einem OS ausgeführt wird. Prozesse können (und werden meist) parallel ausgeführt. Jeder Prozess hat seinen eigenen Kontext, welcher aus Werten im Speicher, Werte in Registern, Ressourcen Zugriff (wie Zugriffsberechtigungen), Instruction Pointer, ... besteht.

Ein Prozess besteht wiederum aus einzelnen Threads. Diese können genau wie Prozesse selbst parallel ausgeführt werden. Allerdings teilen sich diese Threads einen Kontext und Ressourcen. Sie sind die parallelen Bestandteile eines Programms und nicht mehrere Programme.

Wenn ein Thread erstellt wird, muss er warten bis er von dem OS-Scheduler ausgeführt wird. Wenn der Prozess vom Scheduler ausgewählt wird, wird er für eine gewisse zeit ausgeführt. Wenn der OS-Scheduler z. B. auf I/O warten muss, wird die Ausführung blockiert. Sobald die "Abhängigkeit" aufgelöst ist, muss der Prozess erneut warten um

ausgeführt zu werden. Ein Prozess muss auch warten, wenn der OS-Scheduler einen anderen Prozess ausführen möchte. Der Übergang zwischen diesen States wiederholt sich, bis der Prozess terminiert wird.

Das Betriebssystem erfüllt somit mehreren Aufgaben:

- Prozesse/Threads starten
- Prozesse/Threads terminieren (um Ressourcen freizugeben)
- Ressourcen-Nutzung kontrollieren (alleinige der Nutzung vermeiden)
- CPU-Zeit zuweisen
- Kommunikation zwischen Prozessen ermöglichen
- Synchronization zwischen Threads ermöglichen

Wenn das OS zwischen zwei Prozessen wechselt, muss ein "Context Switch" durchgeführt werden. Der Context ist in PCB (Process Control Blocks) gespeichert. Process Level Parallelism, also das wechseln zwischen Prozessen, ist komplex und aufwendig. Das wechseln zwischen Threads ist weniger aufwendig, da kein Kontextwechsel notwendig ist. Das wechseln des Kontexts beinhaltet immer einen Overhead.

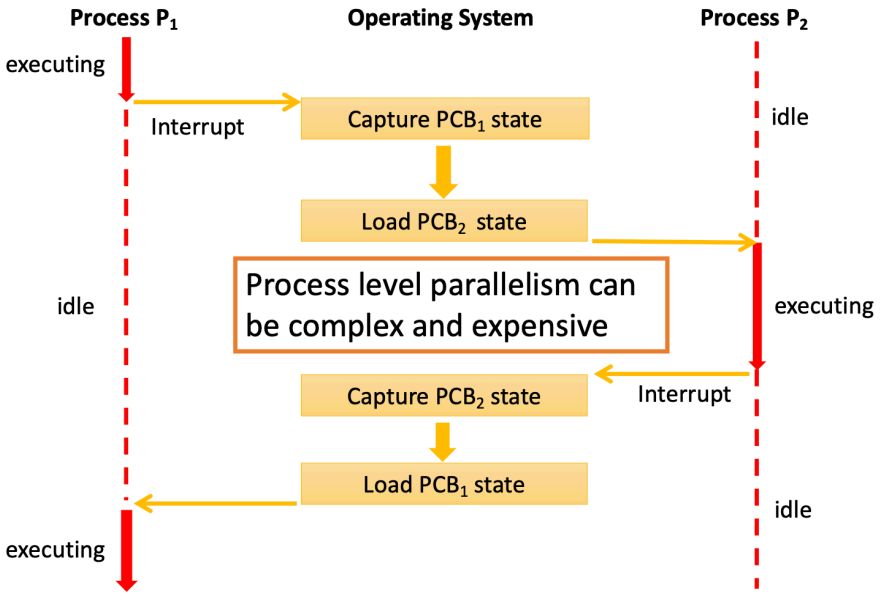


Figure 2: Process Level Parallelism & Context Switches

Multitasking Multitasking beschreibt die concurrent Ausführung von Prozessen/Threads. Auf einem Kern wird dies durch Multiplexing ermöglicht. Auf mehreren Kernen kann dies wirklich parallel geschehen.

(Time) Multiplexing erweckt den Anschein von Parallelität. Die CPU/ein Kern wechselt schnell zwischen verschiedenen Prozessen (durch Kontext-Wechsel) oder Threads. Da dies sehr schnell passiert, entsteht der Eindruck, dass die Prozesse/Threads parallel ausgeführt würden. Multiplexing ermöglichst asynchrone Operationen (wie I/O Operationen).

Denn währen eine Operation ausgeführt wird, kann einfach ein anderer Prozess/Thread ausgeführt werden.

Multithreading Threads sind unabhängige Ausführungen, welche im gleichen Prozess laufen. Threads eines Prozesses teilen den Kontext (wie den Speicher-Addressraum) und können einfach miteinander kommunizieren. Zwischen Threads zu wechseln benötigt keinen Kontext-Wechsel (speichern & laden eines PCBs).

Wenn nur ein Kern zur Verfügung steht ist die Ausführung nicht wirklich parallel, aber kann nichtsdestotrotz concurrent sein. Diest wird durch Multiplexing wie gerade erläutert ermöglicht.

2.1 Java Threads

In Java sind Threads eine Menge von Instruktionen, welche in einer spezifischen Reihenfolge ausgeführt werden. Threads werden in Java mit der ‘Thread‘ (‘java.lang.Thread‘) Klasse implementiert, welche das Interface ‘Runnable‘ implementiert. Die Klasse beinhaltet mehrere ”native”-Methoden, welche aus Effizienzgründen in C implementiert sind.

Wichtige Methoden der ‘Thread‘ Klasse sind folgende. Wenn man ein ‘Therad‘ Objekt erstellt ist es im Zustand NEW. In diesem Zustand wird der Thread noch nicht ausgeführt.

- ‘start()‘ - erstellt und startet einen neuen Thread
- ‘run()‘ - führt die Instruktionen des Threads aus, erstellt jedoch KEINEN neuen Thread sondern nutzt den aktuellen Thread
- ‘interrupt()‘ - stoppt und erzeugt eine Exception im Thread

In Java hat ein Thread verschiedene Zustände.

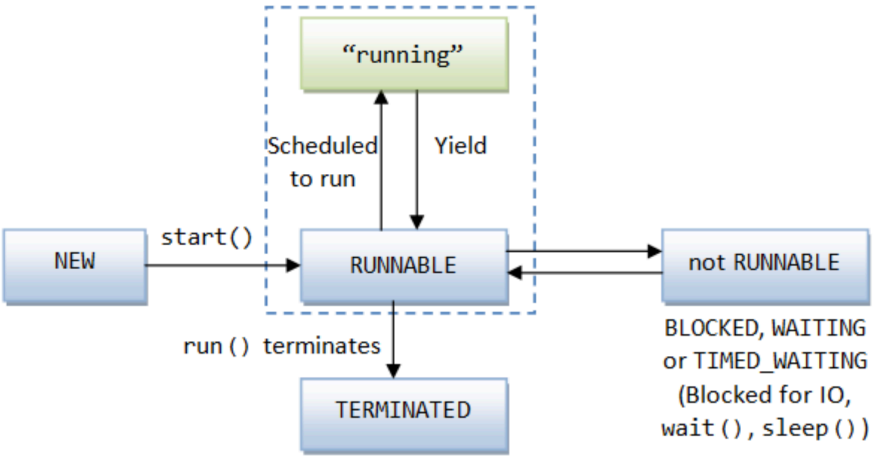


Figure 3

In Java gibt es einen main Thread des Prozesses, welcher die ‘main()‘ Methode ausführt. Selbst wenn der main Thread terminiert, laufen andere Threads, welche erzeugt wurden, weiter. Ein Programm/Prozess terminiert, wenn alle ‘non-deamon‘ Threads terminieren. Deamon-Threads sind ”Service-Threads” wie der Garbage Collector, welche von der JVM zur Verfügung gestellt werden. Diese können auch nach Programmterminierung weiterlaufen.

Man kann ‘Thread.currentThread()‘ nutzen, um ein Objekt mit Informationen über den aktuellen Thread zu erhalten. Es stehen folgende Methoden zur Verfügung:

- ‘getId()‘
- ‘setName(String s)‘
- ‘setPriority(Thread.MAX_PRIORITY)‘ for instance - Priorität (1-10) für den Scheduler setzen
- ‘getState()‘ - returns z. B. ‘State.TERMINATED‘

Bemerkt, dass die Priorität normalerweise einen Effekt hat. Es gibt jedoch keine Garantie das ein Thread mit höherer Priorität zuerst gescheduled wird.

Auf dem Objekt eines Threads kann ‘join()‘ aufgerufen werden. Der Thread, welcher ‘join()‘ aufruft, ist dann so lange BLOCKIERT/NOT RUNNABLE bis der andere Thread terminiert. Aähnlich kann ‘.wait()‘ bzgl. Locks (weiter unten betrachtet) gesehen werden. Auch heir wird der Thread NOT RUNNABLE. Allerdings nun nicht bis der Thread terminisiert, sondern bis ein ‘nofity()‘/‘notifyAll()‘ augerufen wird.

Bemerke, dass es zu Problemen kommen kann (wahrscheinlich), wenn ein Thread ein Lock hält und währenddessen terminiert (z. B. durch eine Exception). Je nach implementation wird das Lock nicht mehr freigegeben (Deadlock) oder die geschützte Datenstruktur ist inkonsistent.

2.1.1 Threads erstellen

Eine ältere Möglichkeit ist: Wir erstellen eine Subklasse von ‘Thread‘ und implementieren die Methode ‘run()‘, welche beim Starten des Threads ausgeführt wird. Wir können den Thread ausführen, indem wir ein Objekt der Subklasse erstellen und ‘start()‘ auf diesem aufrufen.

```
1  public class MyThread extends Thread {
2    @Override
3    public void run() {
4      // Code, der im neuen Thread ausgeführt wird
5      System.out.println("Neuer Thread wird ausgeführt.");
6      }
7
8      public static void main(String[] args) {
9          // Erstellen und Starten eines neuen Threads
10         MyThread myThread = new MyThread();
11         myThread.start();
12
13         // Code, der im Hauptthread ausgeführt wird
14         System.out.println("Hauptthread wird fortgesetzt.");
15     }
16 }
```

Eine neuere Methode ist: Wir erstellen eine Klasse, welche Runnable implementiert. Ein Objekt dieser Klasse übergeben wird dem Konstruktor von Thread. Erneut starten wir wie gewoht mit ‘start()‘.

```
1  public class MyRunnable implements Runnable {
2      @Override
3      public void run() {
```

```
4           // Code, der im neuen Thread ausgeführt wird
5           System.out.println("Neuer Thread wird ausgeführt.");
6       }
7
8       public static void main(String[] args) {
9           // Erstellen eines Runnable-Objekts
10          MyRunnable myRunnable = new MyRunnable();
11
12          // Erstellen eines Thread-Objekts und Übergabe des Runnable
                -Objekts als Argument
13          Thread myThread = new Thread(myRunnable);
14
15          // Starten des neuen Threads
16          myThread.start();
17
18          // Code, der im Hauptthread ausgeführt wird
19          System.out.println("Hauptthread wird fortgesetzt.");
20      }
21  }
```

2.1.2 Threads joinen

Nun möchten wir warten, bis ein anderer Thread terminiert. Durchgängig (in einer while-Schleife) zu testen, ob ein Thread bereits terminiert hat, ist sehr ineffizient. Dies nennt man busy waiting.

Beim Joining schläft der wartende Thread und wird vom Scheduler wieder aufgewacht, wenn der andere Thread terminiert ist. Da joining typischerweise einen Kontext-Wechsel overhead beinhaltet, kann busy waiting für kurzlebige Threads performanter sein.

2.1.3 Exception

Wenn in einem Thread eine Exception auftritt, können wir dies in der Konsole sehen. Denn der Standard-Exception Handler druckt eine Fehlermeldung. Doch die anderen Threads sind davon nicht beeinträchtigt und erfahren nichts vom Fehler des anderen Threads. Selbst wenn ein Thread den Thread joined, welcher eine Exception wirft, terminiert die ‘join()‘ Methode ohne Exception.

Eine Möglichkeit damit umzugehen ist, dass der Thread, welcher die Exception wirft, selbst ‘try-catch‘ Blöcke verwendet, um den Fehler zu beheben.

Alternativ kann man einen ‘UncaughtExceptionHandler‘ setzen. Mit diesem können wir ‘uncaught‘ Exceptions manuell handhaben. Einen solchen Handler kann man mit einem einzelnen Thread registrieren:

```
1   Thread thread = new Thread(new Runnable() {
2       public void run() {
3           // Code that may throw an uncaught exception
4       }
5   });
6
7   thread.setUncaughtExceptionHandler(new Thread.
        UncaughtExceptionHandler() {
8       public void uncaughtException(Thread t, Throwable e) {
9           // Handle the uncaught exception here
10          System.out.println("Uncaught exception in thread: " + t.
                getName());
```

```
        e.printStackTrace();
    }
});

thread.start();
```

Man kann ihn als defaultUncaughtExceptionHandler setzen:

```
  Thread.setDefaultUncaughtExceptionHandler(new Thread.
     UncaughtExceptionHandler() {
    public void uncaughtException(Thread t, Throwable e) {
        // Handle the uncaught exception here
        System.out.println("Uncaught exception in thread: " + t.
           getName());
        e.printStackTrace();
    }
});
```

Man kann ihn für eine ‘ThreadGroup‘ definieren:

```
ThreadGroup group = new ThreadGroup("MyThreadGroup");

// Set the uncaught exception handler for all threads in the group
group.setUncaughtExceptionHandler(new Thread.
   UncaughtExceptionHandler() {
    public void uncaughtException(Thread t, Throwable e) {
        // Handle the uncaught exception here
        System.out.println("Uncaught exception in thread: " + t.
           getName());
        e.printStackTrace();
    }
});

Thread thread1 = new Thread(group, new Runnable() {
    public void run() {
        // Code for thread 1
    }
});

Thread thread2 = new Thread(group, new Runnable() {
    public void run() {
        // Code for thread 2
    }
});

// Start the threads
thread1.start();
thread2.start();
```

Eine weiter Möglichkeit ist es Interrupts zu nutzen. Wir können für einen Thread einen zweiten Thread erstellen, welcher den ersten Thread überwacht. Z. B., wenn der erste Thread zu lange braucht, kann der zweite Thread den ersten mit einem Interrupt unterbrechen. Dafür nutzt man die Methode ‘interrupt()‘ auf einem Thread Objekt. Dies führt dazu, dass der erste Thread terminiert.

```
public class InterruptExample {
    public static void main(String[] args) {
```

```
3           Thread mainThread = Thread.currentThread();
4
5           Thread thread = new Thread(() -> {
6               try {
7                   // Einige Berechnungen oder Aktivitäten
8                   Thread.sleep(5000); // Simuliert eine längere
                        Operation
9               } catch (InterruptedException e) {
10                  System.out.println("Thread unterbrochen.");
11              }
12          });
13
14          thread.start();
15
16          // Haupt-Thread unterbricht anderen Thread nach 2 Sekunden
17          try {
18              Thread.sleep(2000);
19              thread.interrupt();
20          } catch (InterruptedException e) {
21              e.printStackTrace();
22          }
23
24          try {
25              // Warten, bis der andere Thread beendet ist
26              thread.join();
27          } catch (InterruptedException e) {
28              e.printStackTrace();
29          }
30
31          System.out.println("Programm beendet.");
32      }
33  }
```

3 Shared Memory & Data Races

Gemeinsam genutzter Speicher, mit den Ergebnissen, dass Verschachtelungen schlecht sein können, und dass wir Datenrennen berücksichtigen müssen, ist der Hauptgrund dafür, dass parallele Programmierung schwierig ist.

Im Allgemeinen betrachten wir drei Zustände des Speichers:

- **immutability** - Daten können nicht verändert werden und somit ohne Probleme beliebig concurrent gelesen werden
- **isolated mutability** - Daten können verändert werden, jedoch hat immer nur ein Thread Zugriff auf die Daten
- **mutability/shared data** - Daten können verändert werden, alle Thread können jederzeit die Daten lesen und beliebig verändern

Im weiteren Verlauf werden wir Fork/Join (Cilk-style parallelism), parallel patterns, pack, etc. betrachten. Diese Ansätze lässt sich der Kategorie isolated mutability zuordnen, da durch das Konzept der Parallelisierung immer nur ein Thread Daten verarbeitet.

Weiterhin werden wir Locks betrachten. Diese sind relevant, wenn die Programmstruktur isolated mutability nicht implizit vorgibt. Dann können locks genutzt werden,

um ‘critical sections‘ zu schützen, sodass immer nur ein Thread auf bestimmte Ressourcen zugreifen kann. mutability/shared memory ist der schwierigste Fall und wir bei lock-free Parallelität näher betrachtet.

Allgemein sollte man immer immutability wählen, wenn möglich. Denn so wird für Parallelität kein Overhead hinzugefügt und die Programmierung ist sehr einfach. Ansonsten wird heute meist auf Locks zurückgegriffen, da diese sehr verbreitet und vergleichsweise einfach zu nutzen sind. Lock-free programming ist hingegen kompliziert und wird nur in kritischen Systemen (Flugzeugen, Atomkraftwerken, etc.) genutzt - wie später näher erläutert.

3.1 (Bad) Interleavings

(Bad) Interleavings sind der Grund, weshalb mutability in der Praxis häufig unzureichend ist. Denn wenn zwei Threads auf die gleiche Ressource zugreifen kann as zu verschiedensten Problemen kommen. Wenn man die Anweisungen eines Threads betrachtet, kann man diesen Thread sequentiell betrachten. Für jeden Thread individuell haben wir eine Garantie über die Ausführungsreihenfolge. Aber bei mehreren Threads kannder OS-Scheduler die Anweisungen der individuellen Threads in beliebiger Reihenfolge ausführen, so lange die Reihenfolge innerhalb eines jeden Threads erhalten bleibt.

Ein einfaches Beispiel betrachtet zwei Threads. Einer addiert und einer subtrahiert 1 auf/von einer Integer Variable. In Bytecode ist +1/-1 in mehrere Instruktionen aufgeteilt (laden der Variablen, addieren, schreiben). Wir möchten am Ende klar -1+1=0 Veränderun. Durch Bad interleaving können wir jedoch am ende auch +1 oder -1 erhalten.

```
1 x=0
2 Thread 1:
3   localX = x
4   lcoalX increment
5   x = localX
6 Thread 2:
7   localX = x
8   lcoalX decrement
9   x = localX
```

Ein (Bad) Interleaving beschreibt, dass mehrere voneinander abhängige Ausführungsströme überlagert ausgeführt werden (die Ausführungen/calls werden interleaved) - entweder auf unterschiedlichen Kernen oder durch multiplexing. Dies ist kein Problem, wenn die Ausführungen unabhängig sind. Wenn sie jedoch auf die gleichen Daten zugreifen kann es zu Problem kommen, bzw. Probleme sind sehr wahrscheinlich.

3.2 Races

Eine race condition tritt auf, wenn das Ergebnis der Berechnung vom spezifischen Scheduling der Instruktionen eines Ausführung abhängt. Es handelt sich um Probleme, die wegen nicht-deterministischem Timing und Ordnung von Ereignissen in concurrent Programmen auftreten. Race conditions existieren nur wegen concurrency (muss nicht zwingend Parallelität sein), da bei einem Thread keine (bad) interleavings auftreten können.

- **Data Race - Low Level Race Condition** (niedriges semantisches level)

  Fehlerhaftes Programmverhalten, das durch unzureichend synchronisierte Zugriffe mehrerer Threads auf eine gemeinsam genutzte Ressource verursacht wird, z. B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben desselben Speicherplatzes.

- **Bad Interleaving - High Level Race Condition** (hohes semantisches level)

  Fehlerhaftes Programmverhalten, das durch eine ungünstige Ausführungsreihenfolge eines Multithreading-Algorithmus verursacht wird, der ansonsten gut synchronisierte Ressourcen nutzt.

3.3 Shared Resources

Um unbeabsichtigtes Verhalten mehrerer Threads zu vermeiden, betrachten wir verschiedene "Hazards" für unsere Programme und deren Ausführungen.

- Thread Safety Hazard - "nothing bad ever happens" (in allen möglichen interleavings)
- Thread Liveness Hazard - "something good eventually happens" (z. B. dass kein einzelner Thread eine Ressource endlos blockiert)
- Thread Performance Hazard - das Ziel paralleler Programmierung ist natürich ein Performance Gewinn

Thread Safety Properties, also Eigenschaften, welche korrekte Ausführungen ermöglichen, beinhalten:

- keine data races (mehrere Threads schreiben auf die gleiche Ressource)
- mutual exclusion
- linearlizability
- atomicity
- schedule-determinism
- keine Deadlocks
- Invarianten

4 Parallel Architectures

Paralleles Rechnen kann sich auf viele Anwendungen beziehen: Mehrere Kerne auf einem Computer, mehrere CPUs in einem System, mehrere Computer in einer Einrichtung, verteiltes Rechnen auf einem "Supercomputer", ... Bei der parallelen Programmierung denken wir meist an die Entwicklung für ein System mit mehreren Kernen.

- Simultaneous Multithreading (Hyper-Threading, Intel-Begriff), i.e., multiplexing

  ein physischer Core (ggf. mehrere physische Cores), kein direkter paralleler Performancegewinn (besser als idle)

- Multicores

  mehrere Kerne, jeder Kern hat eigene Hardware, einige geteilte Ressourcen (u. A. Caches)

- Symmetric Multiprocessor System, i.e., multiple CPUs

  nur geteilter main memory für mehrere CPUs

- Non-Uniform Memory Access

  main memory is Verteilt (z. B. auf mehrere Computer in einem Netzwerk)

- Distributed Memory

  clusters, Rechenzentren, ... - wird mit Message Passing betrieben

Die Beschäftigung mit parallelen Architekturen ermöglicht es uns, besser zu verstehen, warum bestimmte Probleme bestehen, und ermöglicht es uns daher, die Software unter Berücksichtigung der bestehenden Hardwarebeschränkungen weiter zu optimieren.

Moderne Computer sind den frühen Computern konzeptionell noch sehr ähnlich. Die ersten Allzweckcomputer verwendeten die Harvard-Architektur, bei der Daten und Befehle getrennt wurden. Heute wird bei der (in Princeton entwickelten) Von-Neumann-Architektur ein gemeinsamer Speicher für Daten und Befehle verwendet. Im Kern entspricht das Von-Neumann-Modell der imperativen Natur der meisten Programmiersprachen. Wir betrachten Instruktionen, die semantisch sequentiell, also nacheinander, ausgeführt werden. Für parallele Ausführungen haben wir wenig bis gar keine Garantien. Dies hat historische Gründe, da die parallele Datenverarbeitung erst in den letzten 20 Jahren an Popularität/Relevanz gewonnen hat.

4.1 Memory Hierarchy

Seit der Erfindung der Computer wurden die Prozessoren immer schneller. In der Zwischenzeit konnte die Speichergeschwindigkeit nicht mit den Verbesserungen der Verarbeitungsgeschwindigkeit mithalten. Stattdessen wurde der Speicher immer größer. Daher ist der Speicherzugriff der größte Engpass in der heutigen Computerwelt.

Um dennoch eine angemessene Leistung zu ermöglichen, verwenden moderne Computer verschiedene Cache-Ebenen: Register, L1-Cache, L2-Cache, (L3-Cache,) Hauptspeicher und Festplatte. Damit wird das Prinzip der Datenlokalität umgesetzt. Die räumliche Lokalität beschreibt, dass auf eng beieinander liegende Daten oft gemeinsam zugegriffen wird, und die zeitliche Lokalität beschreibt, dass auf Daten oft wiederholt in kurzer Zeit zugegriffen wird.

Beachten Sie diese Geschwindigkeitsvergleiche. Sie sollen nur die Größenordnung der Geschwindigkeitsunterschiede aufzeigen. Die tatsächlichen Werte variieren je nach gewählter Hardware deutlich.

- Registers sind etwa 2x schneller als L1 (0.5 ns)
- L1 ist etwa 5x schneller als L2 (1 ns)
- L2 ist etwa 30x schneller als der Hauptspeicher (4-7 ns)
- Hauptspeicher ist etwa 350x schneller als Speicher/Festplatte (100-300 ns)

In Multi-Core Systemen sind der L1 und L2 Cache häufig privat. Threads von jedem Kernen greifen somit auf ihren eigenen Speicher zu. Nur der L3 Cache, memory disk, ... sind normalerweise geteilt. Bei mehren CPUs wird der main/system memory auch zwischen den CPUs geteilt, während der L3 cache jeweils für jede CPU privat ist.

Dies kann Probleme verursachen, wenn zwei Threads auf unterschiedlichen Kernen mit den gleichen Daten arbeiten. Denn beide laden diese Daten in ihren lokalen Cache. Auf diesen lokalen Daten werden dann Operationen ausgeführt und das Ergebnis zu einem unbestimmten Zeitpunkt aus dem Cache wieder in den main memory geschrieben. Das

kann dazu führen, dass Threads auf unterschiedlichen Daten bzw. jeweils nicht auf den neusten Daten operieren. Das kann zu Datenkorruption führen.

Dementsprechend nutzt man coherency protocols. Diese führen Kommunikation zwischen den Caches oder ein administrative Directory ein, um sicherzugehen, dass die neusten Daten zwischen den Caches synchronisiert werden. Ein solches Protokoll is MESI (modified, exclusive, shared, invalid), welches jedoch auch nicht alle Probleme lösen kann (prefetching, postponed writing, ...).

In Java können solche Probleme durch die Verwendung von Synchronisierungsinstruktionen gelöst werden. Vgl. Abschnitt zum Java Memory Model.

4.2 Parallel Execution

Wir betrachten drei grundlegende Ansätze, um mit Parallelität die Performance zu verbessern: Vectorization, ILP (Instruction Level Parallelism) und Pipelining. Ersteres ist sichtbar für den Entwickler/den Compiler, ILP ist nicht direkt sichtbar und Pipelining (als Execution Model) ist auch nicht direkt sichtbar, kann jedoch auf Softwareentwicklung übertragen werden.

4.2.1 Vectorization

Dies ist eine Form von SIMD: Single Instruction, Multiple Data - wir operieren auf mehreren Daten mit einer Instruktion. Wenn man auf mehreren Daten die gleiche Operation ausführen möchte, müsste man ohne SIMD/Vectorization das gleiche für jeden einzelnen Datenpunkt machen. Mit Vectorization kann man jedoch alle Daten gleichzeitig laden und die gleiche Operation auf allen Datenpunkten gleichzeitig ausführen.

Wir haben bereits Threads kennengelernt und entsprechend scheint es sich anzubieten, die Operation für jedes Element auf je einen Thread auszulagern. Das ist möglich, jedoch inperformant. Die Alternative ist 'embarrassingly parallel' - automatisierte Hardware Unterstützung.

Diese Beispiel stellt dar, was der Compiler mit der Hardware automatisiert macht. Bei unrolled wird der Code nicht wirklich so umformatiert. Stattdessen visualisiert es nur, dass jetzt ein Vektor der Länge 4 verarbeitet wird, i.e., die vier Zeilen werden als eine Instruktion ausgeführt.

```
1  SEQUENTIAL
2    for(int i = 0; i<16; i++)
3      c[i] = a[i] + b[i]
4  UNROLLED
5    for(int i = 0; i<16; i += 4) {
6      c[i+0] = a[i+0] + b[i+0]
7      c[i+1] = a[i+1] + b[i+1]
8      c[i+2] = a[i+2] + b[i+2]
9      c[i+3] = a[i+3] + b[i+3]
10   }
11 ASSEMBLY
12   r1 = Vload a, i, i+3
13   r2 = Vload b, i, i+3
14   r3 = Vadd r1, r2
15   Vstore c, i, r3
```

In C++/C kann man Vektoren für den Compiler manuell spezifizieren. Alternativ kann der Compiler bei bekannter Plattform auch erkennen, welche Instruktionen mit den

auf der Plattform verfügbaren Instruktionen vektorisiert werden können. In Java wird dies während der Ausführung durch den JIT Compiler erledigt.

4.2.2 ILP (Instruction Level Parallelism)

Moderne CPUs können mehrere Instruktionen gleichzeitig während einer Clock-Cycle verarbeiten. Dies ist möglich, wenn sich unabhängige Instruktionen finden lassen. Solche CPUs werden superscalar CPUs genannt. (DDCA: Superscalar nur bei bestimmter Form von Parallelität.) Man kann ILP weiter ausreichen:

- Speculative Execution

  Wenn Rechenkapazität aktuell nicht voll ausgelastet sind, können Instruktionen ausgeführt werden, für welche noch nicht alle Daten vorhanden sind/welche ggf. überhaupt nicht ausgeführt werden müssen. Denn im Fall, dass man korrekte Instruktionen ausgeführt hat, ist dies ein Performance gewinn. Andernfalls nicht sehr schlimm, da man nur überschüssige Ressourcen genutzt hatte.

- Out-of-Order Execution

  Wenn Instruktionen nicht voneinander abhängig sind, ist die Ausführungsreihenfolge nicht wichtig. Entsprechend kann die CPU die Ausführungsreihenfolge von unabhängigen Instruktionen einfach ändern.

- Pipelining (below)

4.2.3 Pipelining

Eine Pipeline besteht aus verschiedenen Stages. Eine Stage kann aus mehreren Instanzen bestehen, sodass mehrere Inputs gleichzeitig in einer Stage sein können. Alle Inputs müssen alle Stages durchlaufen. Jede Stage benötigt eine gewisse Zeit, um ausgeführt zu werden. Zum Beginn sind natürlich nicht alle Stages genutzt. Bis alle Stages genutzt werden, spricht man vom lead in. Nach der full utilization, wenn es keine neuen Inputs gibts, spricht man vom lead out, bis die Pipeline leer ist/alle Stages frei sind.

Eine Pipeline wird anhand mehrere Metriken analysiert:

- Throughput (larger is better) - Anzahl der beendeten Inputs/Zeit.
- Latency (lower is better) - Benötige Zeit, um einen Input zu bearbeiten (nicht zwingend konstant)

Eine Grenze (untere) für den Throughput ist $\frac{1}{\max(\text{computation times alles Stages})}$. Eine (obere) Grenze für die Latency is $\#stages \cdot \max(\text{computation times aller Stages})$.

Ist die Latenz konstant, so wird die Pipeline als balanciert bezeichnet. Andernfalls ist die Latenz nicht beschränkt und wächst immer weiter an. Man kann die Länger aller Stages (bzw. der ersten Stage) auf die größte Latenz setzen, um eine balancierte Pipeline zu erhalten. Man kann an Stellen mit Bottlenecks ebenfalls substages einführen oder mehrere Instanzen einer Stage nutzen, um ggf. den Throughput zu erhöhen/die Latenz zu verringern.

In der Theorie kann man eine ideale Pipeline bauen. In der Praxis wird es jedoch immer einen Overhead geben, um eine weitere Stage einzuführen, da Informationen zwischen den Stages gespeichert/gelesen werden müssen etc.

Eine moderne CPU verarbeitet Instruktionen auch mittels einer Pipeline. Die Stages dort sind konzeptuell:

- Instruction Fetch - Lädt die nächste Instruktion vom Speicher und prefetched ggf. zusätzliche Instruktionen
- Instruction Decode - Bereitet Instruktionen auf die Ausführung vor, indem Kontrollsignale für die nächsten Stages erzeugt werden und ggf. bereits Daten geladen werden.
- Execution - Führt die Instruktion aus
- Memory Access - lädt Daten vom Speicher oder speichert Werte in den Speicher
- Writeback - schreibt das Ergebnis (falls gegeben) in ein Register

Lange Zeit konnten CPU Architekten die Ausführungsgeschwindigkeit durch eine höhere Transistorendichte etc. exponentiell steigern. Während wir es heute weiterhin schaffen exponentiell wachsend Transistoren zu verwenden, sind wir an einige Grenzen/Hürden für Performance gestoßen

- Energie-/Wärmeabgabe - Kühlung nicht mehr möglich
- Speicher - Transport von Daten dauert am längsten und verbraucht am meisten Energie
- ILP - keine beliebige Skalierung von Parallelität möglich

Um diesen Herausforderungen zu begegnen beschäftigt man sich mit paralleler Programmierung und parallelen Systemen.

4.3 Registers

Ein Register ist ein fundamentales Speicherobjekt, welches geteilt oder nicht geteilt sein kann. Ein Register r unterstützt ‘r.read()‘ und ‘r.write()‘.

Single Writer Multiple Reader (SWMP) Register sind solche, bei welchen immer nur ein thread schreibt, jedoch mehrere lesen können. Safe und regular registers sind hier SWMP Registers. Bei atomic registers kann es auch mehrere Schreiber geben.

4.3.1 Safe Registers

- Jedes read, welches nicht concurrent mit einem write ist, gibt den aktuellen Wert des Registers zurück.
- Jedes read, welche concurrent mit einem write ist, kann einen beliebigen Wert der Domain des Registers (alle potenziell speicherbaren Werte) zurückgeben.

4.3.2 Regular Registers

- Jedes read, welches nicht concurrent mit einem write ist, gibt den aktuellen Wert des Registers zurück.
- Jedes read, welches mit einem write concurrent ist, kann einen von zwei Werten zurückgeben: den alten oder den neuen Wert.

4.3.3 Atomic Registers

Eine Ausführung J einer dieser Methoden nimmt immer an einem einzigen Punkt $\tau(J)$ Effekt, wobei $\tau(J)$ immer zwischen dem Start und Ende von J liegt. Zwei Operationen auf einem Atomic Register haben immer eine unterschiedliche effect time. Ein Aufruf von 'r.read()' gibt den Wert zurück, welcher von dem am nächsten zurückliegenden 'r.write()' Aufruf geschrieben wurde, zurück.

Bemerke, dass eine totale Ordnung von τ-Werten existiert. Diese Ordnung muss jedoch nicht deterministisch sein, da es keine Garantien zum Ausführungszeitpunkt von τ innerhalb einer Ausführung J gibt.

4.4 Hardware Support for Parallelism / Read-Modify-Write Operations

Wir führen nun zusätzliche Hardwarestrukturen ein, die für lock-free programming und eine effiziente Implementierung von locks ermöglichen. Hierbei handelt es sich um eine Erweiterung der architectural capabilities.

Wie bereits durch die 'Read-Modify-Write Operations' angedeutet, basiert diese Hardware-Unterstützung auf Operationen, die read und write als eine atomare Operation durchführen. Auf x86/amd64 haben wir CMPXCHG (compare and exchange) und LOCK (prefix für andere Operatoinen. In ARM existieren u. A. LDREX (load linked) und STREX (store conditional).

Abstrakter ausgedrückt, fallen diese Low-Level-Beispiele in einige abstrakte Kategorien, die in modernen Architekturen zu finden sind.

- TAS (Test-And-Set) - TSL register/flag (Motorola 68000)
- CAS (Compare-And-Swap) - LOCK, CMPXCHG (Intel x86), CASA (Sparc)
- Load Linked/Store Conditional - LDREX, STREX (ARM), LL, SC (MIPS, POWER, RISC V)

Wir wollen solche "atomaren" Operationen vermeiden, weil sie etwa 100 Mal langsamer sind als normale Operationen.

```
1  boolean TAS(memref s): // demonstration semantics
2    if (mem[s] == 0):
3      mem[s] = 1
4      return true
5    else:
6      return false
7  int CAS(memref a, int old, int new):
8    oldval = mem[a]
9    if (old == oldval):
10     mem[a] = new
11   return oldval // alternatively return whether
12         the new value was adopted
```

JVM Bytecode bietet keine Read-Modify-Write Operationen wie CAS direkt an. Allerdings können wir 'Atomic'-Klassen benutzen, um ähnliches Verhalten nutzen zu können. Z. B. mit 'java.util.concurrent.atomic.AtomicBoolean' und folgenden Funktionen:

```
1  boolean set();
2  boolean get();
3  boolean compareAndSet(boolean expect, boolean update);
4  boolean getAndSet(boolean newValue);
```

Die JVM garantiert kein direktes Mapping, d.h. diese Operationen könnten immer noch mit locks ausgeführt werden. Jedoch gibt es die (nicht dokumentierte) Klasse 'sun.misc.Unsafe', welche ein solch direktes Mapping erlaubt.

Ein einfaches TAS Lock in Java ist:

```
1  public class TASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3
4    public void lock() {
5      while(state.getAndSet(true)) {}
6    }
7
8    public void unlock() {
9      state.set(false);
10   }
11
12   ...
13 }
```

Bei vielen threads nimmt die Performance durch die hohe contention jedoch ab. Dies lösen wir durch zwei Ideen: TATAS an Stelle von TAS und exponential backoff. TATAS steht für Test-And-Test-And-Set. Zunächst testen wir also so lange nur, bis wir einen validen Rückgabewert erhalten, erst dann versuchen wir die TAS Operation. Dies hilft, da Testen weniger Ressourcen benötigt als TAS.

Exponential Backoff dient der Reduktion der contention durch exponentiell länger werdende Pausen zwischen den TAS Versuchen. Durch die Kombination von exponential backoff mit TATAS kann die Dauer/Thread praktich konstant gehalten werden.

```
1  public void lock() {
2    Backoff backoff = null;
3    while (true) {
4      while (state.get()) {};  // spin reading only (TATAS)
5      if (!state.getAndSet(true))  // try to acquire,
6              returns previous val
7        return;
8      else { try {
9        if (backoff == null) // backoff on failure
10         backoff = new Backoff(MIN_DELAY, MAX_DELAY);
11       backoff.backoff();
12     } catch (InterruptedException ex) {} }
13   }
14 }
15 class Backoff {
16   ...
17
18   public void backoff() throws InterruptedException {
19     int delay = random.nextInt(limit);
20     if (limit < maxDelay) { // double limit if less than max
21       limit = 2 * limit;
22     }
23     Thread.sleep(delay);
24   }
25 }
```

5 Basic Concepts of Parallelism

Zunächst muss der Unterschied zwischen parallelism und concurrency dargestellt werden. Parallelism beschreibt, dass mehrere Sachen gleichzeitig getan werden, z. B. indem man mehrere Kerne zum schnelleren Lösen eines Problems verwendet. Bei concurrency handelst es sich hingegen, wenn man mehrere Dinge/Daten gleichzeitig verwaltet, z. B. geteilte Ressourcen. Concurrency kann auch mit einem Kern existieren.

Wenn man Programme auf mehreren Kernen/Prozessoren ausführt gibt es einige grundlegende bewährte Ansätze:

- Variablen lokal lassen
- Aliasing vermeiden
- Variablen unveränderbar/'final' deklarieren

Exceptions wurden in der sequentiellen Programmierung als eine hilfreiche Methode zur Vermeidung und Lokalisierung von Fehlern eingeführt. In der parallelen Programmierung sind diese jedoch leider nur begrenzt hilfreich. Dies liegt daran, dass der Stack Trace die Ursache einer Exception nicht immer vollständig wiedergibt, da die Ursache der Exception in einem anderen Thread liegen kann, welcher ggf. schon länger zurück liegt.

Letztlich muss man sich bewusst werden, dass Schleifen, If-Bedingungen etc. in parallelen Programmen nicht wie durch sequentielle Prorgrammiererfahrung intuitiv erwartet funktionieren. Denn die Variablen/Daten, welche eine Bedgingung (nicht) erfüllen, können ggf. von anderen Threads während der Ausführugn des Blocks verändert werden.

Wir betrachten nun grundlegende Möglichkeiten, um parallele Programme und deren Performance zu charakterisieren. Im weiteren Verlauf des Skriptes werden dann konkrete Ansätze für die parallele Programmierung thematisiert, welche die zuvor beschriebenen Herausforderungen versuchen bestmöglich zu lösen.

Parallelität kann explizit/manuell oder implizit (automatische Optimierungen durch das System) geschehen.

5.1 Work Partitioning & Scheduling

Work Partitioning beschreibt das Aufteilen von Arbeit in einzelne tasks/threads durch den Programmierer. Ein task ist eine Einheit von Arbeit. Die Anzahl der Arbeitseinheiten sollte mindestens so groß wie die Anzahl der Kerne sein, um idle Hardware zu vermeiden.

Beim scheduling handelt es sich um das Zuweisen von Arbeitseinheiten/tasks an einzelne Kerne/processors. Dies wird typischerweise vom System mit dem Ziel erledigt, dass keine Kerne idle laufen.

Es stellt sich die Frage der task granularity. Fine granularity (viele kleine Arbeitspakete) ist gut für portable Programme, die auch effizient auf Systemen mit vielen Kernen ausgeführt werden können. Coarse granularity reduziert hingegen den scheduling overhead. Allgemein gilt, dass man coarse granularity möchte, jedoch signifikant mehr Arbeitspakete als Kerne, um idle Hardware zu vermeiden.

5.2 Scalability

Skalierbarkeit betrachtet, wie gut ein System asymptotisch mit großen Input-Mengen umgeht. Wir betrachten dazu den Speedup - die Geschwindigkeitssteigerung bei steigender Anzahl Prozessoren ($\rightarrow \infty$).

Kernwerte sind:

- T_1 - sequentielle Ausführungszeit
- T_p - Ausführungszeit auf p Kernen/Prozessoren
- $S_p := \frac{T_1}{T_p}$ - parallel Speedup
- $E := \frac{S_p}{p}$ - Effizienz

Wenn man den Speedup berechnet, kann man T_1 auf unterschiedliche weisen bestimmen. Man kann dan parallele Programm einfach auf einem Kern ausführen, dann erhält man den relativen Speedup. Wenn man jedoch ein ein optimiertes sequentielles Programm verwendet erhält man den absoluten speedup.

Mit steigendem Speedup sinkt die Effizienz. Algorithm mit $< 50\%$ Effizienz werden üblicherweise verworfen.

Wir klassifizieren den Speedup:

- $S_p = p$ - linear speedup (Perfektion!)
- $S_p < p$ - sub-linear speedup, Performanceverlust (Normalfall)
- $S_p > p$ - super-linear speedup, scheint nicht intuitiv (nur in Sonderfällen)

Normalerweise haben wir sub-linearen speedup. Dies hat verschiedene Gründe.

- Programme haben nur begrenzte Parallelität (immer noch viele sequentielle Programmteile)
- Parallelität fügt einen Overhead (für die Kommunikation/Synchronization) hinzu
- Grenzend der Architektur/Hardware (z. B. Geschwindigkeitslimits beim Speicherzugriff)
- Datenstrukturen (Linked List iterieren anstatt Tree zu nutzen)

5.2.1 Amdahl's Law

Amdahl's Law beschreibt eine Obergrenze für den Speedup bei unendlich vielen Ressourcen. Die Ausführzeit von T_1 kann in W_{seq} (sequentielle Arbeit) und W_{par} (parallele Arbeit) aufgeteilt werden. Dann haben wir $T_p \geq W_{seq} + \frac{W_{par}}{p}$. Für sen speedup erhalten wir dann $S_p \leq \frac{W_{seq}+W_{par}}{W_{seq}+\frac{W_{par}}{p}}$ bzw. $S_p \leq \frac{1}{f+\frac{1-f}{p}}$ mit f als der Anteil der nicht-parallelisierbaren Arbeit, i.e. $\frac{W_{seq}}{W_{seq}+W_{par}}$. Für $p \to \infty$, we get $S_\infty \leq \frac{1}{f}$.

Amdahl's law ist eine negative Sichtweise auf Parallelität da es eine nicht überwindbare Grenze von parallelen Programmen darstellt.

5.2.2 Gustafson's Law

Dies liefert eine optimistischere Perspektive. Anstatt die Arbeit konstant zu lassen und eine geringe Laufzeit zu betrachten, wird bei Gustafson's Law die Laufzeit unverändert gelassen und mehr Arbeit für die gleiche Laufzeit betrachtet (die Problemgröße wird vergrößert).

$W = p(1-f)T_{wall} + f \cdot T_{wall}$ & $S_p = f + p(1-f) = p - f(p-1)$.

6 Parallel "Methods"

Im Kern geht es bei Divide-and-Conquer darum, ein Problem in x Teilprobleme aufzuteilen, die unabhängig voneinander gelöst und später zur Gesamtlösung kombiniert werden können. In einer einfachen Implementierung würden wir x Threads erstellen, 'start()' für jeden aufrufen, 'join()' für jeden aufrufen, die Lösung lesen und das Ergebnis berechnen.

Dies wird Fork/Join genannt, weil wir vom Haupt-Thread/Problem forken und die Teillösungen joinen. Solche Programme haben kaum Probleme mit gemeinsamem Speicher. Wir müssen nur Datenwettläufe beim Lesen der Ergebnisse von jedem Thread vermeiden.

Einige Probleme sind die reusability und efficiency über mehrere Plattformen hinweg sowie die Tatsache, dass verschiedene Zweige eine unterschiedliche Komplexität aufweisen, was zu einem Ungleichgewicht der Last führt und eine nahezu linear speedup verhindern kann. Um diese Probleme zu lösen, können wir die Anzahl der Threads drastisch erhöhen, weit über die Anzahl der Kerne/Prozessoren hinaus. Zu diesem Zweck verzichten wir auf Java-Threads, die einen großen Overhead verursachen. Dies ist vorwärts portable, es ist immer Arbeit verfügbar (keine ungenutzte Hardware), und das Ungleichgewicht der Last wird weniger zu einem Problem, da wir nicht auf einen langsamen Thread warten müssen, sondern an anderen Threads weiterarbeiten können.

6.1 Divide-and-Conquer

Divide and Conquer ist ein gut bekanntes Konzept.

```
1 if cannot divide:
2   return unitary solution (stop recursion)
3 divide problem in two
4 solve first (recursively)
5 solve second (recursively)
6 combine solutions
7 return result
```

Eine Schwierigkeit beim Teilen und Beherrschen sind unregelmäßige Arbeitsbelastungen. Es gibt jedoch Frameworks, die zeitintensive Arbeit auf bestehende Threads umverteilen und die Verwaltung für uns übernehmen.

Außerdem führt das ständige Erstellen eines neuen Threads zu einem spürbaren Overhead, da Java-Threads auf OS-Threads abgebildet werden. Da dies sehr ineffizient ist, geht uns möglicherweise der Speicher aus, bevor die Berechnung abgeschlossen ist. Um den Overhead zu reduzieren, sollten wir

- einen sequential cutoff verwenden: Führen Sie einen angemessenen Teil der Arbeit in den Threads der letzten Stufe aus, um zu vermeiden, dass ein Thread nur für die Addition von beispielsweise zwei ganzen Zahlen zuständig ist.
- Wenn wieder geteilt werden muss, sollte man nur einen neuen Thread erstellen und die anderen Berechnungen lokal durchführen. Dadurch wird die Anzahl der Threads um 50 % reduziert.

Um nahezu ideale Leistungssteigerungen zu erzielen (von $\mathcal{O}(n)$ auf $\mathcal{O}(\log n)$), müssen wir auch die Ergebniskombination als Teil der Überwindung betrachten.

6.2 Scheduling Tasks

Bisher haben wir die Verwendung von Java-Threads in Betracht gezogen, was sehr ineffizient ist. Ein alternativer Ansatz ist die Planung von Aufgaben auf Threads. Wir haben einen Pool von Threads und einen Satz von Aufgaben. Wir weisen den Threads kontinuierlich Aufgaben zu. Auf diese Weise können wir Fortschritte erzielen, ohne eine riesige Anzahl von Threads zu erstellen und den Overhead vom Threadmanagement vermeiden.

In Java ist dies mit dem 'ExecutorService' umgesetzt, welcher ein Interface zum submitten von tasks zur Verfügung stellt. Diese werden auf den Threads the Pools, welche 'ThreadPoolExecutor' implementieren, ausgeführt. Der ExecutorService hat zwei wichtige Methoden:

- '.submit(Callable⟨T⟩task) → Future⟨T⟩'
- '.submit(Runnable task) → Future⟨?⟩'

Einen ExecutorService erstellt man mit 'ExecutorService esx = Executors.newFixedThreadPool(n);' wobei n die Anzahl der Threads im Pool ist. After having finished working with the ExecutorService, one should call 'esx.shutdown()'.

When having some 'Future' object, one can use '.get()' to wait until the execution finishes and get the result (if exists).

Notice that this does not work in Java if we have deep recursions/divide-and-conquer splits. That is as the pool can only accommodate n threads. If we have n threads now waiting for their children to finish, we have a deadlock as the children can not proceed to execution.

6.3 Cilk style parallelism

Cilk-style parallel programming ist eine Form von Divide-and-Conquer. Dabei wird ein Problem in kleinere unabhängige Probleme unterteilt, welche concurrent bearbeitet werden können.

- Tasks

  eine kleine und unabhängige Arbeitseinheit, concurrently mit anderen tasks ausgeführt, erstellt vom main Programm

- Spawn

  Ein Spawn zeigt an, dass eine Methode als Task ausgeführt werden soll.

- Sync

  Synchronizationspunkt, aufrufende Methode wartet auf alle gespawnted tasks

- Work Stealing

  dynamic load-balancing algorithm, wenn Prozessor idle → versucht den anderen Prozessoren Arbeit wegzunehmen (höhere hardware utilization)

- Lightweight

  optimiert zur Reduktion des Thread creation & task scheduling overheads

Ein task kann Code ausführen, andere tasks spawnen und auf die Ergebnisse anderer tasks warten. Tasks können aber müssen nicht parallel ausgeführt werden, dies hängt vom scheduler ab. Während der Ausführung wird ein task graph (DAG - directed acyclic graph) erstellt, welcher vorgibt in welcher Reihenfolge tasks zu beenden/schedulen sind.

Im task graph sind Gewichte/Kosten vermerkt. Ein fork erzeugt mehrere ausgehende Kanten, während ein join mehrere eingehende Kanten zusammenführt.

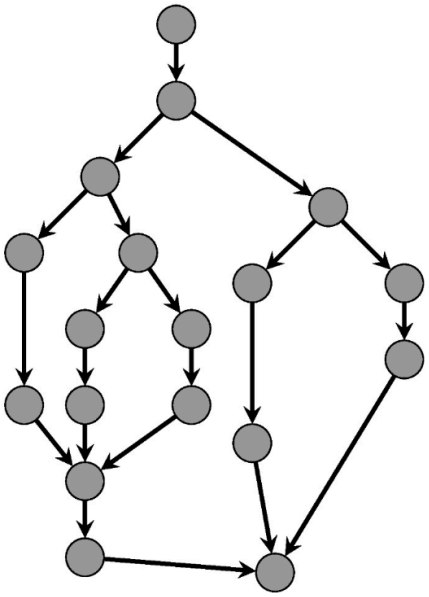


Figure 4: Beispiel task graph

T_1 ist die Summe der Arbeit bzw. Summe der Arbeit aller Knoten. T_∞ ist der critial path/span, der längste Pfad vom Ursprung zur Senke. Dieser entspricht der Ausführungsdauer mit unendlich vielen Prozessoren. Beachte, dass T_p (die Dauer einer konkreten Ausführung) vom scheduler abhängt. Wenn ein graph unterschiedlich gescheduled wird, kann dies zu unterschiedlichen Ausführungen führen.

Intuition sagt uns, dass breitere Graphen besser sind. Häufig ist dies richtig. Denn breitere Graphen sind häufig nicht so tief. Folglich ist der critical path kürzer und mit ausreichend Prozessoren kann einer höhere Performance erreicht werden.

Man kann zeigen, dass $T_p = \frac{T_1}{p} + \mathcal{O}(T_\infty)$ theoretisch. Und durch praktische Analyse des Algorithmus lässt sich $T_p \approx T_1/P + T_\infty$ vermuten.

6.4 Fork/Join Framework

The Fork-Join Framework in Java is based on the idea of Cilk-style parallelism and help us to avoid problems caused with the ExecutorService. Wenn ein task wartet, wird dieser suspended, sodass andere tasks ausgeführt werden können und es nicht zu einem Deadlock kommt. Diese Framework ist eine standard java library und ähnliche Bibliotheken sind auch in anderen Sprachen verfügbar.

Zunächst muss ein ‘ForkJoinPool‘ erstellt werden: ‘ForkJoinPool fjp = new ForkJoinPool();‘. Als Parameter kann die Anzahl der Threads übergeben werden - Standardwert ist die Anzahl der Kerne des Systems.

Thread	‘RecursiveTask⟨V⟩‘ (mit Rückgabewert) oder ‘RecursiveAction‘ (ohne Rückgabewert
‘run()‘	‘compute()‘
kein Rückgabewert	Rückgabewert möglich
‘start()‘	‘fork()‘
‘join()‘	‘join()‘

Die Ausführung eines ForkJoinPools wird mit ‘.invoke()‘ (blockierend und Ergebnis zurückgeben) oder ‘.submit()‘ (nicht blockierend, Task ForkJoinTask zurückgebend) gestartet. An diese Methode wird ein task übergeben.

Zur Optimierung sollte man einen sequentiellen cutoff verwenden und zusätzlich eine der erzeugten tasks mit ‘.compute()‘ im aktuellen Thread aufrufen.

Divide-and-Conquer und insbesondere das Fork/Join Framework können für eine Vielzahl von Anwendungen verwendet werden:

- Maximum/Minimum
- Testen, ob ein Element mit einer bestimmten Eigenschaft existiert
- das linkeste/rechteste Element mit einer bestimmten Eigenschaft finden
- Zählen von Elementen mit einer bestimmten Eigenschaft
- ...

6.4.1 Fork/Join Concepts

Im Fork/Join Framework werden die nächsten tasks in einer Queue gespeichert. Threads nehmen tasks von dieser Queue und fügen sie in einer interne double ended queue (deq) ein, welche sie abarbeiten. Wenn ein Thread keine tasks mehr hat und auch die main queue leer ist, kann er tasks von der queue anderer threads stehlen.

For the runtime we have $T_p = \mathcal{O}(\frac{T_1}{p} + T_\infty)$. The Fork/Join Framework gives an expected-time guarantee with that runtime. But we have no guarantee for an individual execution. To get the best approximation, tasks should have a similar amount of work each.

6.5 data structures

Unterschiedliche Datenstrukturen unterstützen parallele Datenverarbeitung unterschiedlich gut. LinkedLists sind nur begrenzt für parallel programming geeignet. Dies liegt daran, dass immer alle vorherigen Elemente durchlaufen werden müssen (mit Laufzeit $\mathcal{O}(n)$). Dies verlangsamt die parallele Rechenzeit drastisch im Vergleich zu arrays, welche z. B. eine Leseoperation in $\mathcal{O}(1)$ durchführen können.

Nichtsdestotrotz kann sich ein paralleles Programm auch bei LinkedLists ggf. lohnen. Dies ist der Fall, wenn die einzelnen Berechnungen sehr aufwendig sind, sodass sich die Kosten zum Starten eines Threads amortisieren.

Z. B. das Inkrementieren aller Elemente kann mit einer LinkedList parallel und sequentiall nur in $\mathcal{O}(n)$ erledigt werden. Mit einem array könnte ein paralleles Program theoretisch auch in $\mathcal{O}(1)$ auskommen (das Starten der Threads vernachlässigend).

(Balancierte) Trees sind LinkedLists deutlich überlegen. Sie bieten eine ähnliche Flexibilität, während die Laufzeit für einen Zugriff auf $\mathcal{O}(\log n)$ reduziert ist. Beachte jedoch, dass das Balancieren eines Trees in einem parallelen Kontext anspruchsvoll ist - besonders

wenn man Wert auf Performance legt und nicht immer den gesamten tree locken möchte (Locks werden weiter unten eingeführt).

6.6 parallel patterns

Parallel patterns sind allgemeine Ansätze/Muster, die auf eine Vielzahl von Problemen angewendet werden können, um diese mit parallel computation zu lösen. Maps und Reductions sind die mit Abstand häufigsten parallelen patterns, und viele andere patterns werden von diesen allgemeinen patterns abgeleitet.

Häufig werden auch mehrere patterns kombiniert, indem man sie nacheinander anwendet (z. B. erst eine map, dann eine reduction), um ein bestimmtes Problem zu lösen.

Auf großen Clustern/"Supercomputern" ziehen wir oft MapReduce in Betracht, das uns dabei unterstützt, parallelen Code mit maps und reductions zu schreiben. Man muss lediglich angeben, wie man ein einzelnes Element abbildet oder wie man eine Menge von Elementen auf ein neues Element reduziert. Ein System verwaltet dann die Ausführung des MapReduce mit großen Rechenressourcen.

6.6.1 reductions

Reductions nehmen als Eingabe Daten und haben als Ausgabe weniger Daten. Reductions funktionieren, wenn wir eine assoziative Operation auf den Daten durchführen wollen (z.B. max, Zählung, linkstes/rechtestes Element eines Typs, Summe, Produkt, ...).

Eine reduction ruft rekursiv eine Funktion auf einem Teil der Daten auf, bis ein Basisfall erreicht ist. Dann werden die Teilergebnisse nach oben propagiert und schließlich zurückgegeben. Der Rückgabewert eines solchen Funktionsaufrufs kann alles sein: Strings, Integer, Arrays, Objekte, ...

Rekursive Funktionsaufrufe können parallelisiert werden, wie es zuvor mit Divide-and-Conquer/Cilk-style parallelism eingeführt wurde.

Ein Beispiel wäre das Berechnen der Summe. Oder den Max/Min-Wert zu ermitteln. Oder die Anzahl der Werte über und unter einem Schwellenwert zu ermitteln (z. B. das Zählen schwarzer und weißer Pixel in einem Bild).

6.6.2 maps

Eine Abbildung nimmt als Eingabe Daten einer bestimmten Länge (z. B. ein array der Länge n) und bearbeitet jedes Element (jeden Eintrag im array) unabhängig. Die Ausgabe hat dann die gleiche Länge. Die Map führt für jeden Eintrag die gleiche Operation aus bzw. ruft für jeden Eintrag die gleiche Funktion auf. Die einzelnen Ergebnisse werden nicht kombiniert.

Jede Verarbeitung eines einzelnen Elements kann unabhängig erfolgen, d.h. in einem separaten Thread, um eine parallele Berechnung zu ermöglichen.

Ein einfaches Beispiel ist die Vektoraddition.

6.6.3 stencil

Stencils können als eine Verallgemeinerung von Maps betrachtet werden. Anstatt ein Eingabeelement zu einem Ausgabeelement zu verarbeiten, kann ein Ausgabeelement von mehreren (benachbarten) Eingabeelementen abhängen.

Für jede Ausgabe führen wir eine Berechnung durch. Die Eingabe für diese Berechnung wird Schablone oder Kernel genannt. Jede dieser Berechnungen (um eine Ausgabe zu

berechnen) kann unabhängig (in einem separaten Thread) durchgeführt werden, um eine parallele Berechnung zu ermöglichen.

Einige Anwendungsbeispiele sind: smoothing/average filter, median filter, convolutions (z. B. im Zusammenhang mit convolutional neural networks).

6.7 algorithms

Beim Entwurf parallelel Algorithmen ist das grundlegende Ziel, den span/critial path der Ausführung zu verringern. So kann das Programm auf vielen Kernen effizienter ausgeführt werden. Häufig ist dies nur möglich, indem die Arbeit vergrößert wird, z. B., wegen zusätzlichem Kommunikationsaufwand. Diese Arbeitsvergrößerung möchte man bei der Minimierung des spans so gering wie möglich halten, um die Effizienz nicht zu stark zu reduzieren.

Beim Implementieren von parallelen Algorithmen/Programmen greift man meist auf Framework zurück, welche einen Teil der Parallelisierung übernehmen.

- Arbeit den einzelnen Prozessoren/Kernen zuweisen. Dabei sollte vermieden werden, dass Kerne idle laufen/Ressourcen verschwendet werden.
- Konstante Laufzeitfaktoren sollten möglichst klein gehalten werden. Denn parallele Programme können rotz theoretisch guter Laufzeiten, wegen großer Konstanten in der Praxis häufig nur begrenzt verwendet werden.
- Angaben bzgl. der erwarteten Laufzeit bei korrekter Verwendung der Bibliothek durch den Nutzer geben.

Als Nutzer von Frameworks hat man die Verantwortung, diese korrekt zu nutzen.

- Algorithmen entwerden/nutzen, welche konzeptuell gut parallelisiert werden können.
- Alle Arbeitspakete (je Thread) sollten ähnlich groß sein und vergleichsweise klein sein, um gute Skalierbarkeit auf high-performance Systemen zu ermöglichen.

6.7.1 analyzing algorithms

Genau wie für sequentielle Programme, müssen Programmkorrektheit und Programmlaufzeit betrachtet werden. Ersteres ist für die einfachen Algorithmen, welche wir heir betrachten, trivial. Bzgl. Letzerem möchten wir Grenzen für die asymptotische Laufzeit bei gegebener Anzahl von Kernen/parallelen Ausführungsströmen finden.

6.7.2 prefix-sum

Dieser Algorithmus ist ein Beispiel, dass selbst Probleme, welche inherent sequentiell scheinen, durch clevere Ideen parallelisiert werden können.

```
1 int[] prefix_sum(int[] input){
2   int[] output = new int[input.length];
3   output[0] = input[0];
4   for(int i=1; i < input.length; i++)
5     output[i] = output[i-1]+input[i];
6   return output;
7 }
```

Nun betrachten wir folgenden parallelen Algorithmus, welcher in zwei Phasen (+ prep Phase) funktioniert:

1. Prep-Phase (Erstellung eines Binary Trees)

   Hat das Array range $[a, b[$ wird eine Rekursion bis zum Base Case $[x, x+1[$ durchgeführt.

2. Berechnung der Summen

   Es wird die Summe der Elemente der jeweiligen children eines Rekursionsschrittes berechnet (durch Propagierung nach oben).

3. 'fromLeft' Berechnung

   Die root erhält Wert 0. Das linke Kind erhält jeweils den gleichen Wert wie der parent. Das rechte Kind erhält jeweils den Wert 'parent + Summe des left child'.

Das Ergebnis resultiert dann für jedes Element aus der Addition von 'fromLeft' zum eigenen Wert: 'output[i] = fromLeft + input[i]'.

Dieser Ansatz reduziert die Laufzeit erheblich. Während die Arbeit weiterhin bei $\mathcal{O}(n)$ liegt, konnte der span auf $\mathcal{O}(\log n)$ reduziert werden.

Dies kann als allgemeines Muster für mehreren Probleme verstanden werden:

- min/max aller Elemente links/rechts vom i-ten Element.
- alle Elements links/rechts eines i-ten Elementes betrachten

6.7.3 pack

'Pack' ist keine standard Begriff, ein solcher existiert für diesen Ansatz nicht. Die Aufgabe besteht darin, ein output array von einem input array zu erstellen, sodass der output nur elemente des inputs mit einer Bestimmen Eigenschaft enthält. Z. B.: Der Output soll nur Elemente > 10 enthalten.

Konzeptuell können wir drei Phasen betrachten.

1. Parallel map: Berechne bit Vektor für 'valide' Elemente, i.e., $1 \Leftrightarrow$ Element behalten ('bits')
2. Parallel-prefix sum auf dem bit-vector ('bitsum')
3. Parallel map zum output (der Größe vom längsten prefix-sum Element) ('output')

```
1  output = array of size bitsum[n-1]
2  FOR (i=0; i < input.length; i++):
3    IF bits[i] == 1:
4      output[bitsum[i]-1] = input[i]
```

In der Praxis können wir eine weitere Optimierung erhalten, indem wir Phasen 1 und 3 in Phase 2 integrieren. Dies resultiert jedoch in keine Verbesserung der asymptotischen Laufzeit.

Für n threads ist die Laufzeit der maps $\mathcal{O}(1)$ und der prefix-sum $\mathcal{O}(\log n)$. Die totale Laufzeit ist somit $\mathcal{O}(\log n)$, während die Arbeit $\mathcal{O}(n)$ bleibt.

6.7.4 Quicksort

Erinnere, dass Quicksort eine erwartete Laufzeit von $\mathcal{O}(n \log n)$ hat. Die grundlegende Funktionsweise des Algorithmus:

1. Ein pivot Element auswählen. - $\mathcal{O}(1)$
2. Partition der Daten in Elemente kleiner als das Pivot, das Pivot, Elemente größer als das Pivot. - $\mathcal{O}(n)$
3. Elemente links und rechts vom pivot rekursiv sortieren. - $2 \cdot \mathcal{O}(\frac{n}{2})$ (randomisiertes Pivot)

Wenn man die Rekursion parallelisiert, erhält man:

$$T(n) = \mathcal{O}(n) + T(\frac{n}{2}) = \Theta(n) \qquad \text{using the Master Theorem}$$

Man kann eine zusätzliche Optimierung der asymptotischen Laufzeit durchführen, indem man die Partition parallelisiert. Dies hat allerdings nur begrenzten praktischen Mehrwert da, da der Algorithmus dann nicht mehr 'in-place' ist und man hohe Konstanten einführt. Allerdings kann man alle kleineren und größeren Elemente als Pack berechnen (wie oben), sodass die Laufzeit für die Partition $2 \cdot \mathcal{O}(\log n) \leq \mathcal{O}(\log n)$ ist. Man erhält mit dem Master Theorem:

$$T(n) = \mathcal{O}(\log n) + T(\frac{n}{2}) = \Theta(\log^2)$$

7 Memory Models & Memory Reordering

Als Programmierer/Entwickler einer Programmiersprache muss klar sein, welche Garantieren wir bzgl. des Speichers annehmen können, um korrekte Programme schreiben zu können. Die Memory Models von Programmiersprachen bieten entsprechend (meist minimale) Garantien bzgl. des Effektes von Speicheroperationen. Solche Garantien der Programmiersprache, welche durch den Compiler umgesetzt werden müssen, sind notwendig, da das Verhalten je nach Plattform, Laufzeitumgebung, ... unterschiedlich sein kann.

Hardware-Entwickler haben sich beim Design der CPU/ISA für ein Memory Model entschieden. Und Compiler-Entwickler haben eine Abstraktion von diesem Modell in der Programmiersprache zur Verfügung gestellt. Somit hat man als Programmierer nur einen relativ kleinen Freiraum bzgl. Speicherverwaltung.

7.1 Memory Reordering

Da Recheneinheiten ursprünglich ausschließlich sequentiell gearbeitet haben, führen heutige Compiler Optimierungen durch, welche bei sequentieller Ausführung nicht auffallen aber bei paralleler Ausführung jedoch zu Problemen führen können. Solche Optimierungen werden nicht abgeschafft, da dies zu ca. 30 % Performanceverlusten führt.

Solche Optimierungen beinhalten dead code elimination, register hoisting, locality optimizations, pre-evaluation, ... Da Speicherzugriffe über 100x langsamer als einfache arithmetische Operationen sein könne, können Compiler z. B. ‘x = 1; y = x+1;‘ durch ‘x = 1; y = 2;‘ ersetzen. Zudem ändern Compiler typischerweise nicht einfach die Reihenfolge von Instruktionen. Allerdings führen moderne CPUs häufig Code Out-of-Order in einer entsprechenden Pipeline aus.

In kurz:

- Der Compiler und die Hardware (uarch) können Optimierungen/Änderungen durchführen, welche die Semantik der sequentiellen Ausführung nicht beeinflussen.
- Moderne Compiler geben keine Garantie bzgl. der globalen Ordnung von Speicherzugriffen.
- Moderne Multiprozessoren geben keine Garantie bzgl. der Ausführungsreihenfolge von Instruktionen.

7.2 Architectural Memory Models

Ein Grund für die Umordnung von Speicherzugriffen liegt bei der Mikro-Architektur. Unterschiedliche ISAs erlauben unterschiedliche Umordnungen. Keine dieser Umordnungen verursache Probleme bei sequentiellen Ausführungen - bei parallelen Ausführungen ist dies jedoch anders.

Memory ordering in some architectures[7][8]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC			x86	AMD64	IA-64	z/Architecture
					RMO	PSO	TSO	oostore[a]			
Loads reordered after loads	Y	Y	Y	Y	Y			Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y		Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y					Y	
Atomic reordered with stores	Y	Y		Y	Y	Y				Y	
Dependent loads reordered	Y										
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y		Y	

Figure 5: Garantien verschiedener ISAs hinsichtlich Umordnungen

7.3 Java Memory Model (JMM)

Als durchschnittlicher Programmierer setzt man sich selten mit dem Speichermodell der ISAs auseinander. Stattdessen sind die Memory Models der Programmiersprachen relevant - wir betrachten hier Java. Der Compiler muss die Garantien des Speichermodells der Programmiersprache mit dem Speichermodell der jeweiligen ISA umsetzen.

Das JMM gibt Garantien für das Verhalten von Programmen, wenn bestimmte Operatoren wie ‘volatile‘ und ‘synchronized‘ verwendet werden. Für die weitere Analyse betrachtet das JMM actions, wie ‘read(x):1‘ (lese von x den Wert 1), welche während Ausführungen betrachtet werden. Beim Schreiben eines Programms müssen wir erlaubte Ordnungen der actions des Programms betrachten, um dessen Korrektheit zu evaluieren.

- Program Order (PO)
- Synchronization Order (SO)
- Synchronizes With Order (SW) und Happens Before Order (HB)

7.3.1 Program Order (PO)

Die Program Order betrachtet die Ausführungen einzelner Threads. Die PO garantiert intra-thread consistency. Dies kann als typische sequentielle Ausführungsreihenfolge der actions verstanden werden. Die PO ist dann die Vereinigung der Ordnungen der einzelnen Threads.

Eine PO ist valide, wenn diese als Ausführungsreihenfolge so geschehen kann.

7.3.2 Synchronization Order (SO) & Synchronizes With Order (SW)

Die SO ist eine totale Ordnung von synchronization actions. Synchronization actions beinhalten

- read/write von volatile Variablen
- lock/relese monitors
- first/last action eines thread
- actions, welche einen thread starten
- actions, welche bestimmen ob ein thread terminierte

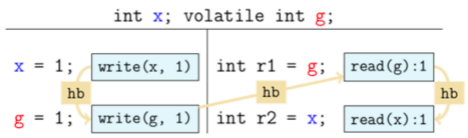
Diese Ordnung ist global für alle Threads und ist jeweils mit der PO vereinbar. Zudem ist die SO consistent. Dies bedeutet, dass alle reads in SO jeweils das letzte write in SO order sehen.

Die Synchronizes With Order (SW) ist eine Teilmenge der SO. Es werden nur solche Relationen zwischen actions betrachtet, einem release-acquire pair entsprechen. a steht in Relation zu b, wenn a ist ein unlock und b ein lock des selben Monitors oder a ist ein volatile write und b ein volatile read der selben volatile Variablen.

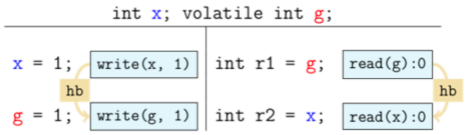
7.3.3 Happens Before Order (HB)

Die HB Order ist die Transitive Hülle der SW und PO, welche transitiv ist. Zudem gilt HB consistency: Wenn eine Variable gelesen wird, sieht man entweder die letzte Schreiboperation (in HB) oder eine ungeordnete Schreiboperation. Races sind entsprechend möglich, wenn es keine Synchronisierung zwischen Threads gibt.

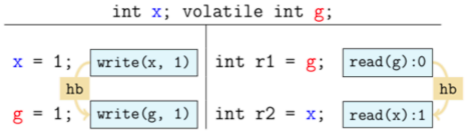
7.3.4 Beispiele



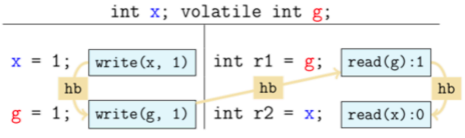
Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$
$(r1, r2) = (1, 1)$



Case 2: HB consistent, observe the default value
$(r1, r2) = (0, 0)$



Case 3: HB consistent (!), reading via race!
$(r1, r2) = (0, 1)$



Case 4: HB **in**consistent, execution can be thrown away

Figure 6

Als Teil vom JMM ist definiert, dass Synchronisierungsoperationen nicht umgeordnet werden dürfen. Dies bedeutet, dass 'synchronized' Blöcke so wie spezifiziert nacheinander ausgeführt werden und es nicht durch Optimierungen zu einer geänderten Ausführungsreihenfolge kommen darf.

Beachte, dass verschiedene Threads ggf. in unterschiedliche Caches schreiben und Änderungen eines Threads somit nicht zwingend für andere Threads sichtbar sein. Um die unmittelbare Sichtbarkeit von Operationen zu erzwingen ist das 'volatile' keyword bei der Initialisierung von Variablen notwendig. Beachte allerdings dass volatile Variablen langsamer als normale Variablen sind, jedoch performanter als Locks.

8 (In)Korrektheitsbeweise

Selbstverständlich muss die Korrektheit von Algorithmen bewiesen werden. Für parallele Programme gibt es dafür unterschiedliche Ansätze:

- Proof by exhaustion/enumeration.

  Hier werden alle möglichen interleavings/Ausführungsreihenfolgen betrachtet, um zu zeigen, dass alle valide sind.

  In der Praxis ist dieser Ansatz häufig nicht relevant, da es sehr viele mögliche interleavings gibt. Mit n threads und jeweils k Statements existieren $\binom{nk}{n}\binom{(n-1)k}{k}\cdots\binom{2k}{k}$ interleavings. Für 2 Threads gilt $\leq \mathcal{O}(\frac{4^k}{\sqrt{2k}})$.

- Proof by contradiction.

  Hier wird angenommen, dass die Korrektheit nicht gilt und gezeigt, dass dies bei gegebenen Programm nicht möglich ist, indem das Programm rückwärts durchlaufen wird

- State-Space-Diagram

  In State Space Diagrammen bildet man alle (relevanten, um den state space zu verringern) Zustände des Programms (bestehend aus Variablenwerten, Ausfühungspunkte der Threads) als Nodes eines Graphen ab. Die Nodes sind gerichtet verbunden, wenn ein entsprechender state Wechsel möglich ist.

  So kann z. B. mutual exclusion widerlegt werden, indem gezeigt wird, dass ein State mit zwei critical section Ausführungen möglich ist. Zudem kann ein Deadlock als ein State ohne ausgehende Kanten und ein Livelock als loop ohne progress dargestellt werden.

8.1 Beispiel

```
1  class C {
2    private int x = 0;
3    private int y = 0;
4
5    void f() {
6      x = 1; // A
7      y = 1; // B
8    }
9
10   void g() {
```

```
11      int a = y; // C
12      int b = x; // D
13      assert (b >= a);
14    }
15  }
```

Wenn wir bei diesem C-Programm aggressive Optimierungen des Compilers anschalten, schlägt dieses Programm nach unbestimmter Laufzeit jedoch auch fehl, obwohl wir beide Beweismethoden erfolgreich anwenden können, wenn wir annehmen, dass Instruktionen innerhalb eines Threads geordnet ausgeführt werden. Dies liegt an memory reordering, eine Optimierung des Compilers, welche die sequentielle Ausführung innerhalb eines Threads so invalidiert, dass es bei einer rein sequentiellen Ausführung nicht ersichtlich ist.

9 Mutual Exclusion with Locks

Locks sind eine Form der Synchronization von Threads/Prozessen. Um bad interleavings zu vermeiden und Programmkorrektheit zu garantieren, können Locks eingesetzt werden. Locks stellen sicher, dass jeweils nur ein Thread Zugriff auf eine Ressource hat bzw. immer nur ein Thread einen bestimmten Codeabschnitt, welcher mit dieser Ressource verknüpft ist, ausführen kann. Dies implementiert das Konzept von mutual exclusion (gegenseitigem Ausschluss). Zusätzlich haben Locks (in Java) auch eine synchronisierende Eigenschaft. Das bedeutet, dass nach dem Freigeben eines Locks, die ggf. modifizierten Daten für alle Threads sichtbar sind. Dies entspricht data consistency.

Locks müssen von der Programmiersprache/dem System zur Verfügung gestellt werden. Ein fundamentales Lock unterstützt folgende Methoden, welche atomic sein müssen:

- ‘new‘ - Erstellen eines neuen Locks, ’not held’
- ‘acquire‘ - blockiert, wenn das LocK ‘held‘ ist & setzt lock auf ‘held‘, wenn ‘not held‘ wird und blockiert nicht mehr (immer nur ein Thread)
- ‘release‘ - setzt lock auf ‘not held‘, wenn ‘held‘ und der Thread das lock besessen hat

Folglich kann immer nur ein Thread das lock halten. Das lock zu halten ist Voraussetzung um die critical section, der vom Lock geschützte Code Block, auszuführen. Dadurch wird der Zugriff die Daten, welche in der critical section verarbeitet werden, kontrolliert.

Critical Sections sind Codeblöcke mit bestimmten Eigenschaften

1. Mutual exclusion: Instruktionen der critical section mehreren Prozesse dürfen nicht interleaved werden.
2. Freedom from deadlock: Wenn mehrere Prozesse eine kritische Sektion betreten möchte/ein lock akquirieren möchte, muss einer von diesen irgendwann Erfolg haben.
3. Freedom from starvation: Wenn ein Prozess eine kritische Sektion betreten möchte/ein lock akquirieren möchte, muss dies irgendwann gelingen.

Beim Arbeiten mit Locks müssen einige Aspekte beachtet werden:

- Locks müssen wieder freigegeben werden (selbst bei Exceptions).
- Das selbe Lock muss für alle critical Sections verwendet werden, die eine Datenstruktur schützen.

Locks können nur ein mal ‘acquired‘ werden. Somit kann in einer critical section eigentlich keine Methode aufgerufen werden, welche das gleiche lock verwendet. Um dies jedoch zu ermöglichen existieren reentrant locks, welche die Anzahl der ‘acquire‘ minus Anzahl der ‘release‘ eines Threads zählen.

Ein Problem von Locks ist, dass es nicht direkt klar ist, welches Lock zu welcher Datenstruktur gehört. Um solche Probleme anzugehen ist eine ausführliche Dokumentation der verwendeten Locks sowie die Annotation von Datenstrukturen mit den für die Bearbeitung benötigten Locks notwendig.

Nun betrachten wir wir Locks implementiert werden können. Dazu machen wir einige Annahmen:

- atomic reads und writes
- keine read-write Umordnungen (nicht notwendig, nur der Einfachheit halber)
- Threads verlassen critical sections irgendwann
- weitere implizite Annahmen (keine sudden thread deaths, ...)

Für die Entwicklung von Locks müssen wir n Prozesse betrachten, welche jeweils wie folgt aufgebaut sind:

```
1 local variables
2 loop
3   non-critical section
4   preprotocol
5   critical section
6   postprotocol
```

Zusätzlich betrachten wir bei locks meistens fairness. Dies bedeutet in etwa, dass nicht ein thread wiederholt das lock erhält während andere threads verhungern oder sehr lange/länger warten müssen. Formal teilen wir das protocol in zwei Schritte: Doorway interval D (endliche viele Schritte) und waiting interval W (unbegrenzt viele Schritte). Ein lock ist first-come-first-served, i.e., fair, wenn $D_A \rightarrow D_B \Rightarrow CS_A \rightarrow CS_B$ gilt.

9.1 Single Core Systems

Single-core locks sind einfach zu implementieren. Denn das wechseln des Threads auf einem Core benötigt interrupts. Wenn man also im preprotocol IRQs (interrupt requests) abschaltet und im postprotocol wieder anschaltet, wird die critical section garantiert ohne Unterbrechungen ausgeführt.

9.2 2 Core Systems

Beachte, dass es viele triviale falsche Algorithmen gibt. So kann man z. B. nicht die Intention eines Threads zu locken als (volatile) Variable speichern. Solche Algorithmen können meist gut mit State Space Diagrammen als falsch identifiziert werden.

9.2.1 Decker's Algorithm

Wir betrachten Prozesse P und Q mit ‘volatile boolean wantp=false, wantq=false, integer turn=1‘.

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

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

9.2.2 Peterson's Algorithm, fair

Wir betrachten Prozesse P/1 und Q/2 mit 'volatile boolean array flag[1..2]=[false, false]' und 'volatile integer victim=1'.

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

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

```
19    flag[Q]=false
```

An dieser Stelle beweisen wir die Korrektheit des Peterson Locks. Dazu müssen wir uns an atomic registers erinnern und führen den Begriff/die Notation von Intervallen ein. Ein Thread P erzeugt events $p_0, p_1, ...$, welche statements entsprechen. Das j-te Auftreten des i-ten Events wird als p_i^j bezeichnet. Wir schreiben $a \to b$, falls Event a vor Event b auftritt (total, wenn intra-tread). (a_0, a_1) ist das Interval der Events x mit $a_0 \to x \to a_1$. Wir schreiben $I_A \to I_B$ mit $I_A = (a_0, a_1), I_B = (b_0, b_1)$ falls $a_1 \to b_0$.

Beweis mutual exclusion (by contradiction) Es gelten diese intra-thread orders:

- $W_P(flag[P] = true) \to W_P(victim = P) \to R_P(\neg flag[Q]) \vee R_P(victimQ) \to CS_P$
- $W_Q(flag[Q] = true) \to W_Q(victim = Q) \to R_Q(\neg flag[P]) \vee R_Q(victimP) \to CS_Q$

Ohne Verlust der Allgemeinheit $W_Q(victim = Q) \to W_P(victim = P)$. Somit muss $R_P(victim)$ P lesen. Um in die critical section zu kommen müssen wir somit $R_P(\neg flag[Q])$ als wahr lesen. Aufgrund von Transitivität von $\to$ ist dies jedoch unmöglich.

Wir müssen nicht betrachten, dass einer/beide Threads bereits die critical section durchlaufen sind, da nach einem einzigen Durchlauf die Situation wieder als Ausgangslage dargestellt werden kann. Zudem müssen wir nicht betrachten, dass beide gleichzeitig in der critical section waren, da dies durch obige Analyse und Überlegung unmöglich ist. (Würden wir es annehmen, würde der Beweis nicht funktionieren.)

Beweis von freedom of starvation (by exhaustive contradiction) P würde verhungern (endlos while Schleife), wenn $flag[Q] == true$ und $victim == P$ für immer gilt. Wir zeigen, dass dies für alle möglichen Zustände von Q nicht möglich ist.

- Q hängt in der non-critical section fest.

  Dann wäre $flag[Q] == flase$ (entweder initial oder durch Verlassen der critical section)

- Q betritt und verlässt die critical section wiederholt

  Beim betreten setzt Q $victim == Q$. Dies kann nicht geändert werden, sodass P fortfahren kann.

- Q steckt auch im pre-protocol (der while Schleife) fest

  Diest ist unmöglich da $Q == victim == P$ gelten müsste.

- Q steckt in der critical section fest

  Dies hatten wir zu Beginn per Definition ausgeschlossen.

Java Implementation Dies ist eine illustrative Java Implementation, welche aus verschiedenen Gründen praktisch nicht funktioniert.

```
1  class PetersonLock {
2    // volatile boolean flag[] = new boolean[2];
3    // This line does not work, because only the reference
4    // to the array is volatile/atomic but not the array
5    // elements themselves. Hence, this is completely useless.
6    // Instead, we need to use AtomicIntegerArray.
7    AtomicIntegerArray aia = new AtomicIntegerArray(2);
```

```
8    // Of course, also the below usage of flag[] would
9    // also have to be updated.
10   volatile int victim;
11
12   public void Acquire(ind id) {
13     flag[id] = true;
14     victim = id;
15     while (flag[1-id] && victim == id);
16   }
17
18   public void Release(int id) {
19     flag[id] = false;
20   }
21 }
```

9.3 Multicore Systems

9.3.1 Filter Lock

Das Filter Lock ist starvation- und deadlock-free sowie garantiert mutual exclusion. Allerdings ist es nicht fair, insbesondere ist es nicht first-come-first-served. Zudem hat dieses Lock Laufzeit $\mathcal{O}(n^2)$ mit Speicher $\mathcal{O}(n)$ (sehr hohe Ressourcennutzung).

Das Filter Lock ist eine Generalisierung des Peterson Locks für n Prozesse. Jedem thread t ist ein level $level[t]$ zugeordnet. Um die CS zu betreten, muss ein thread alle Level passieren. Für jedes Level l gibt es ein victim $victim[l]$, welches bei einem Konflikt andere Threads passieren lassen muss.

```
1  int[] level(#threads)
2  int[] victim(#threads)
3
4  lock(me) {
5    for (int i=1; i<n; ++i) {
6      level[me] = i;
7      victim[i] = me;
8      while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
9    }
10   // level[me] = n; // siehe Anmerkung unten
11 }
12
13 unlock(me) {
14   level[me] = 0;
15 }
```

Alternativ ist es auch möglich den Check der nächsten Level auf das aktuelle und das nächste Level zu beschränken. Dadurch kann die Performance weiter erhöht werden, besonders bei geringer contention, vielen Threads/hohem n und kurzen critical sections. Eine minimale weitere Verbesserung erreicht man dann durch Nutzung der einen auskommentierten Zeile, da so auch das vorletzte Level von 'anstehenden' Threads effizienter genutzt wird.

Dies ist eine Implementation in Java:

```
1  import java.util.concurrent.atomic.AtomicIntegerArray;
2
3  class FilterLock {
4    AtomicIntegerArray level;
```

```
5   AtomicIntegerArray victim;
6   volatile int n;
7
8   FilterLock(int n) {
9     this.n = n;
10    level = new AtomicIntegerArray(n);
11    victim = new AtomicIntegerArray(n);
12  }
13
14  //∃k ≠ me: level[k] >= i (lev)
15  boolean others(int me, int lev) {
16    for (int k = 0; k < n; ++k)
17      if (k != me && level.get(k) >= lev)
18        return true;
19    return false;
20  }
21
22  public void Acquire(int me) {
23    for (int lev = 1; lev < n; ++lev) {
24      level.set(me, lev);
25      victim.set(lev, me);
26      while(me == victim.get(lev) && others(me,lev));
27    }
28  }
29
30  public void Release(int me) {
31    level.set(me, 0);
32  }
33 }
```

9.3.2 Bakery Algorithm, fair

Dieser Algorithmus basiert auf einer allgemein bekannten Idee. Wenn ein thread das lock akquirieren möchte, muss er ein Ticket/eine Zahl ziehen, welcher größer als alle ausstehenden Tickers ist. Sobald dieses Ticket die kleinste Zahl hat, darf der Thread in die critical section.

```
1  integer array[0..n-1] label = [0, ..., 0];
2  boolean array[0..n-1] flag = [false, ..., false];
3
4  lock(me):
5    flag[me] = true;
6    label[me] = max(label[0],...,label[n-1]) + 1;
7    while (∃k≠me: flag[k] && (k,label[k]) <_l (me,label[me]));
8    // <_l REFERS TO THE LEXICOGRAPHIC ORDERING WITH label
9    // BEING CONSIDERED FIRST THIS IS TO PROVIDE THE GLOBAL
10   // ORDERING IN CASE OF UNFORNUATE SIMULTANEOUS ENTRY
11 unlock(me):
12   flag[me] = false;
```

When we are the only thread trying to enter the critical section/trying to lock, this has runtime $\mathcal{O}(n)$. Memory requirements are still $\mathcal{O}(n)$. Anyway, we won't improve this any more according to a theorem from "Bounds on Shared Memory for Mutual Exclusion", 1993): If S is an atomic read/write system with at least two processes and S must solve

mutual exclusion with global progress (deadlock-freedom), then S must have at least as many variables as processes.

Dies ist eine Implementation in Java:

```
1  class BakeryLock {
2    AtomicIntegerArray flag;
3    AtomicIntegerArray label;
4    final int n;
5
6    BakeryLock(int n) {
7      this.n = n;
8      flag = new AtomicIntegerArray(n);
9      label = new AtomicIntegerArray(n);
10   }
11
12   int MaxLabel() {
13     int max = label.get(0);
14     for (int i = 0; i<n; i++) {
15       max = Math.max(max, label.get(i));
16     }
17     return max;
18   }
19
20   boolean Conflict(int me) {
21     for (int i = 0; i<n; i++) {
22       if(i != me && flag.get(i) != 0) {
23         int diff = label.get(i) - label.get(me);
24         if( diff<0 || (diff == 0 && i < me)) {
25           return true;
26         }
27       }
28     }
29     return false;
30   }
31
32   void Acquire(int me) {
33     flag.set(me, 1);
34     label.set(me, MaxLabel() + 1);
35     while(Conflict(me));
36   }
37
38   public void Release(int me) {
39     flag.set(me, 0);
40   }
41 }
```

9.4 Remark on Real-World Implementations

It has been said that one can theoretically show the lower bound $\mathcal{O}(n)$ for memory. For this reason and that we have to use volatile/atomic registers for such implementations, such approaches work but are not used in practice. Another reason is that those algorithms are not wait-free (discussed later). Instead we use extended hardware support for parallelism.

9.5 Spinlock

Durch die Verwendung von den atomic operations TAS und CAS können wir nun das spinlock implementieren. Beachte, dass diese Implementationen deadlock-free sind und mutual exclusion garantieren, jedoch starvation nicht verhindern und keine Fairness garantieren.

```
1  TAS
2    Init(lock):
3      lock = 0;
4    Acquire(lock):
5      while (!TAS(lock));
6    Release(lock):
7      lock = 0;
8
9  CAS
10   Init(lock):
11     lock = 0;
12   Acquire(lock):
13     while (CAS(lock,0,1)!=0);
14   Release(lock):
15     CAS(lock,1,0);
```

9.6 Probleme mit Locks (Deadlocks, Livelocks, Starvation)

Deadlock Ein Deadlock ist eine Situation in einem Mehrprozesssystem, bei dem zwei oder mehr Prozesse unendlich lange auf Ressourcen warten, die von den anderen Prozessen gehalten werden. Jeder Prozess in der Deadlock-Situation hat eine Ressource, die der andere Prozess benötigt, und sie warten aufeinander, um fortzufahren. Weil keiner der Prozesse die benötigte Ressource freigeben kann oder will, sind sie in einem Zustand der ewigen Blockade gefangen.

Formall können wir Graphentheorie benutzen: Ein Deadlock der Threads $T_1, ..., T_n$ existiert, wenn der gerichtete Graph, welcher die Relation zwischen $T_1, ..., T_n$ und $R_1, ..., R_m$ (Ressourcen) beschreibt einen Zyklus enthält.

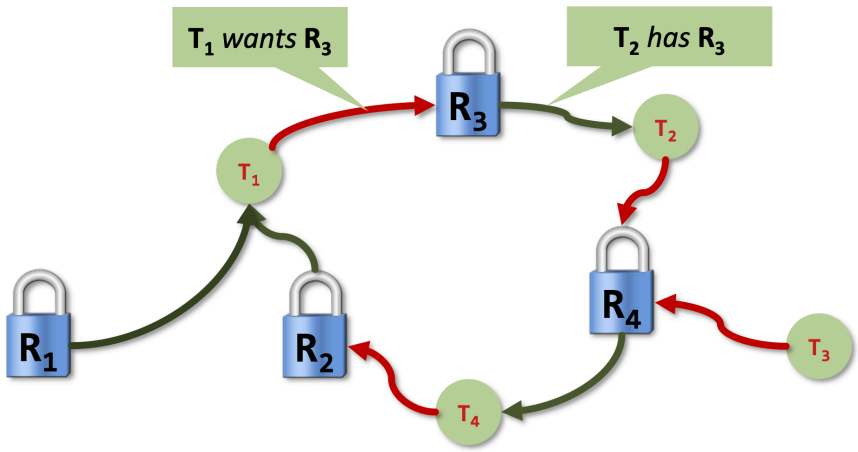


Figure 7: Deadlock durch Graphentheorie, Kante *lock* → *ressource* Intention das Lock zu akquirieren, *ressource* → *lock* Ressource wird vom Lock gehalten

Allgemein gilt, dass Deadlocks nicht aufgelöst werden können. Denn Auflösen führt

im Allgemeinen zu inkonsistenten States. Entsprechend sollte Deadlocks erkannt werden, da dann das Programm terminiert werden kann. Diese deadlock detection geschieht mit dargestellten Graphen.

Zuvor sollten jedoch Deadlocks grundsätzlich vermieden werden. Bei einem Lock ist dies keine Herausforderung. Sobald man mehrere Locks hat, nutzt man...

- two-phase locking with retry

  Wenn man ein Lock hält, schreibt man Daten nicht direkt auf die Ressourcen sondern in einen Buffer. Erst wenn das Lock freigegeben wird, überträgt man die Daten vom Buffer. Wenn man in der critical section einen Deadlock detektiert/die Ausführung nicht weiterlaufen kann, ist der Buffer zu verwerfen und die Ausführung erneut zu versuchen. Dies wird häufig in Datenbanken verwendet.

- resource ordering (populärste Methode)

  Hier wird eine globale Ordnung auf allen Locks definiert. Sobald man lockt, müssen alle verwendeten locks in der vorgegebenen Ordnung gelockt werden. So werden deadlocks vermieden, da es nie passieren kann, dass es sich überkreuzende Abhängigkeiten gibt. Dies wird typischerweise verwendet, wenn ein globaler State aktualisiert wird.

  Um eine globale Ordnung zu erhalten gibt es verschiedene Ansätze:

  - Einen Index auf Basis eines atomaren Counters zu jeder Instanz hinzufügen.
  - Wenn jeweils Locks von zwei Typen gelockt werden, kann eine statische Reihenfolge dieser Typen definiert werden.
  - Sind die Locks/Objekte in einer azyklischen Datenstruktur geordnet, ist dadurch bereits eine Ordnung vorgegeben.
  - Zudem kann man die ID/einen Identifier des Objektes verwenden, welcher z. B. durch die Programmiersprache zur Verfügung gestellt wird.
  - In C und ähnlichen Sprachen kann man auch die Speicher-Adresse eines Objektes nutzen.

Natürlich gibt es noch weitere Möglichkeiten. Diese sind häufig jedoch nicht in der Praxis verwendbar. Z. B. das Verwenden kleinerer und sich nicht überlappender critical section entspricht häufig nicht der Semantik des Programms. Oder das Verwenden eines globalen Locks an Stelle vieler einzelnen Locks macht das Programm effektiv sequentiell und reduziert die Performance erheblich.

Ein Biepsiel, in welchem selbst die Verwendung eines Indexes erhebliche Nachteile bieten würde ist der ‘StringBuffer‘ in Java. Die Entwickler haben sich entschieden die Möglichkeit eines Deadlocks und ungewünschten Verhaltens aus Performance-Gründen in Kauf zu nehmen und diese Problematik in der Dokumentation zu thematisieren. Dann ist es Verantwortung des API-Nutzers die Klasse korrekt zu verwenden.

Deadlock-freedom bedeutet: ’Eventually something good happens.’

Livelock Ein Livelock ist ein Spezialfall eines Deadlocks, bei dem zwei oder mehr Prozesse ständig ihren Zustand ändern, um einen Deadlock zu vermeiden, aber keiner von ihnen kann weitermachen. Hier sind die Prozesse also nicht wirklich ”blockiert” (sie sind in der Lage, ihren Zustand zu ändern), sie kommen nur nicht voran, weil jeder Prozess versucht, auf den anderen zu reagieren.

Livelock-freedom bedeutet: ’Eventually something good happens.’

Starvation Starvation bezeichnet eine Situation in einem System, in der ein Prozess unendlich lange auf die Erfüllung seiner Anforderungen (z.B. Zugriff auf eine Ressource oder Ausführung) warten muss, weil andere Prozesse immer wieder bevorzugt werden. In unserem Fall wartet ein Thread unendlich lange um ein lock zu erhalten. Die "verhungernden" Prozesse kommen in solch einem System nicht voran, da sie ständig übergangen werden.

Starvation-freedom bedeutet: 'Nothing ad ever happens.'

9.7 Locks in Java

On Java gibt es verschiedene Möglichkeiten Locks zu nutzen. Die einfachste (und häufig performanteste) Möglichkeit sind die intrinsischen Locks/Monitore. Zusätzlich gibt es auch externe Locks. Beide unterstützen Condition Variables.

9.7.1 Monitors/interne Locks

Diese Locks werden mit dem 'synchronized' Keyword verwendet. Mit diesem Keyword können Methoden annotiert werden. Eine annotierte Methode nimmt zum Beginn der Methode das intrinsische Lock/den Monitor des Objektes des Methode und gibt dieses am Ende wieder frei. Ist das Lock bereits vergeben, muss der aktuelle Thread warten bis er das Lock erhält.

```
1 synchronized method() { ... }
2
3 \par Alternativ kann auch einen synchronized block direkt
     spezifizieren. Dazu muss man zusätzlich noch das Objekt
     definieren, von welchem man das Lock verwenden möchte.
4 \begin{lstlisting}
5 synchronize(someObject) { ... }
```

Da jedes Objekt zur Synchronisierung verwendet werden kann, kann man auch innerhalb eines Objektes ein (oder mehrere) dummy Objekte erstellen, um unterschiedliche Teile separat locken zu können. Denn jedes Objekt hat exakt ein intrinsisches Lock/Monitor.

Die internen Locks sind reentrant. Das bedeutet: Man kann von einem synchronisierten Block anderen Code aufrufen, welcher das gleiche Lock benötigt. Wären die internen Locks nicht reentrant, müsse man erst das eigene Lock abgeben.

Es ist wichtig zu wissen, dass Exceptions innerhalb einem synchronized Block dazu führen, dass die Ausführung 'normal' abgebrochen wird und die Exception weitergereicht wird. Dies bedeutet, dass die gelockte Datenstruktur im Anschluss Inkonsistenten aufweisen kann. Jedoch wird das Lock immer wieder freigegeben. Dies ist anders als bei den externen locks.

9.7.2 externe Locks

Durch das 'java.util.concurrent.locks' package können in Java die externen Locks verwendet werden. Diese sind komplizierter zu verwenden, bieten im Gegenzug jedoch mehr Funktionalität: reader-writer Szenaraien, Interrupts, mehrere condition variables. Um sicherzugehen dass das Lock selbst bei einer Exception wieder freigegeben wird, verwendet man bei externen Locks üblicherweise 'try{...} finally {...}'-Blöcke.

Es stehen folgende Methoden zur Verfügung

- import java.util.concurrent.locks.Lock;

- import java.util.concurrent.locks.ReentrantLock;
- Lock lock = new Lock(); - lock erstellen, analog für Reentrant Lock
- lock.lock(); - lock nehmen
- lock.unlock(); - lock freigeben, falls gehalten (sonst Exception)
- lock.tryLock(); - wenn das Lock aktuell nicht frei ist, wird nicht gelockt
- lock.tryLock(long timeout, TimeUnit unit);
- lock.newCondition(); - neue condition Variable (unten erläutert)
- lock.lockInterruptibly(); - locken, sodass interrupt möglich sind

	Java synchronized	Java lock API
release	von der JVM gemanaged	manuell gemanaged
scope	critical sections auf Methoden/Blöcke begrenzt	methodenübergreifendes Locking möglich
waiting	Thread blockiert beim warten, kann nicht unterbrochen werden	'trylock()' reduziert die Lockdauer & 'lockInterruptedly()' ermööglicht Interrupts beim Locken
general	clean code, einfach zu maintainen, einfach Bugs zu vermeiden	flexibler, Fehleranfälliger

9.8 Condition Variables

Condition Variables sind wichtig, um bestimmte Probleme paralleler Programme zu umgehen. Z. B. Producer-Consumer Szenarien in welchen in einen Buffer/Queue geschrieben und aus dieser gelesen wird: Wir möchten sichergehen, dass die interne Struktur korrekt bleibt, dass es keinen Deadlock gibt (ein Consumer blockiert ohne was erhalten zu können), kein ineffizientes busy waiting, die komplizierte Verwendung von Semaphoren zum Zählen vermeiden, ...

Condition Variables ermöglichen es ein Lock freizugeben und wieder anzufragen, falls bestimmte Bedingungen erfüllt sind. In Java kann mit intrinsic Locks kann nur eine Bedingung getestet werden. Merkt ein Thread, dass seine Bedingung nicht wahr ist, gibt dieser das lock frei/ruft 'wait()' auf (dies geht entsprechend nur ein einem synchronisierten Block). Dann wird das Lock freigegeben und der Thread wird an eine interne Queue angehängt und wartet auf die Benachrichtigung, dass die Bedingung wahr ist. Andere Threads können diese Benachrichtigung mit 'notify()' und 'notifyAll()' senden. Notify wacht einen Thread auf und 'notifyAll()' benachrichtigt alle wartenden Threads. Auch diese Methoden können nur aufgerufen werden, wenn das Lock gehalten wird.

WICHTIG: Hat ein Thread 'wait()' aufgerufen und wird nun wieder aufgewacht, setzt die Ausführung normal fort. Allerdings muss nun die Bedingung erneut überprüft werden, da ggf. ein anderer Thread den Zustand bereits wieder verändert haben könnte oder es sich um einen spurious ("gurndlosen") wakeup gehandelt hat. Dies sollte mit einer while Schleife getan werden, sodass wieder geschlafen wird, falls die Bedingung weiterhin falsch ist.

Für ein besseres Verständnis in Java kann man sich an die Thread states erinnern (NEW, RUNNABLE, WAITING, BLOCKED, TERMINATED, ...). Wenn ein Thread

‘.wait()‘ aufruft, wird dieser an eine interne Queue angehängt und geht in den WAITING state. Durch notify/notifyAll kann er aus der Queue und WAITING in BLOCKED gesetzt werden, um wieder gescheduled zu werden. Dies ist die ’signal and continue’ Semantik. Es gibt allerdings auch noch andere signaling Semantiken.

- signal and wait

  Der signalisierende Prozess gibt das lock frei und geht auf die waiting to entry queue (NICHT die condition variable queue). Das lock wird direkt an den signalisierten Prozess weitergereicht.

- signal and continue

  Der signalisierende Prozess läuft weiter. Der signalisierte Prozess wird auf die entry queue (NICHT die condition variable queue) gesetzt.

- signal and exit

- signal and urgent wait

- ...

In diesem abschnitt haben wir bisher intrinsic locks betrachtet. Diese sind zwar einfach, allerdings begrenzt. Entsprechend betrachten wir nun externe locks (in Java). Haben wir ein ‘Lock lock‘, können wir beliebig condition variablen erstellen: ‘Condition someCondition = lock.newCondition();‘. Nun stehen die Methoden ‘lock.await()‘ (wie ‘wait()‘), ‘lock.signal()‘ (wie ‘notify()‘), und ‘lock.signalAll()‘ (wie ‘notifyAll()‘) zur Verfügung. Da das Objekt ‘lock‘ selbst ein intrinsic lock besitzt mussten die Namen anders gewählt werden.

Bei den externen Locks kann es auch zu spurious wakeup kommen.

Für ein Beispiel zur Verwendung und Optimierung eines Problems mit condition variables kann der Abschnitt Producer-Consumer Pattern verwendet werden.

9.8.1 Sleeping Barber Problem

Dies beschreibt ein fiktives Szenario nach E. Dijkstra

- Ein Friseur schneidet Haare. Wenn er fertig ist, überprüft er ob das Wartezimmer leer ist. Falls dies der Fall ist, schläft er.

- Wenn Kunden kommen überprüfen diese, ob der Friseur schläft. Falls dies der Fall ist, wachen sie diesen auf. Andernfalls setzen sie sich ins Wartezimmer.

Hier kann ein Deadlock auftreten wenn beide Akteure jeweils gleichzeitig ihr Kriterium überprüfen. Dann sitzen alle Kunden endlos im Wartezimmer während der Friseur endlos schläft.

Die Lösung des Problems ist die Einführung zusätzlicher counter:

- $p \leq 0 \Leftrightarrow$ buffer ist voll und $-p$ Produzenten warten

- $c \leq 0 \Leftrightarrow$ buffer ist leer und $-c$ Konsumenten warten

Für ein Beispiel zur Verwendung dieser Variablen, siehe ”Producer-Consumer Pattern”. Der Code dort ist jedoch nicht für das Sleeping Barber Problem an sich anfällig.

9.9 Reader-Writer Locks

Speicher/Register verhalten sich unterschiedlich, basierend auf den verwendeten Operationen:

- mehrere concurrent reads einer location: kein Problem
- mehrere concurrent writes einer location: Problem
- mehrere concurrent reads & writes einer location: Problem

Locks, wie bisher verwendet, überkompensieren. Wir können mehrere Threads den Zugriff erlauben, vorausgesetzt dass diese ausschließlich lesen möchten. Ein "reader/writer lock" kann drei Zustände haben:

- not held
- held for writing (von einem Thread)
- held for reading (von einem oder mehr Threads)

Wenn writers/readers die Anzahl der writer/reader sind, muss zu jeder Zeit gelten:

- $0 \leq \text{writers} \leq 1$
- $0 \leq \text{readers}$
- $\text{writers} \cdot \text{readers} = 0$

Unterstützte Methoden dieser abstrakten Datenstruktur sind:

- new - erstellt ein neues, not held lock
- acquire_write - blockiert falls aktuell 'held for reading' oder 'held for writing, andernfalls setzt auf 'held for writing' und setzt die Ausführung fort
- release_write - setzt auf 'not held', vorausgesetzt das write gehalten
- acquire_read - blocks falls aktuell 'held for writing', andernfalls setzte auf 'held for reading' und den reader counter um eines Erhöhen
- release_read - decrement den reader counter, falls 0 setze auf 'not held'

Implementierungen müssen sich zunächst für ein Fairness-Modell entscheiden. Bei unbekannten Implementationen sollte angenommen werden, dass writern Priorität gegeben wird. Üblicherweise erhält kein reader, der nach einem writer das lock anfragt, das lock bis der writer seine Schreiboperation beendet hat. Zudem existiert häufig die Möglichkeit ein reader lock auf ein writer lock 'upzugraden'. In Java steht diese Funktionalität nicht zur Verfügung. Dort benutzen wird 'java.util.concurrent.locks.ReentrantReadWriteLock'. Mit 'lock.readLock()' oder 'lock.writeLock()' erhalten wir ein Objekt, auf welchem wir jeweils 'lock()' und 'unlock()' aufrufen können.

Nun betrachten wir einige Implementationen, welche unterschiedlichen Fairness-Modellen entsprechen.

Version 1 Reader haben Priorität. Alle reader können locken. Writer locken nur, falls kein reader mehr lesen möchte.

```
1  class RWLock {
2    int writers = 0;
3    int readers = 0;
4    synchronized void acquire_read() {
5      while(writers>0)
6        try { wait(); }
7        catch(InterruptedException e) {}
8      readers++;
9    }
10   synchronized void release_read() {
11     readers--;
12     notifyAll();
13   }
14   synchronized void acquire_write() {
15     while(writers>0 || readers>0)
16       try { wait(); }
17       catch(InterruptedException e) {}
18     writers++;
19   }
20   synchronized void lrelease_write() {
21     writers--;
22     notifyAll();
23   }
24 }
```

Version 2 Writer haben Priorität. Sobald ein writer locken möchte, kann dieser mit Priorität fortfahren.

```
1  class RWLock {
2    int writers = 0;
3    int readers = 0;
4    int writersWaiting = 0;
5    synchronized void acquire_read() {
6      while(writers > 0|| writersWaiting > 0)
7        try { wait(); }
8        catch(InterruptedException e) {}
9      readers++;
10   }
11   synchronized void release_read() {
12     readers--;
13     notifyAll();
14   }
15   synchronized void acquire_write() {
16     writersWaiting++;
17     while(writers > 0||readers > 0)
18       try { wait(); }
19       catch(InterruptedException e) {}
20     writersWaiting--;
21     writers++;
22   }
23   synchronized void release_write() {
24     writers--;
25     notifyAll();
```

```
26    }
27  }
```

Version 3 Dies ist eine ziemlich faire Variante. Wenn ein writer terminiert können k aktuell wartende reader noch passieren, bevor der nächste writer an die Reihe kommen kann.

```
1  class RWLock {
2    int writers = 0;
3    int readers = 0;
4    int writersWaiting = 0;
5    int readersWaiting = 0;
6    int writersWait = 0;
7    synchronized void acquire_read() {
8      readersWaiting++;
9      while(writers>0 ||
10       (writersWaiting > 0 && writersWait <= 0))
11       try { wait(); }
12       catch(InterruptedException e) {}
13     readersWaiting--;
14     writersWait--;
15     readers++;
16   }
17   synchronized void release_read() {
18     readers--;
19     notifyAll();
20   }
21   synchronized void acquire_write() {
22     writersWaiting++;
23     while(writers>0 || readers>0 || writersWait>0)
24       try { wait(); }
25       catch(InterruptedException e) {}
26     writersWaiting--;
27     writers++;
28   }
29   synchronized void lrelease_write() {
30     writers--;
31     writersWait = readersWaiting;
32     notifyAll();
33   }
34 }
```

9.10 Weitere Betrachtungen

Bzgl. locking gibt es unterschiedliche Herangehensweisen. Als Teil von Lock Granularity werden wir fine-grained und coarse-grained locking betrachten. Darüber hinaus existiert auch optimistic und lazy synchronization. Diese werden als lock-free Methoden ebenso separat betrachtet. Locks sind fundamental pessimistisch: Es wird (auch wenn keine anderen Threads vorhanden sind) angenommen, dass es zu Interferenz kommt und wir locken müssen.

9.10.1 Lock Granularity

Bemerke, dass gelocke Codesegmente nicht mehr parallel ausgeführt werden. Stattdessen sind diese rein sequentiell. Dies bedeutet, dass jeder gelockte Block zwar zur Programmkorrektheit beiträgt, jedoch die Performance der Parallelität zu nicht macht. Entsprechend ist die Lock Granualrity wichtig. Grundsätzlich sagt man, dass nur das gelockt werden soll, was wirklich absolut notwendig ist. Die Granularity zu reduzieren ist zwar mit Aufwand verwunden, da man Safety Hazards ausschließen muss, bietet jedoch auch Performance Verbesserungen.

Corase-grained-locking ist einfacher zu implementieren, besonders wenn man die Struktur einer Datenstruktur verändern möchte. Fine-grained-locking hingegen ist besonders bei komplexen Datenstruktur häufig schwierig fehlerfrei zu implementieren. In der Praxis empfiehlt es sich mit einem corase locking anzufangen und diesen zunehmend zu verfeinern.

Als Beispiel werden wir ein set als linked list implementieren. (Nur zu demonstrativen Zwecken!) Eine corase-grained Implementation ist:

```
1  public class Set<T> {
2    private class Node {
3      T item;
4      int key;
5      Node next;
6    }
7    private Node head;
8    private Node tail;
9    public synchronized boolean add(T x) {...}
10   public synchronized boolean remove(T x) {...}
11   public synchronized boolean contains(T x) {...}
12 }
```

Nun fine-grained locking zu etablieren ist sehr fehleranfällig, da viele edge cases betrachtet werden müssen. Nur ein Element der Liste zu locken reicht z. B. nicht aus. Bei ‘remove(...)‘ nur das vorherige Element zu locken kann dazu führen, dass das remove ignoriert wird, dass ein anderes remove ignoriert wird, oder dass ein insert ignoriert wird. Das Problem: Auf das nächste Element wurde potenziell bereits zugegriffen. Entsprechend müssen wir dieses ebenfalls locken (hand-over-hand locking). Wir erhalten folgende ‘remove(...)‘ Methode:

```
1  public boolean remove(T item) {
2    Node pred = null;
3    Node curr = null;
4    head.lock();
5    try {
6      pred = head;
7      curr = pred.next;
8      curr.lock();
9      try {
10       while(curr.key < key) {
11         pred.unlock();
12         pred = curr;
13         curr = curr.next();
14         curr.lock();
15       }
16       if(curr.key == key) {
17         pred.next = curr.next; // delete
```

```
18          return true;
19        }
20        return false;
21      } finally { curr.unlock(); }
22    } finally { pred.unlock(); }
23  }
```

Wir erkennen, dass diese Implementation weiterhin Probleme hat. Konkret ist das Durchlaufen der linked liste sehr aufwendig und ein langsamer Thread kann andere (nach ihm die Liste durchlaufende threads) aufhalten. Betrachte optimistic & lazy synchronization als nächsten Ansatz.

9.10.2 Größe der Critical Section

Große critical sections reduzieren die Performance, da andere Threads unnötig blockiert werden. Kleine critical sections können hingegen schnell zu bugs führen, da zu kleine sections intermediate staet für alle Threads sichtbar machen können. Zudem kann häufiges Thread-switching und Cache trashing die performance weiter verringern.

10 Semaphores

Semaphoren können als Verallgemeinerung von Locks verstanden werden. Locks können keinen Zustandswechsel von Ressourcen kommunizieren und ebenso keine Ordnung des Ressourcenzugriffs. Faire locks ausgenommen, welche in der Praxis jedoch selten sind, da diese deutlich teurer zu implementieren sind.

Mit Semaphoren erlauben wir nicht immer nur einem Thread den Zugriff auf eine Ressource, sondern definieren eine obere Grenze der Threads, welche die Semphore halten dürfen. Formal: Eine Semaphore ist ein ganzzahliger abstrakter Datentyp mit einem initialen Wert $S \geq 0$ und zwei Operationen, welche jeweils vollständig atomar sind/sein müssen:

```
1  acquire(S):
2    wait until S>0
3    decrement(S)
4  release(S):
5    increment(S)
```

Alternativ verwendet man eine blockierende Queue, um die Energie- und Zeitverschwendung des spinlocks zu vermeiden:

```
1  acquire(S) // ATOMIC!!!
2    if S > 0:
3      decrement(S)
4    else:
5      put(S.Q, self)
6      block(self)
7  release(S) // ATOMIC!!!
8    if Q.S.isEmpty:
9      increment(S)
10   else:
11     get(Q.S, p)
12     unblock(p)
```

Häufig werden Semaphoren als mächtiger als Locks angesehen, obwohl eine Semaphore mit einem Lock und einer Variable implementiert werden kann. Mit Semaphoren können bestimmte Situationen z. B. eleganter gelöst werden. Haben wir zwei unabhängige Berechnungen können wir durch die Verwendung einer Semaphore (welche wir auf 0 initialisieren) feststellen wenn die Berechnung abgeschlossen ist, da wir eine bestimmte Reihenfolge vorgeben.

11 Barriers

Dieses Beispiel verdeutlicht ein Konzept, welches als rendezvous bekannt ist. Zwei Threads führen eine geteilte Berechnung durch. Nun möchten wir lokal wissen, wann die gemeinsame Berechnung beendet ist. Dazu muss klar sein, wenn beide Threads mit ihrer Berechnung fertig ist.

```
1  Thread A
2    xA=...;
3    lock();
4    x=x+xA;
5    unlock()
6  Thread B
7    xB=...;
8    lock();
9    x=x+xB;
10   unlock()
```

Ein rendezvous ist ein Ort (pro Thread) im Code, an welchem threads P und Q aufeinander warten müssen, um fortzufahren. Es handelt sich also um einen einzelnen Synchronisierungspunkt.

Der Begriff rendezvous wird typischerweise für eine Synchronisierung von zwei Threads verwendet. Barrier ist der allgemeinere Begriff, welcher sich auf das Problem für n Threads bezieht.

11.1 2 Thread Rendezvous

Dies ist eine einfache Implementation von 2 thread rendezvous.

```
1  Thread P
2    init:
3      P_Arrived=0
4    pre:
5      ...
6    rendezvous:
7      release(P_Arrived)
8      acquire(Q_Arrived)
9    post:
10     ...
11 Thread Q
12   init:
13     Q_Arrived=0
14   pre:
15     ...
16   rendezvous:
17     release(Q_Arrived)
18     acquire(P_Arrived)
```

```
19    post:
20      ...
```

11.2 Barriers

2 Threads Rendezvous ist gut, um das Problem und eine Lösung zu demonstrieren. Auf modernen Hochleistungsclustern gibt es jedoch normalerweise > 1 Million Threads auf 20K+ GPUs. Man verwendet ein BSP model (bulk-synchronous parallel model), um Synchronisation zu ermöglichen und gleichzeitig Paradigmen wie distributed memory zu unterstützen, die hier nicht betrachtet werden.

Hier werden wir eine turnstyle Implementierung betrachten. Dies ist ein gängiges Konzept, bei dem Threads nacheinander das turnstyle barrier passieren, sobald alle Threads am barrier angekommen sind. Damit dies ohne Race Conditions funktioniert, betrachten wir ein two-phase barrier.

```
1   init:
2     mutex = 1; (semaphore)
3     barrier1 = 0; (semaphore)
4     barrier2 = 1; (semaphore)
5     count = 0;
6   pre:
7     ...
8   barrier:
9     acquire(mutex)
10      count++
11      if (count==n)
12        release(barrier1)
13        acquire(barrier2)
14    release(mutex)
15
16    acquire(barrier1)
17    release(barrier1)
18
19    acquire(mutex)
20      count--
21      if(count==0)
22        acquire(barrier1)
23        release(barrier2)
24    release(mutex)
25
26    acquire(barrier2)
27    release(barrier2)
28  post:
29    ...
```

Diese beiden Phasen könnten auch aufgeteilt werden, so dass es sich um zwei getrennte barriers handelt. Dann müssten wir sicherstellen, dass diese beiden Schranken auch abwechselnd verwendet werden.

12 Producer-Consumer Pattern

Das Producer-Consumer-Pattern gehört zu den vorherrschenden parallelen Programmierungsmodellen, da häufig einige Threads Daten produzieren, die dann von einem anderen Thread

verwendet/weiterverarbeitet werden.

T_0 kann X berechnen und es an T_1 weitergeben. Dann berechnet T_1 X' und gibt es an T_2 weiter. Und so weiter. Dies entspricht einem data-flow program/einer pipeline. X muss nicht synchronisiert werden, da es immer nur von einem thread verwendet wird. Stattdessen muss ein Synchronisierungsmechanismus zur Übergabe von X existieren.

Wir betrachten nun Queues, um X durch die (data-flow) pipeline zu reichen. Jeder Thread/jede Node operiert fundemantel wie folgt. Dabei können auch mehrere Nodes vom gleichen Input konsumieren und für die gleiche Queue produzieren.

```
1 WHILE true:
2   input = q_in.dequeue()
3   output = do_something(input)
4   q_out.enqueue(output)
```

Für eine korrekte Implementation der Queue betrachten wir eine begrenzte FIFO queue als cirtuclar buffer. Bemerke: Circular buffer sind ein weit verbreitetes Konzept, welches auch an anderen Stellen eingesetzt werden kann. Es handelt sich um ein Array der Länge x mit zwei pointern 'in' (wo das nächste Element einzufügen ist) und 'out' (wo das nächste zu entfernende Element liegt). Bei enqueue/dequeue werden somit 'in'/'out' durch das array geschoben. Da es sich um ein endliches array handelt, müssen wir für die Indices modulare Arithmetik verwenden.

Nun folgt eine Beispiel-Implementation auf Basis von Semaphoren. Beachte, dass wir in einem circular buffer der Länge n nur $n-1$ Werte speichern, um den Zustand voll vom Zustand leer zu unterscheiden.

```
1  import java.util.concurrent.Semaphore;
2
3  class Queue {
4    int in, out, size;
5    long buf[];
6    Semaphore entries, credits, lock;
7
8    Queue(int s) {
9      size = s;
10     buf = new long[size];
11     in = 0;
12     out = 0;
13     entries = new Semaphore(0);
14       // use the counting feature of semaphores!
15     credits = new Semaphore(size);
16       // use the counting feature of semaphores!
17     lock = new Semaphore(1);
18       // binary semaphore (lock)
19   }
20   void enqueue(long x) {
21     try {
22       // IMPORTANT - THE ORDER OF ACQUIRING SEMAPHORES
23       // HERE IS IMPORTANT TO AVOID A DEADLOCK
24       credits.acquire();
25       lock.acquire();
26       buf[in] = x;
27       in = next(in);
28     } catch (InterruptedException ex) {
29     } finally {
30       lock.release();
```

```
31        entries.release();
32      }
33    }
34    long dequeue() {
35      long x=0;
36      try {
37        // IMPORTANT - THE ORDER OF ACQUIRING SEMAPHORES
38        // HERE IS IMPORTANT TO AVOID A DEADLOCK
39        entries.acquire();
40        lock.acquire();
41        x = buf[out];
42        out = next(out);
43      } catch (InterruptedException ex) {
44      } finally {
45        lock.release();
46        credits.release();
47      }
48      return x;
49    }
50  }
```

Die Verwendung von Semaphoren hier ist nicht optimal. Eigentlich möchten wir (nur) so lange warten, bis eine entsprechende Veränderung stattgefunden hat, sodass wir weiter machen können. Eine bessere Implementation würde entsprechend (zwei) condition variables für das verwendete Lock nutzen. Hier sind die entsprechend aktualisierten Methoden.

```
1   class Queue {
2     int in, out, size;
3     long buf[];
4
5     Queue(int s) {
6       this.size = s;
7       buf = new long[size];
8       in = 0;
9       out = 0;
10    }
11    synchronized void enqueue(long x) {
12      while(isFull())
13        try {
14          wait();
15        } catch (InterruptedException e) {}
16      doEnqueue(x);
17      notifyAll();
18    }
19    synchronized long dequeue() {
20      long x;
21      while(isEmpty())
22        try {
23          wait();
24        } catch (InterruptedException e) {}
25      x = doEnqueue(x);
26      notifyAll();
27      return x;
28    }
29  }
```

Wir verwenden 'notifyAll()', da zwei Bedingungen existieren, wir mit dem intrinsic lock allerdings nur eine conditional variable zur Verfügung haben. Um nicht nur den falschen Thread aufzuwachen müssen wir entsprechend alle Threads aufwachen. Alternativ können wir externe Locks verwenden, um diese Ineffizienz zu vermeiden (wobei die Verwendung externer Locks an sich ineffizienter ist).

```
1  class Queue {
2    int in = 0, out=0, size;
3    long buf[];
4    final Lock lock = new ReentrantLock();
5    final Condition notFull = lock.newCondition();
6    final Condition notEmpty = lock.newCondition();
7    Queue(int s) {
8      size = s;
9      buf = new long[size];
10   }
11   void enqueue(long x) {
12     lock.lock(); // hier sollte noch
13         try{}finally{} verwendet werden
14     while (isFull())
15       try {
16         notFull.await();
17       } catch (InterruptedException e){}
18     doEnqueue(x);
19     notEmpty.signal();
20     lock.unlock();
21   }
22   long dequeue() {
23     long x;
24     lock.lock(); // hier sollte noch
25         try{}finally{} verwendet werden
26     while (isEmpty())
27         try {
28         notEmpty.await();
29       } catch (InterruptedException e){}
30     x = doDequeue();
31     notFull.signal();
32     lock.unlock();
33     return x;
34   }
35 }
```

Dies ist jedoch weiterhin nicht optimal, da die signale auch gesendet werden, falls kein thread wartet. Da das Senden von Signalen auf dem low-level ziemlich teuer ist bietet sich eine weitere Optimierung an.

```
1  class Queue {
2    int in = 0;
3    int out = 0;
4    int size;
5    long buf[];
6    final Lock lock = new ReentrantLock();
7    int c = 0;
8    final Condition notFull = lock.newCondition();
9    int p;
10   final Condition notEmpty = lock.newCondition();
```

```
11    Queue(int s) {
12      size = s;
13      p = size-1;
14      buf = new long[size];
15    }
16    void enqueue(long x) {
17      lock.lock();
18      p--;
19      if(p<0)
20        while(isFull())
21          try { notFull.await() }
22          catch(InterruptException e) {}
23      doEnqueue(x);
24      c++;
25      if(c<=0)
26        notEmpty.signal();
27      lock.unlock();
28    }
29    long dequeue() {
30      long x;
31      lock.lock();
32      c--;
33      if(c<0)
34        while(isEmpty())
35          try { notEmpty.await() }
36          catch(InterruptException e) {}
37      x = doDequeue();
38      p++;
39      if(p<=0) notFull.signal();
40      lock.unlock();
41      return x;
42    }
43  }
```

13 Optimistic & Lazy Synchronization

Im Abschnitt "Lock Granularity" haben wir bereits gesehen, dass zum Entfernen einer Node das Locken zweier Nodes notwendig ist, um Inkonsistenten in der Datenstruktur zu vermeiden. Wir hatten zwei Probleme identifiziert: Hand-over-Hand locking ist sehr teuer, um die passende Stelle in der Liste zu finden und einzelne Threads können andere aufhalten.

13.1 Optimistic Synchronization

Optimistic synchronization beschreibt das Konzept, bei welchem wir zunächst annehmen, dass alles auch ohne locks gut laufen wird. Erst an einer kritischen Stelle locken wir und validieren, dass wirklich alles gut gelaufen ist.

Wir können erneut die linked liste zur Implementation eines sets betrachten. Sobald wir diese ohne locks durchlaufen haben und die passende Stelle gefunden und gelockt haben, müssen wir folgende Validierung durchführen:

```
1  public Boolean validate(Node pred, Node curr) {
2    Node node = head;
```

```
3   while(node.key <= pred.key) { // reachable?
4     if(node == pred)
5       return pred.next = curr; // connected?
6     node = node.next;
7   }
8   return false;
9 }
```

Die Korrektheit dieser Implementation kann selbstverständlich bewiesen werden. Die Vorteile sind klar:

- keine contention beim list traversal
- das traversal der liste ist wait-free (formale Definition später)
- weniger lock Nutzung/acquisitions

Doch nichts kommt ohne Nachteile aus:

- die liste muss zwei mal traversiert werden
- die contains Methode müsste auch locks akquirieren
- die Implementation ist nicht starvation-free

13.2 Lazy Synchronization

Lazy synchronization versucht, optimistic synchronization weiter zu verbessern. Jetzt gehen wir nicht nur davon aus, dass die Dinge richtig laufen, sondern wir beheben auch nicht unbedingt das Problem, wenn die Dinge nicht richtig waren. Wir nehmen unsere Änderung vor, vermerken sie, sind aber zu faul, die Datenstruktur zu aktualisieren, wenn dies nicht sofort funktioniert. Aber da wir eine Notiz erstellt haben, können andere threads helfen, wenn sie diese Notiz in der Datenstruktur selbst sehen.

Das Beispiel des linked-list sets ist hier nicht optimal, da wir unsere Änderung immer selbst vornehmen können. Wir führen als Notiz ein, dass wir Vermerken, ob eine Node entfernt wurde. Somit müssen wir die liste nicht zwei Mal traversieren.

```
1  public boolean remove(T item) {
2    int key = item.hashCode();
3    while(true) {
4      Node pred = this.head;
5      Node curr = head.next;
6      while(curr.key < key) {
7        pred = curr;
8        curr = curr.next;
9      }
10     pred.lock();
11     try {
12       curr.lock();
13       try {
14         if(!pred.marked && !curr.marked
15             && pred.next == curr) {
16           if(curr.key != key) {
17             return false;
18           } else {
19             curr.marked = true;
```

```
20              pred.next = curr.next;
21              return true;
22            }
23          }
24        } finally { curr.unlock(); }
25      } finally { pred.unlock(); }
26    }
27  }
```

Ein Vorteil von lazy synchronization in unserem Beispiel ist jedoch, dass wir nun auch die contains Methode ohne locks/wait-free (formale Definition später) implementieren können.

```
1  public boolean contains(T item) {
2    int key = item.hashCode();
3    Node curr = this.head;
4    while(curr.key < key) {
5      curr = curr.next;
6    }
7    return curr.key == key && !curr.marked;
8  }
```

13.3 Lazy-Skip-List

Die Lazy-Skip-List ist ein Beispiel von lazy synchronization. Lazy-Skip Lists werden, anders als linked-list basierte sets, auch in der Praxis verwendet. Denn da die Liste nicht immer komplett traversiert werden muss, haben diese relativ geringe Kosten haben (nur noch erwartete logarithmische Laufzeit).

Der Grund, weshalb wir keine AVL trees o. Ä. verwenden ist, dass korrektes fine-grained Locking/lazy synchronization dieser sehr sehr schwer ist, da es sich häufig um globale Operationen von sehr unterschiedlichem Ausmaß handelt.

Eine skip list ist eine multi-level Liste, in welcher nodes unterschiedliche Höhen haben. Die Höhe der Nodes wird zufällig bestimmt mit $P(\text{Höhe} = n) = \frac{1}{2}^n$. Zusätzlich existiert ein head und tail mit jeweils höchstem Level, welches höher als alle anderen Levels ist.

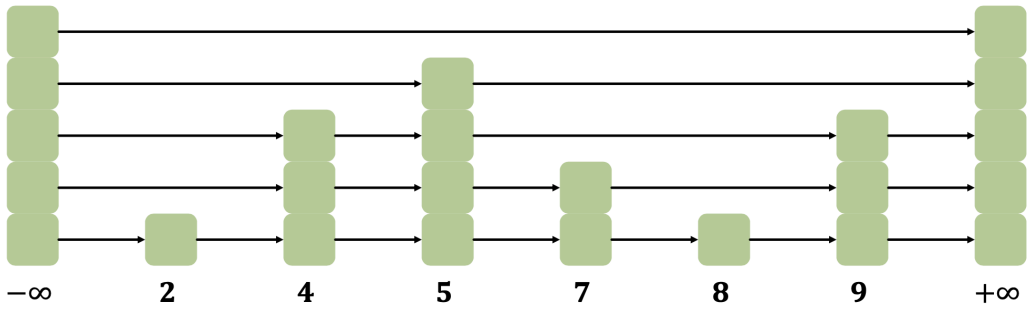


Figure 8: Lazy Skip List Visualizsierung

- Searching/Sequential find
  - Starte mit dem höchsten Level des heads als aktuelles Level
  - Suche das nächste Auftreten dieses Levels
    - Gefunden

- Nachher: Iteration vom nächstes Auftreten (ggf. passiert → nicht enthalten)
- vorher: Iteration mit nächst tieferem Level

- Adding
  - Wir finden die Vorgänger auf allen Leveln (z. B. durch modifiziertes searching), locken diese und validieren. Schlägt die Validierung fehl: Iterieren.
  - Erstellen einer neuen Node mit zufälliger Höhe und Einfügen (splice the list).
  - Markieren aller Level der Node als fully linked und unlocken alle Vorgänger.
- Removing
  - Wir finden die Vorgänger auf allen Leveln (z. B. durch modifiziertes searching), locken diese und validieren. Schlägt die Validierung fehl: Iterieren.
  - Locken das victim.
  - Logical remove: Setzen die remove Marker aller Level des Victims.
  - Physical remove: Redirect pointers.
  - Unlock aller locks.
- Contains
  - Searching/sequential find
  - Validieren, dass das Element logic entfernt wurde & dass alle Element fully linked sind.

14 Lock-free/non-blocking Programming/Synchronization

Bisher haben wir locks sehr ausgiebig betrachtet. Allerdings können Locks zu Deadlocks und anderen Probleme führen. In sicherheitskritischen Systemen (Flugzeugen, Atomkraftwerken, ...) dürfen entsprechend keine Locks verwendet werden, um Funktionalität zu garantieren. Abgesehen von reliability können auch weitere Probleme betrachtet werden.

- spinlock
  - Fairness/kein FIFO Verhalten ohne zusätzliche queues
  - Verschwendung von Ressourcen, besonders bei contention
  - kein Notification System
- scheduled locks
  - Unterstützung von der Laufzeitumgebung (OS, scheduler) notwendig
  - Datenstrukturen der Locks müssen geschützt werden (häufig mit spinlocks)
  - hohe wakeup Latenz (durch den OS scheduler)
- allgemein: Overhead (besonders bei hoher contention)

Die Probleme lassen sich zusammenfassen: Pessimismus ("Brechstangen"-Tool), Overhead (Performance-Reduktion, Grenzen entsprechend Amdahl's law) und Blocking Semantics (Deadlocks, Starvation, Sudden Thread deaths, ...)

Lock-free synchronization ist charakterisiert durch:

- **lock-freedom:** Mindestens ein Thread macht Fortschritt, selbst wenn andere threads concurrent laufen. Dies impliziert globalen Fortschritt aber schließt Starvation nicht aus.
- **wait-freedom:** Alle Threads machen eventuell Fortschritt. Dies Garantiert, dass keine Starvation auftritt.

Im Vergleich zu lock-basierter Programmierung ergibt sich:

	non-blocking (no locks)	blocking (locks)
everyone makes progress	wait-free	starvation-free
someone makes progress	lock-free	deadlock-free

Figure 9: Vergleich blocking und non-blocking Charakteristika

Bei locking/blocking Implementierungen kann ein einzelner Threads alle anderen Threads beliebig lange aufhalten. Bei non-blocking Implementierungen ist dies nicht möglich. Thread death oder andere Verzögerungen eines Threads halten nie den globalen Fortschritt auf.

Um non-blocking Algorithmen implementieren zu können, muss man entsprechende Unterstützung der Hardware voraussetzen. Währen load linked/store conditional sehr gut zu nutzen sind, müssen wir auf x86 heute meist mit CAS, welches mächtiger als TAS ist, arbeiten. CAS kann wait-free von der Hardware zur Verfügung gestellt werden.

Hierbei handelt es sich um ein Beispiel eines wait-free Algorithmus. Dies implementiert einen wait-free Counter.

```
1  public class CASCounter {
2    private AtomicInteger value;
3    public int getVal() {
4      return value.get();
5    }
6    // increment and return new value
7    public int inc() {
8      int v;
9      do {
10       v = value.get();
11     } while (!value.compareAndSet(v, v+1));
12     return v+1;
13   }
14 }
```

Wie auch in dieser Implementierung muss man bei non-blocking Algorithmen eine Operation häufig mehrmals versuchen, bis sie erfolgreich ist. Bei geringer contention ist dies natürlich praktisch kein Problem. Bei steigender contention jedoch schon, das atomic Operationen sehr teuer sind und häufig fehlschlagen. Durch die Verwendung eines exponential backoffs beim retry kann die Performance im Vergleich zu blocking Algorithmen jedoch deutlich erhöht werden.

Ein großes Problem mit CAS ist, dass ein positives return suggeriert, dass in der Zwischenzeit kein anderer Thread geschrieben hat. Dies ist jedoch nicht korrekt, da ein Wert auch zwei oder mehr Mal geändert sein worden kann. Dieses (in der Tat sehr gravierende Problem) wird ABA Problem genannt. Instruktionen wie load linked/store conditional können nicht zum ABA Problem führen, sind aber weniger verbreitet als CAS.

14.1 Lock-free Stack

Beachte, dass das ABA Problem die Korrektheit dieser Implementierung beeinflusst. Gegeben eine Liste mit mind. 3 Elementen, welche wir 1, 2, 3, ... nennen. Lasse Thread A pop() ausführen und bereits head=1 und next=2 gespeichert haben. Nun führt Thread B pop():1, pop():2, push(1) aus, wobei die Referenz für 1 identisch bleibt. Der Stack hat dann die Form 1, 3, ... Der CAS Test von Thread A hat nun Erfolg und der nun invalide Stack hat die Form 2, 3, ...

```
1  public static class Node {
2    public final Long item;
3    public Node next;
4
5    public Node(Long item) {
6      this.item = item;
7    }
8
9    public Node(Long item, Node n) {
10     this.item = item;
11     this.next = n;
12   }
13 }
14
15 public class ConcurrentStack {
16   AtomicReference<Node> top = new AtomicReference<Node>();
17   public void push(Long item) {
18     Node newi = new Node(item);
19     Node head;
20     do {
21       head = top.get();
22       newi.next = head;
23     } while (!top.compareAndSet(head, newi));
24   }
25   public Long pop() {
26     Node head, next;
27     do {
28       head = top.get();
29       if (head == null)
30         return null;
31       next = head.next;
32     } while (!top.compareAndSet(head, next));
33     return head.item;
34   }
35 }
```

14.2 Lock-free List-Based Set

Es ist verlockend add einfach mittels ‘CAS(vorher.next,nachher,neu)‘ (wobei neu auf nachher zeigt) und remove mittels ‘CAS(vorher.next,entfernen,nachher)‘ zu implementieren. Dies scheitert jedoch an gleichzeitigen Aufrufen von add/remove.

Die Idee zur Lösung des Problems ist das Verwenden eines mark-bits, welches angibt, ob ein Element logisch entfernt wurde. Das physische Entfernen kann dann separat erledigt werden. Fürs remove haben wir ‘CAS(remove.mark,false,true)‘ und ‘CAS(vorher.next,remove,nachher)‘ und für add ‘!vorher.mark‘ und ‘CAS(vorher.next,nachher,neu)‘, was jedoch nur funk-

tioniert, wenn diese beiden atomics jeweils auch zusammen atomar ausgeführt werden. Ansonsten kann der Zustand zwischendurch auch wieder invalidiert werden.

Die Herausforderung ist, atomar zwei CASs durchzuführen. sind der pointer und das mark-bit zwei separate Variablen benötigen wir Double CAS, welches moderne ISAs jedoch nicht zur Verfügung stellen. Entsprechend müssen wir den pointer und das mark bit in einer Speicherstelle kombinieren. Dies ist möglich, da moderne CPUs 64-bit Speicherstellen haben, die Adressen von pointern jedoch nie alle 64 bits benötigen (schließlich haben wir heute nie nur annähernd 18 Petabytes RAM).

Um diesen mark-bit zu einer Referenz hinzuzufügen bietet Java folgende Methode an:

```
1  Java.util.concurrent.atomic
2  AtomicMarkableReference<V> {
3    boolean attemptMark(V expectedPointer, boolean newMark)
4    boolean compareAndSet(V expectedPointer, V newPointer,
5      boolean expectedMark, boolean newMark)
6
7    V get(boolean[] markHolder)
8    V getPointer()
9    boolean isMarked()
10   set(V newPointer, boolean newMark)
11 }
```

Dies ist eine Implementierung (mit Teilweise Pseudo-Code) in Java. Dieser Code entspricht dem Konzept von lazy synchronization, da nur das logische Entfernen garantiert wird. Das physische Entfernen kann Fehlschlagen. Dies ist allerdings nicht weiter schlimm, da es von anderen Threads übernommen werden kann. Konkret ist dies in der 'find()' Methode implementiert. Dies ist ein wiederkehrender Ansatz, besonders in lock- und wait-free Algorithmen. Dies ist NICHT wait-free, sondern nur lock-free.

```
1  class Window {
2    public Node pred;
3    public Node curr;
4    Window(Node pred, Node curr) {
5      this.pred = pred;
6      this.curr = curr;
7    }
8  }
9  public Window find(Node head, int key) {
10   Node pred = null, curr = null, succ = null;
11   boolean[] marked = {false};
12   while (true) {
13     pred = head;
14     curr = pred.next.getReference();
15     boolean done = false;
16     while (!done) {
17       marked = curr.next.get(marked);
18       succ = marked[1:n]; // pseudo-code to get next ptr
19       while (marked[0] && !done) { // marked[0] is marked bit
20         if pred.next.compareAndSet(curr, succ, false, false) {
21           curr = succ;
22           succ = curr.next.get(marked);
23         }
24         else done = true;
25       }
26       if (!done && curr.key >= key)
```

```
27          return new Window(pred, curr);
28        pred = curr;
29        curr = succ;
30      }
31    }
32  }
33  public boolean remove(T item) {
34    boolean snip;
35    while (true) {
36      Window window = find(head, key);
37      Node pred = window.pred, curr = window.curr;
38      if (curr.key != key) {
39        return false;
40      } else {
41        Node succ = curr.next.getReference();
42        snip = curr.next.attemptMark(succ, true);
43        if (!snip)
44          continue;
45        pred.next.compareAndSet(curr, succ, false, false);
46        return true;
47      }
48    }
49  }
50  public boolean add(T item) {
51    boolean splice;
52    while (true) {
53      Window window = find(head, key);
54      Node pred = window.pred, curr = window.curr;
55      if (curr.key == key) {
56        return false;
57      } else {
58        Node node = new Node(item);
59        node.next = new AtomicMarkableRef(curr, false);
60        if (pred.next.compareAndSet(curr, node, false, false))
61          return true;
62      }
63    }
64  }
```

14.3 Lock-free Unbounded Queue

(Theoretically) Unbounded queues sind in verschiedenen Anwendungen von großer Bedeutung (OS Kernel, Scheduler, ...). Wir erstellen nun eine lock-free Implementierung. In den meisten Situationen haben wir keinen Garbage Collector, wie in Java, wodurch das ABA Problem akut wird.

Im Weiteren betrachten wir nun die Queue mit einem sentinel head pointer, welcher $-\infty$ entspricht. Dies hat praktische Gründe für die Implementierung, da wir dann die Sonderfälle von Null-Pointern nicht beachten müssen und enqueue/dequeue nur auf tail/-head operieren muss. Enqueue fügt eine Referenz an tail an und bewegt tail eins nach rechts. Dequeue bewegt head eins nach rechts, falls die queue nicht leer ist.

Dies funktioniert allerdings nicht trivial, da bei enqueue das hinzufügen der Referenz getrennt vom Verschieben von tail stattfindet. In der Zwischenzeit kann bereits ein anderer Thread versuchen zu enqueuen, welcher dann einen inkonsistenten State vorfindet. Da wir

einen lock-freien Algorithmus implementieren möchten, können wir andere Threads nicht warten lassen bis das Problem gelöst ist. Entsprechend verwenden wir wieder den Ansatz von lazy synchronization: Wir akzeptieren den State und lassen andere Threads helfen ihn zu reparieren.

Nun betrachten wir eine Implementierung. Beginnend mit einer Node:

```
1  public class Node<T> {
2    public T item;
3    public AtomicReference<Node> next;
4
5    public Node(T item) {
6      next = new AtomicReference<Node>(null);
7      this.item = item;
8    }
9
10   public void SetItem(T item) {
11     this.item = item;
12   }
13 }
```

Dies ist die eigentliche Implementeation der Queue:

```
1  public class NonBlockingQueue extends Queue {
2    AtomicReference<Node> head = new AtomicReference<Node>();
3    AtomicReference<Node> tail = new AtomicReference<Node>();
4
5    public NonBlockingQueue() {
6      Node node = new Node(null);
7      head.set(node); tail.set(node);
8    }
9
10   public void enqueue(T item) {
11     Node node = new Node(item);
12     while(true) {
13       Node last = tail.get();
14       Node next = last.next.get();
15       if (next == null) {
16         if (last.next.compareAndSet(null, node)) {
17           tail.compareAndSet(last, node);
18           return;
19         }
20       }
21       else
22         tail.compareAndSet(last, next);
23     }
24   }
25
26   public T dequeue() {
27     while (true) {
28       Node first = head.get();
29           Node last = tail.get();
30           Node next = first.next.get();
31           if (first == last) {
32         if (next == null)
33           return null;
34         else
35           tail.compareAndSet(last, next);
```

```
36        } else {
37          T value = next.item;
38          if (head.compareAndSet(first, next))
39            return value;
40        }
41      }
42    }
43  }
```

14.4 Reuse & ABA Problem

Das ABA Problem wurde bereits vorher erwähnt. Nun wird es im Detail erläutert. Es bezieht sich auf Situationen, in welchen eine Datenstruktur so verändert und (teilweise) zurück geändert wird, sodass es für eine CAS Operation so scheint, als seien keine Änderungen vorgenommen worden. Dies kann passieren, wenn eine Datenstruktur aus Objekten besteht, welche in dieser umsortiert werden. Z. B. können wir einen Stack betrachten.

```
1  public class ConcurrentStack {
2    AtomicReference<Node> top = new AtomicReference<Node>();
3
4    public void push(Long item) {
5      Node newi = new Node(item);
6      Node head;
7      do {
8        head = top.get();
9        newi.next = head;
10     } while (!top.compareAndSet(head, newi));
11   }
12
13   public Long pop() {
14     Node head, next;
15     do {
16       head = top.get();
17       if (head == null)
18         return null;
19       next = head.next;
20     } while (!top.compareAndSet(head, next));
21     return head.item;
22   }
23 }
```

Im folgenden Beispiel stehen 1, 2, 3, 4, ... für Objekte. Die mehrmalige Verwendung von 1 bezieht sich also auf das gleiche Objekt.

```
1  1, 2, 3, 4, ...
2    A: pop() wird aufgerufen, head=1, next=2, dann Ausführung
       pausiert
3    B: pop() wird aufgerufen, head=1, next=2, entfernt mit CAS(head
       ,1,2)
4  2, 3, 4, ...
5    B: pop() wird aufgerufen, head=2, next=3, entfernt mit CAS(head
       ,2,3)
6  3, 4, ...
7    B: push(1) wird aufgerufen, head=1, next=2, entfernt mit CAS(head
       ,1,2)
```

```
1, 3, 4, ...
  A: Ausführung setzt fort, CAS(head,1,2) ist erfolgreich
2, 3, 4, ...
```

Dies ist nicht legitim. Denn wir haben 3 pop() und 1 push() aufgerufen. Es sollten also 2 Elemente entfernt worden sein. Allerdings ist nur 1 nicht mehr auf dem Stack.

Dieses Problem ist noch gravierender, wenn wir Programmiersprachen ohne Garbage Collector betrachten. Denn in diesen als Teil vom Speichermanagement Objekte häufig wiederverwendet. Reuse von Ressourcen passiert auf allen systemebenen OS, Sprache/-Compiler, ... Für unseren Fall betrachten wir zu demonstrativen Zwecken NodePooling.

```
1  public class NodePool {
2    AtomicReference<Node> top new AtomicReference<Node>();
3
4    public void put(Node n) {
5      Node head;
6      do {
7        head = top.get();
8        n.next = head;
9      } while (!top.compareAndSet(head, n));
10   }
11
12   public Node get(T item) {
13     Node head, next;
14     do {
15       head = top.get();
16       if (head == null)
17         return new Node(item);
18       next = head.next;
19     } while (!top.compareAndSet(head, next));
20     head.item = item;
21     return head;
22   }
23 }
```

Beachte, dass der NodePool natürlich auch für das ABA Problem anfällig ist. Dies lässt sich (wie weiter unten klar wird) mit verschiedenen Ansätzen lösen. Z. B. können diese Pools thread-local sein.

Wir erhalten einen aktualisierten Stack mit dem NodePool.

```
1  public class ConcurrentStack {
2    AtomicReference<Node> top = new AtomicReference<Node>();
3
4    public void push(Long item) {
5      Node head;
6      Node new = pool.get(item);
7      do {
8        head = top.get();
9        new.next = head;
10     } while (!top.compareAndSet(head, new));
11   }
12
13   public Long pop() {
14     Node head, next;
15     do {
16       head = top.get();
```

```
17        if (head == null)
18          return null;
19        next = head.next;
20      } while (!top.compareAndSet(head, next));
21      Long item = head.item;
22      pool.put(head);
23      return item;
24    }
25  }
```

Dieser Code wird manchmal funktionieren und machmal nicht funktionieren. Die Fälle, in welchen er nicht funktioniert, lassen sich folgendermaßen erklären.

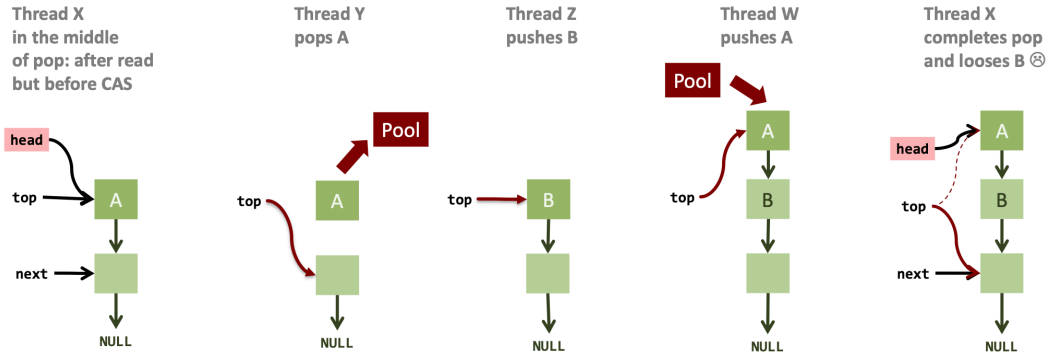


Figure 10: ABA Problem durch Reuse

Das ABA Problem tritt auf, wenn ein thread nicht erkennt, dass eine einzelne Speicherstelle vorübergehend von einer anderen thread verändert wurde, und daher fälschlicherweise annimmt, dass der Gesamtzustand nicht verändert wurde.

Es existieren verschiedene Ansätze um das ABA Problem zu lösen:

- DCAS (Double CAS)

  Steht auf den meisten modernen Plattformen nicht zur Verfügung.

- Garbage Collection

  Der Garbage Collector muss selbst lock-free und ohne Garbage Collector implementiert sein. Dies wäre für Kernel-Operationen viel zu langsam. Zudem kann das ABA Problem auch ohne Garbage Collector auftreten.

- Pointer Tagging
- Hazard Pointers
- Load Linked/Store Conditional
- Transactional Memory

14.5 Pointer Tagging

Pointer Tagging verhindert das ABA Problem nicht, sondern reduziert nur drastisch die Wahrscheinlichkeit, dass es auftritt. Wie zuvor nutzt Pointer Tagging aus, das Adressen

nicht alle 64 bits moderner Speicherstellen belegen. Dazu weisen wir den Speicher an, aligned addresses (z. B. nur solche, welche durch 32 teilbar sind) zu verwenden. Dann bleiben uns 5 freie freie bits im Register.

Beim pointer tagging nutzen wir diese freien bits nun, um einen einfachen Zählen zu implementieren. Immer wenn eine Änderung an der Variable/Referenz vorgenommen wird, erhöhen wir den Zähler um 1 (modular). Dann hat man nur validen Zugriff, wenn die Daten unverändert geblieben sind oder genau ein Vielfaches von x Mal verändert wurden und dann wieder den gleichen Wert/die gleiche Referenz haben. In unserem 5 bit Beispiel entspricht dies 31 weiteren Veränderungen. Das dies passiert ist sehr unwahrscheinlich.

14.6 Hazard Pointers

Das ABA Problem stammt von der Wiederverwendung von Referenzen. Wenn Threads erkennen könnten, dass ein Objekt noch nicht wiederverwendet werden sollte, weil noch ein anderer thread mit diesem arbeitet, könnten alle Schwierigkeiten beseitigt werden.

Die Idee von Hazard Pointern ist für jedes Objekt ein array mit n (Anzahl der threads) Elementen zu nutzen. Jeder Thread speichert in diesem Array eine Referenz, wenn er mit einem Objekt arbeitet, dass noch nicht wiederverwendet werden sollte, i.e., es erstellt einen hazard pointer. Bevor nun ein anderer Thread ein Objekt wiederverwendet, überprüft es alle Einträge des hazard arrays.

```
1  public class NonBlockingStackPooledHazardGlobal extends Stack {
2    AtomicReference<Node> top = new AtomicReference<Node>();
3    NodePoolHazard pool;
4    AtomicReferenceArray<Node> hazarduous;
5
6    public NonBlockingStackPooledHazardGlobal(int nThreads) {
7      hazarduous = new AtomicReferenceArray<Node>(nThreads);
8      pool = new NodePoolHazard(nThreads);
9    }
10
11   boolean isHazardous(Node node) {
12     for (int i = 0; i < hazarduous.length(); ++i)
13       if (hazarduous.get(i) == node)
14         return true;
15     return false;
16   }
17
18   void setHazardous(Node node) {
19     hazarduous.set(id, node); // id is current thread id
20   }
21
22   public int pop(int id) {
23     Node head, next = null;
24     do {
25       do {
26         head = top.get();
27         setHazardous(head);
28       } while (head == null || top.get() != head);
29       next = head.next;
30     } while (!top.compareAndSet(head, next));
31     setHazardous(null);
32     int item = head.item;
33     if (!isHazardous(head))
```

```
34        pool.put(id, head);
35      return item;
36    }
37
38    public void push(int id, Long item) {
39      Node head;
40      Node newi = pool.get(id, item);
41      do{
42        head = top.get();
43        newi.next = head;
44      } while (!top.compareAndSet(head, newi));
45    }
46  }
```

15 Transactional Memory

Transactional Memory ist eine weitere Möglichkeit des lock-free programming. Aus Perspektive des Programmierers ist diese anderen Ansätzen überlegen, da die Programmierung einfacher und weniger Fehleranfällig ist. Die Unterstützung durch die Hardware oder Software ist allerdings ziemlich kompliziert, sodass bis heute die Performance der primäre Bottleneck ist.

Wir betrachten zunächst erneut einige Schwierigkeiten des von blocking sowie non-blocking Programmen:

- Deadlocks
- Convoying (lock/resource holding thread ist descheduled und behindert andere Threads)
- Priority Inversion (low-priority threads können high-priority threads aufhalten)
- Data & Locks nicht verknüpft (umfangreiche Dokumentation & Disziplin notwendig)
- Locking overhead
- Lack of composability (keine Konvention für mehrere Locks/Ressourcen → hoher Aufwand für Dokumentation)
- Pessimism (Locks handeln immer entsprechend des worst-case Szenarios)
- 'Hard-wired' locking (Änderungen am Locking Schema bedeuten Änderungen am gesamten Programm)
- Inkonsistente states (non-blocking Programme sind lazy, zusätzlicher Aufwand, DCAS wegen hoher Kosten/nicht Verfügbar keine Alternative)

Das Konzept von Transactional Memory baut auf atomic blocks auf. Ein solcher atomic block/transaction kann mehrere Instruktionen enthalten. Die Anforderung ist, dass diese nach außen sichtbar atomar ausgeführt werden.

```
1 atomic {
2   instruction1
3   instruction2
4   ...
5 }
```

Der Ansatz ist ähnlich zum locking. Der Unterschied liegt in der Ausführung des atomic blocks, welcher declarative ist. Es wird nur die Anforderung gestellt, dass der block effektiv atomar auszuführen ist. Anders als bei locks wird allerdings keine Vorgabe gemacht, wie dies zu geschehen hat. Die Umsetzung wird dem System (Hardware und/oder Software) überlassen. Z. B. wird nicht grundsätzlich gefordert, dass nur ein Thread den block ausführen kann.

Dieser Ansatz hat mehrere Vorteile:

- TM ist einfacher und weniger Fehleranfällig
- TM ist high-level und bietet mehr Abstraktion für den Programmierer
- TM ist composable
- TM ist optimistisch (die Ausführung eines Threads wartet grundsätzlich nicht auf andere Threads)

Die Idee ist ähnlich zu Datenbanken, welche den ACID properties genügen: Atomicity, Consistency, Isolation, Durability. Alles bis auf Durability ist auch hier gegeben, wobei C von der korrekten Semantik des Programms abhängt.

Schwierigkeiten dieses Konzeptes sind:

- hohe Performance zu erreichen ist schwierig
- TM ist nicht etabliert
- es können keine I/O Operationen durchgeführt werden (nicht in Isolation möglich)

TM implementiert atomicity indem Änderungen eines atomic blocks atomar sichtbar werden und andere Threads nur den Ausgangs- oder Endzustand, jedoch keinen Zwischenzustand, sehen können. Jeder Thread läuft in einem atomic block also in Isolation von anderen threads. Damit intermediate states für andere threads nicht sichtbar sind gibt es zwei Ansätze:

- Erstellen eines Snapshots aller Daten zum Beginn
- (early) abort sobald inconsistencies festgestellt werden (Allgemein kann man denken, dass zwei Blöcke sequentiell ausgeführt werden, um schnell zu sehen, ob Inkonsistenten/Fehler auftreten.)

Ein concurrent Control (CC) Mechanism ist für den korrekten Umgang mit abzubrechenden transactions verantwortlich.

Weitere Design-Entscheidungen bei Transactional Memory sind:

- strong vs. weak isolation

  Strong isolation bedeutet, dass transactions auch mit geteilten Daten arbeiten können, welche außerhalb von transactions bearbeitet werden. Weak isolation beschreibt, dass nur Daten bearbeitet werden dürfen, welche ausschließlich in transactions bearbeitet werden.

  Bei reference-based STM Implementierungen ist der veränderbare sate in besonderen Variablen, welche nur innerhalb von transactions bearbeitet werden können. Alle anderen Daten können in transactions dann nur gelesen werden.

- nested transactions

  Flat nesting beschreibt, dass wenn ein innerer atomic block abbricht, so muss auch der äußere atomic block abbrechen. Wenn der innere atomic block committed, ist dies zunächst nur für den äußeren atomic block, jedoch nicht für andere atomic blocks, sichtbar.

  Closed nesting beschreibt, dass das Abbrechen des inneren atomic blocks nicht direkt zum Abbruch des äußeren atomic blocks führt. Wenn der innere atomic block committed, ist dies zunächst nur für den äußeren atomic block, jedoch nicht für andere atomic blocks, sichtbar.

Wir unterscheiden grundsätzlich zwischen zwei Ansätzen um transactional memory zu implementieren: Software-based und Hardware-based (und natürlich hybride Ansätze). Aktuell sind Implementierungen von TM noch nicht ausgereift und ein aktives Forschungsfeld.

15.1 Hardware Transactional Memory (HTM)

Bis heute ist es keinem Hersteller gelungen eine performante Implementierung zu produzieren. Intel hat es mit der Haswell Architektur jedoch versucht. Es gab auch andere versuche anderer Hersteller, welche es jedoch nie zur Markteinführung geschafft haben.

Auf Assembly Ebene sind die Instruktionen bei Haswell ‘xbegin‘ (begin transaction), ‘xend‘ (end transaction) und ‘xabort‘ (abort transaction).

```
1 xbegin L0
2 <transaction code>
3 xend
4 <commit was successful>
5 ...
6 L0:
7 <transaction aborted>
```

15.2 Software Transactional Memory (STM)

STM ist weiter verbreitet verfügbar, häufig in Form von Bibliotheken paralleler Programmiersprachen. Die Performance meist sehr begrenzt. Der triviale Ansatz zur Implementierung ist ein global lock zu verwenden. Dass dies nicht getan wird sollte nun offensichtlich sein.

Wir betrachten nun clock-based STM (die clock ist ein counter), welches Operationen eines atomic blocks vermerkt, auf Inkonsistenten reagiert, und beim commit testet, ob atomicity und isolation garantiert werden können.

Jede transaction hat ein lokales read-set und ein lokales write-set, welches alle gelesenen und geschriebenen Objekte enthält.

- read ein Objekt O
  - falls O im read-set: lese das Objekt
  - falls O nicht im read-set: teste, ob die timestamp $\leq$ transaction birthdate
    - nein: exception/abort
    - ja: Kopie des Objektes zum read-set hinzufügen

- write ein Objekt O
  - falls O im write-set: verwende das Objekt
  - falls O nicht im write-set: füge eine Kopie zum write-set hinzu und verwende diese
- commit
  - locke Objekte des read-set und write-set (entsprechend order, sodass kein deadlock)
  - falls nicht alle Objekte time stamp $\leq$ transaction birthdate: abort
  - increment und speichere Wert der clock zwischen
  - kopiere Elemente des write sets in den globalen Speicher mit aktualisiertem timestamp T
  - gebe alle locks frei
  - return 'commited'

15.3 Java STM ('scalar-stm')

Wir arbeiten in Java mit der 'scalar-stm' Library, einer Implementierung von STM, da Java keine direkte Unterstützung für transactional memory hat. Dies ist eine reference-based STM Implementierung. Durch den Aufbau von Java gibt es einige Limitierungen:

- jede transaction muss als 'Runnable' oder 'Callable' Object/Lambda deklariert werden
- es existiert kein Compiler-Check für die korrekte Verwendung von STM Referenzen

Nun folgt das Account-Transfer Beispiel.

```
1  class AccountSTM {
2    private final Integer id; // account id
3    private final Ref.View<Integer> balance;
4
5    AccountSTM(int id, int balance) {
6      this.id = new Integer(id);
7      this.balance = STM.newRef(balance);
8    }
9
10   void withdraw(final int amount) {
11     // assume that there are always sufficient funds...
12     STM.atomic(new Runnable() { public void run() {
13       int old_val = balance.get();
14       balance.set(old_val - amount);
15     }});
16   }
17
18   void deposit(final int amount) {
19     STM.atomic(new Runnable() { public void run() {
20       int old_val = balance.get();
21       balance.set(old_val + amount);
22     }});
23   }
24
```

```
25    public int getBalance() {
26      int result = STM.atomic( new Callable<Integer>() { public
            Integer call() {
27        int result = balance.get();
28        return result;
29      }});
30      return result;
31    }
32
33    static void transfer_retry(final AccountSTM a, final AccountSTM b
          , final int amount) {
34      STM.atomic(new Runnable() { public void run() {
35        if (a.balance.get() < amount)
36          STM.retry();
37        a.withdraw(amount);
38        b.deposit(amount);
39      }});
40    }
41  }
```

Hier haben wir 'STM.retry()' verwendet. Dies nutzen wir als Ersatz für die condition variables von locks. Der retry wird ausgeführt, nachdem es eine Veränderung von Variablen im bis zum retry erzeugten read-/write-set gibt.

15.4 Beispiel - Dining Philosophers

Die Dining Philosophers sind ein klassisches Beispiel-Problem, welches normalerweise verwendet wird, um Schwächen von Locks, die Gefahren von Deadlocks etc. aufzuzeigen. Mit TM wird dieses Problem für den Programmierer deutlich einfacher.

Das Problem beschreibt 5 Philosophen mit 5 Gabeln, welche an einem runden Tisch sitzen. Jeder Philosoph benötigt zwei Gabeln zum Essen. Gabeln können nicht gleichzeitig von zwei Philosophen verwendet werden. Die Schwierigkeit besteht darin, deadlocks zu vermeiden. Da die Struktur des Problems grundsätzlich zyklisch ist, kann nur schwer eine faire globale Ordnung eingeführt werden.

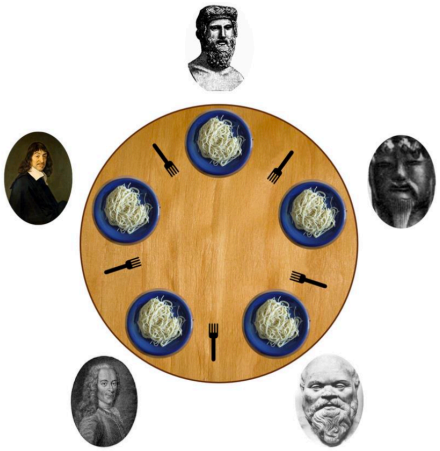


Figure 11: Dining Philosophers Visualisierung

Es bietet sich allerdings eine relative einfache STM Implementierung an:

```
1  private static class Fork {
2    public final Ref.View<Boolean> inUse = STM.newRef(false);
3  }
4
5  class PhilosopherThread extends Thread {
6    private final int meals;
7    private final Fork left;
8    private final Fork right;
9
10   public PhilosopherThread(Fork left, Fork right) {
11     this.left = left;
12     this.right = right;
13   }
14
15   public void run() {
16     for (int m = 0; m < meals; m++) {
17       // THINK
18       pickUpBothForks();
19       // EAT
20       putDownForks();
21     }
22   }
23
24   private void pickUpBothForks() {
25     STM.atomic(new Runnable() { public void run() {
26       if (left.inUse.get() || right.inUse.get())
27         STM.retry();
28       left.inUse.set(true);
29       right.inUse.set(true);
30     }});
31   }
32
33   private void putDownForks() {
34     STM.atomic(new Runnable() { public void run() {
35       left.inUse.set(false);
36       right.inUse.set(false);
37     }});
38   }
39
40   public static void main(String[] args) {
41     Fork[] forks = new Fork[tableSize];
42
43     for (int i = 0; i < tableSize; i++)
44       forks[i] = new Fork();
45
46     PhilosopherThread[] threads = new PhilosopherThread[tableSize];
47
48     for (int i = 0; i < tableSize; i++)
49       threads[i] = new PhilosopherThread(forks[i], forks[(i + 1) %
             tableSize]);
50   }
51 }
```

16 Theoretical Concepts

Bisher haben wir high-level Konstrukte und Implementierungen betrachtet, um parallele Programme schreiben zu können. Jetzt werden wir die zugrunde liegende Theorie betrachten.

In der sequentiellen Programmierung werden Objekte als eine Menge von Daten und Methoden betrachtet, welche die Daten auf eine bestimmte Weise manipulieren. Man kann Hoare Logik verwenden, um über die Korrektheit eines sequentiellen Programms zu argumentieren, indem man pre- und post-conditions berücksichtigt. Dieser Ansatz ist fundamental sequentiell, da davon ausgegangen wird, dass die Methoden zu einem einzigen Zeitpunkt wirksam werden.

Für die gleichzeitige Analyse müssen wir den Begriff des Methodenaufrufs erweitern: Ein Methodenaufruf ist das Intervall, das mit dem Aufruf beginnt und mit einer Antwort endet. Zwischen Aufruf und Antwort wird eine Methode als "pending" bezeichnet.

sequentiell	concurrent
relevanter Zustand nur zwischen Methodenaufrufen	Methoden können überlappen, Objekte sind ggf. nie 'zwischen' Methodenaufrufen
Methoden werden isoliert betrachtet	alle möglichen Interaktionen zwischen concurrent Aufrufen müssen betrachtet werden
neue Methoden können ohne Betrachtung alter Methoden hinzugefügt werden	muss berücksichtigen, dass alles mit allem interagieren kann
'global' lock	'object/thread' lock

Figure 12: Vergleich von sequentiellen und concurrent Programmen

Wir führen noch ein paar weitere Begriffe ein. hisotry beschreibt eine Ausführung. Wir separieren einen Methodenaufruf in invocation (A q.enq(x), (thread, object, method, arguments)) und response (A q: void, (thread, object, result)).

- Invocation und response match wenn Thread- und Objektnamen passen.
- Eine invocation is pending, falls es keine matching response gibt.
- Eine subhistory ist complete, falls es keine pending responses gibt.
- Projections betrachten eine Teilmenge aller invocations und responses einer history entsprechend einem Kriterium, welches entweder ein Objekt oder ein Thread sein kann. ($H|q$, projection von H auf q)
- Sequential histories sind histories ohne interleavings von Methodenaufrufen unterschiedlicher Threads, wobei eine pending invocation am Ende erlaubt ist.
- Well formed histories sind histories, in welchen alle thread projections sequential sind.
- Equivalent histories are histories, bei welchen die Projektionen auf die jeweiligen Threads identisch sind.

- Legal histories sind sequentielle histories bei welchen die Projektion eines jeden Objektes der sequentiellen Spezifikation dieses Objektes genügt.
- Eine Methodenaufruf preceeds einen anderen Methodenaufruf falls das response event dem invocation event vorausgeht. Gibt es keine precedence überlappen die Methodenaufrufe. ($m_0 \rightarrow_H m_1$, $\rightarrow_h$ partial order auf history H, total wenn H sequentiell ist)

16.1 Linearizability/Linearisierbarkeit

Linearizability formalisiert das Konzept von locks, also das Zurückführen paralleler Ausführungen auf sequentielle Teile. Jede Methode nimmt hier zu einem Punkt zwischen Aufruf und Rückgabe Effekt. Hält dies für alle möglichen Ausführungen eines Objektes wird es als linearziable bezeichnet.

Formal: Eine history H ist linearizable falls sie durch (a) Hinzufügen von beliebig vielen responses für pending invocations, welche Effekt haben, und (b) Verwerfen von beliebig vielen invocations, welche keinen Effekt haben, zu einer history G ergänzt werden kann und G dann equivalent zu einer legal sequential history S mit $\rightarrow_G \subset \rightarrow_S$ ist.

Häufig betrachten wir jedoch nicht alle möglichen Ausführungen sondern nur eine Ausführung und möchten überprüfen, ob diese linearizability erlaubt/dieser widerspricht. Wir zeigen linearizability indem wir in einer history linearization points/Linearisierungspunkte setzen.

Atomare register sind Speicherorte für primitive Typen, welche an einem Punkt linearizable sind. Bei gelockten Methoden ist der linearization point normalerweise beim unlock. Falls eine Methode nicht gelockt ist, finden sich die linearization points dort, wo die Methode für externe Betrachter (final) Effekt nimmt.

Ein wichtiges Theorem in Bezug zu linearizability ist das Composability Theorem: Eine history H ist linearizable genau dann wenn für jedes Objekt x, $H|x$ linearizable ist. Entsprechend kann linearizability von Objekten unabhängig und in Isolation bewiesen werden, woraus linearizability von Kombinationen dieser folgt.

16.2 Sequential Consistency

Formal: Eine history H ist linearizable falls sie durch (a) Hinzufügen von beliebig vielen responses für pending invocations, welche Effekt haben, und (b) Verwerfen von beliebig vielen invocations, welche keinen Effekt haben, zu einer history G ergänzt werden kann und G dann equivalent zu einer legal sequential history S ist.

Entsprechend ist sequential consistency schwächer als linearizability und wird von dieser impliziert. Man muss jedoch beachten, dass hier composability NICHT gilt.

Hier wird überprüft, ob bei einer Ausführung die Operationen eines Threads die program order (PO) respektieren, man jedoch die real-time order bzw. die Relation zwischen den Threads verändern kann. Dieses Modell wird häufiger in concurrent Situationen zur Beschreibung von multiprocessor Architekturen verwendet. Z. B. ist das Peterson lock sequential consistent.

16.3 Quiescent Consistency

Dies soll als Beispiel für weitere Consistency Modelle dienen. Dies approximiert noch näher was hardware wirklich garantiert, wenn man nur einfache register betrachtet. Die Idee ist, dass die real-time order gültig sein muss, falls die Betrachtung Perioden von quiescence überschreitet. Andernfalls machen wir keinerlei Annahmen.

Sequential consistency und quiescent consistency implizieren sich nicht gegenseitig.

16.4 Vergleich der Modelle

Auf Hardware-Ebene ist es üblicherweise zu teuer sequential consistency durchgängig zu garantieren. Für das Peterson Lock ist dies jedoch notwendig. Deswegen bieten viele Programmiersprachen/Umgebungen an, dass man durch explizite Annotation ein Verhalten ähnlich zu sequential consistency erhält. In Java ist ein solches keyword 'volatile', wie bereits verwendet. Auch locking durch 'synchronized' hat einen solchen Effekt. Zudem fügt der compiler manchmal implizit Synchronisationen ein.

16.5 Consensus

Consensus ermöglicht uns den Vergleich der Mächtigkeit verschiedener Synchronisierungsoperationen wie CAS. Consensus an sich ist stark - wenn eine Datenstruktur Consensus implementiert bedeutet dies, dass man mit dieser z. B. aus CAS implementieren kann.

Fundamental ist consensus ein Interface:

```
1 public interface Consensus<T> {
2   T decide (T value);
3 }
```

Die Anforderungen an 'decide(...)', welches von einer Anzahl von Threads mit individuellen Input-Werten aufgerufen wird, sind:

- wait-free: returns in endlicher Zeit für jeden Thread
- consistent: alle Threads erhalten den gleichen Rückgabewert
- valid: der Rückgabewert muss der Input eines Threads sein

Dies impliziert, dass der Wert des ersten Threads für alle Threads als Rückgabewert gewählt werden muss.

Zusätzlich definieren wir die consensus number. Eine Klasse implementiert n-thread consensus falls es ein consensus protocol für n Threads auf Basis beliebig vieler Instances der Klasse und beliebig vielen atomaren registern definiert. Die consensus number C ist das größte n, sodass C n-thread consensus implementiert/löst.

Theorem: Atomare register haben consensus number 1.

Corollary: Es existiert keine wait-free Implementierung von n-thread consensus mit $n \geq 2$ nur auf Grundlage von read-write registern.

Compare-and-Swap/CAS hat consensus number Unendlich. Dies kann am einfachsten bewiesen werden, indem eine gültige Implementation angegeben wird.

```
1  class CASConsensus {
2    private final int FIRST = -1;
3    private AtomicInteger r = new AtomicInteger(FIRST); // supports
         CAS private
4    AtomicIntegerArray proposed; // suffices to be atomic register
5    ... // constructor (allocate array proposed etc.)
6    public Object decide (Object value) {
7      int i = ThreadID.get();
8      proposed.set(i, value);
9      if (r.compareAndSet(FIRST, i)) // I won
10       return proposed.get(i); // = value
11     else
```

```
12          return proposed.get(r.get());
13      }
14  }
```

Das Array könnte hierbei auch noch 'eliminiert' werden, falls wir einen Wert einer Variable zum FIRST state zuordnen können, i.e., garantieren können, dass dieser State nie als Parameter and decide übergeben wird.

FIFO queues können genutzt werden, um 2 thread consensus zu implementieren. Dazu wird ein Array der Länge zwei benötigt, in welchem jeder Thread seinen Wert speichert. Und eine FIFO queue, welche zwei distinke Elemente enthält. Ein Thread schreibt bei Aufruf von 'decide' zuerst seinen Wert ins array und nimmt danach den ersten Wert von der FIFO queue. Ist dies erst erste distinkte Wert so wird der eigene Wert zurückgegeben, andernfalls wird der Wert des anderen Threads zurückgegeben.

Daraus lernen wir: Es existiert keine wait-free Implementierung einer FIFO queue nur mit atomaren registern.

Die consensus number ist hilfreich, da sie Aussagen darüber erlaubt, was womit implementiert werden kann. Wir wissen z. B., dass

- wait-free FIFO: 2
- Test-and-Set (TAS), Get-and-Set, Get-and-Increment: 2
- CAS: ∞
- atomic register: 1

Man kann beweise, dass kein Algorithmus existiert, welcher consensus für $n \geq 3$ implementiert ohne auch consensus für eine unendliche Anzahl von Threads zu lösen.

17 Distributed Memory & Message Passing

Praktisch alle Probleme beim parallel programming stammen von geteilten Daten. Dies ist die Ursache für race conditions, die Komplexität von locks, ... Der Ansatz von Distributed Memory und Message Passing ist es solche geteilten Zustände komplett zu vermeiden.

Eine Möglichkeit dies umzusetzen ist functional programming. Dies wird in anderen Vorlesungen thematisiert. Hier betrachten wir Message Passing. Jeder Thread hat seinen privaten mutable state. Threads arbeiten über das Austauschen von Nachrichten zusammen, nicht über das Teilen von Speicher.

Das Senden von Nachrichten kann synchronous oder asynchronous geschehen:

- synchronous

  - send: Der sender blockiert bis die Nachricht erhalten wurde.

  - receive: Der Empfänger wartet bis die erwartete Nachricht empfangen wurde.

- asynchronous

  - send: Der Sender wartet nicht bis die Aktion beendet um zu returnen (fire-and-forget). Die Nachricht wird in einen Buffer für den Empfänger geschrieben.

Synchronous Methoden können verwendet werden, um Daten zu transferieren oder Prozesse zu synchronisieren. Allgemein wird jedoch asynchronous Kommunikation verwendet, um lokalen Fortschritt zu ermöglichen.

Für das Senden von Nachrichten unterscheiden wir ferner zwischen blocking und non-blocking. Das Empfangen von Nachrichten ist immer non-blocking.

- blocking

  Eine Methode terminiert, nachdem der lokale Teil des Sendevorgangs abgeschlossen ist. Der globale Teil der Übertragung ist möglicherweise noch nicht beendet.

- non-blocking

  Die Methode terminiert sofort. Dazu müssen wir einen Buffer mit den zu sendenden Daten übergeben und garantieren, dass dieser nicht verändert wird, bis der Sendevorgang abgeschlossen ist.

Der Unterschied zwischen (a)synchronous und (non-)blocking ist, dass (a)synchronous die Kommunikation zwischen Sender und Empfänger beschreibt, während (non-)blocking das lokale Data-Handling beim Senden beschreibt.

Message Passing wird von verschiedensten Frameworks/Interfaces/APIs implementiert, darunter CSP (Communicating Sequential Process), Actor programming model (GO Programmiersprache), und MPI (Message Passing Interface), welches wir hier verwendet werden. MPI ist der Nachfolger von PVM (Parallel Virtual Machines) und existiert seit den 1980ern.

MPI ist eine Standard API, welche als Library für viele Programmiersprachen implementiert ist. Als solche ist MPI sehr flexible und portable: MPI-Code kann in einer Vielzahl von Laufzeitumgebungen ausgeführt werden (high-performance cluster, general-purpose computers, ...). MPI ist der de-facto Standard für distributed parallel computing mit fast 100 % Marktanteil.

Bei MPI werden immer processes betrachtet. Diese sind in groups sortiert, welchen verschiedene communicators zugewiesen sein können. Wir sagen, dass eine group aus mehreren colors besteht. Die Kombination aus group und color identifiziert einen communicator. Beim Starten eines MPI Programms gibt es zu Beginn den 'MPI_COMM_WORLD' mit allen Prozessen. Für größere und modulare Anwendungen ist es bad-practice den Standard communicator zu nutzen. Ein communicator ist doe communication domain von MPI processes. Innerhalb eines communicators ist Kommunikation zwischen den enthaltenen Prozessen erlaubt. Innerhalb eines communicators enthält ein process einen eindeutigen rank/identifier.

MPI folgt dem SPMD (single program multiple data) Paradigm - hier eigentlich besser: single program multiple instances, da auf gleichen Daten gearbeitet werden kann. Es wird also in jedem Prozess das gleiche Programm ausgeführt. Abhängig vom rank des Programms wird dann differenziert, was in diesem Prozess ausgeführt wird.

```
1 public static void main(String args []) throws Exception { MPI.Init
     (args);
2   // Get total number of processes (p)
3   int size = MPI.COMM_WORLD.Size();
4   // Get rank of current process (in [0..p-1])
5   int rank = MPI.COMM_WORLD.Rank();
6   MPI.Finalize();
7 }
```

In der Standard-Konfiguration ist das Senden blocking und synchronization hängt von der Implementierung ab. Die Kombination von blocking und synchronous Kommunikation kann zu deadlocks führen, falls große Datenmengen transferiert werden und der Buffer des Empfängers nicht ausreichend groß ist.

Um jedes MPI Programm zu schreiben reichen sechs von MPI definiert Funktionen aus:

- MPI_INIT - initialize the MPI library (must be the first routine called)
- MPIT_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 - send a message to another process
- MPI_FINALIZE - clean up all MPI state (must be the last MPI function called by a process)

17.1 Communication

Hier betrachten wir point-to-point communication.

```
1 Comm.Send(
2   Object buf, // pointer to the array of items
3   int offset,
4   int count, // number of items to be sent
5   Datatype datatype, //  data type of items
6   int dest, // destination process id/rank
7   int tag // data id tag
8 )
```

Der Tag kann verwendet werden um innerhalb eines communicators zwischen verschiedenen Nachrichten(-Typen) zu unterschieden.

```
1 void Comm.Recv(
2   Object buf, // pointer to the buffer to receive to
3   int offset,
4   int count, // number of items to be received
5   Datatype datatype, // data type of items
6   int src, // source process id/rank or MPI_ANY_SOURCE
7   int tag // data id tag or MPI_ANY_TAG
8 )
```

17.2 Collective Communication

17.2.1 Broadcast

$\mathcal{O}(n)$ work und $\mathcal{O}(n \log n)$ levels/time.

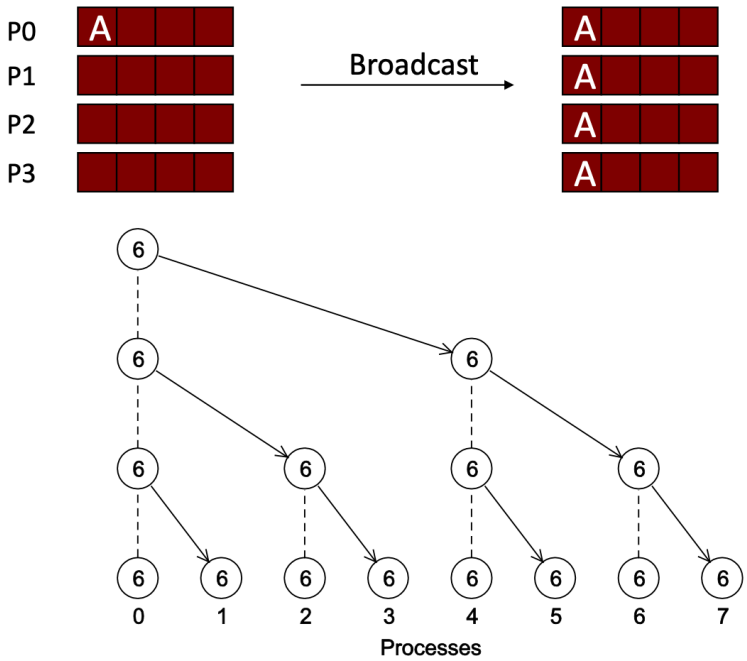


Figure 13: Broadcast

17.2.2 Gather

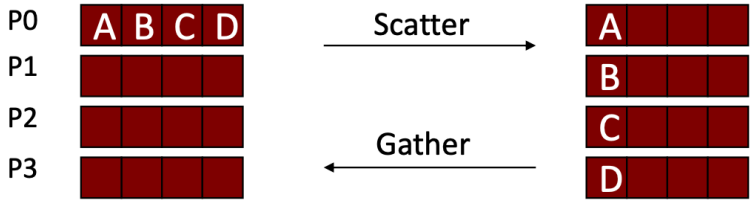


Figure 14: Gather & Scatter

Gather nimmt einen Wert von jedem Prozess in einem Communicator und "sammelt" sie bei einem Prozess. Zum Beispiel sammelt gather alle Komponenten des Vektors im Zielprozess, der dann alle Komponenten verarbeiten kann.

17.2.3 Scatter

Scatter hat eine Reihe von Werten, die auf alle Prozesse in einem Kommunikator verteilt werden. Die Anzahl der zu verteilenden Elemente sollte der Größe des Kommunikators entsprechen.

Scatter kann z.B. in einer Funktion verwendet werden, die einen gesamten Vektor auf Prozess 0 einliest, aber nur die benötigten Komponenten an jeden der anderen Prozesse sendet.

17.2.4 Reduce

$\mathcal{O}(n)$ work und $\mathcal{O}(n \log n)$ levels/time.

```
1 public void Reduce(java.lang.Object sendbuf,
2   int sendoffset,
3   java.lang.Object recvbuf,
4   int recvoffset,
5   int count,
6   Datatype datatype,
7   Op op,
8   int root)
```

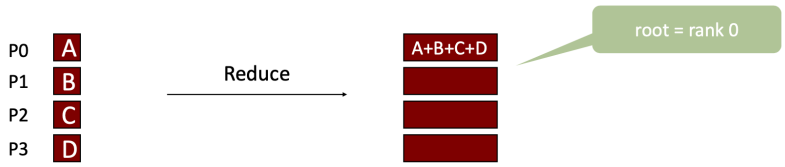


Figure 15: Reduce

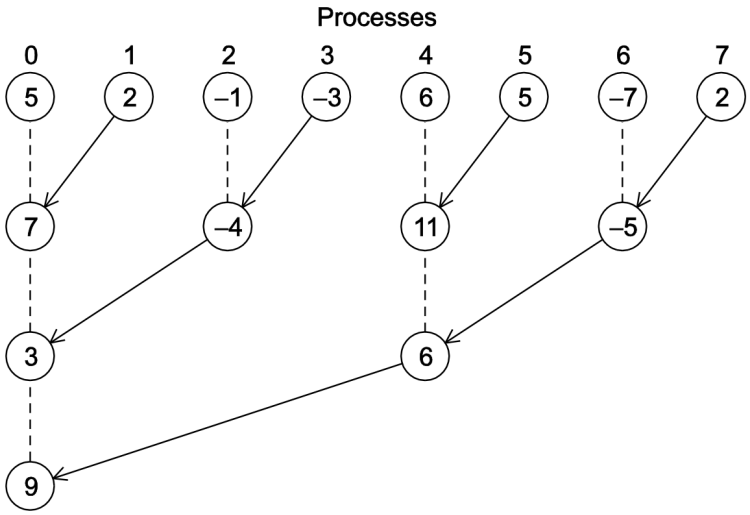


Figure 16: Reduce Implementierung

17.2.5 Scan

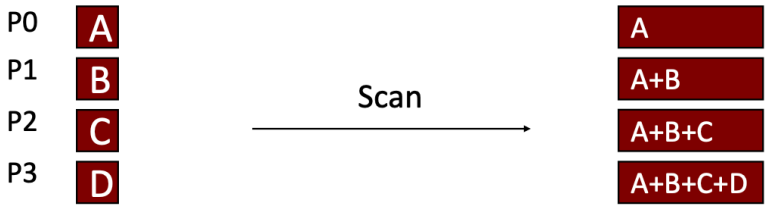


Figure 17: Scan

17.2.6 Allreduce

$\mathcal{O}(n)$ work und $\mathcal{O}(n \log n)$ levels/time mit konstantem Faktor 2.

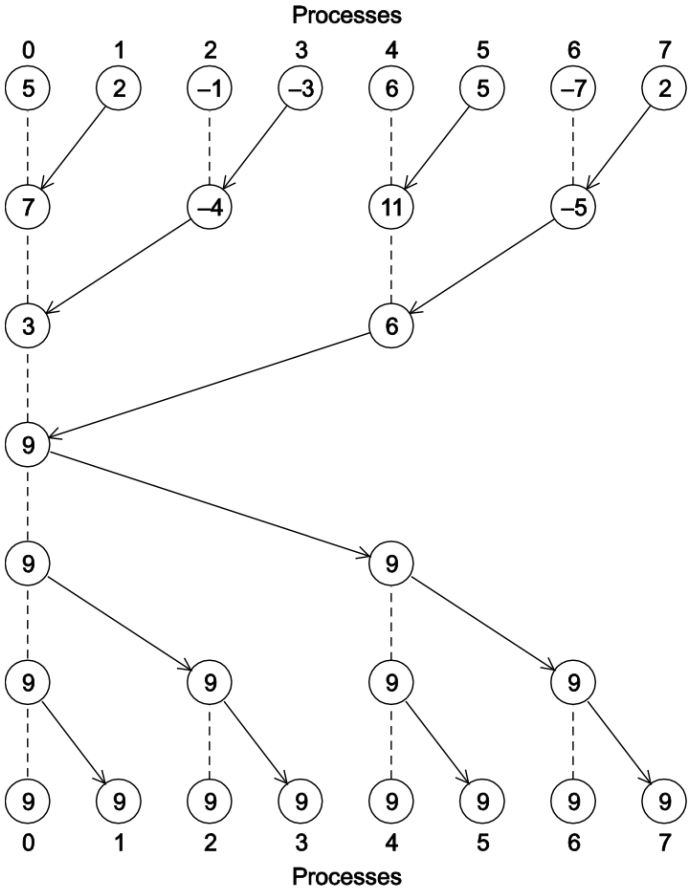


Figure 18: Allreduce als reduce + broadcast

Durch eine Optimierung kann der konstante Faktor entfernt werden.

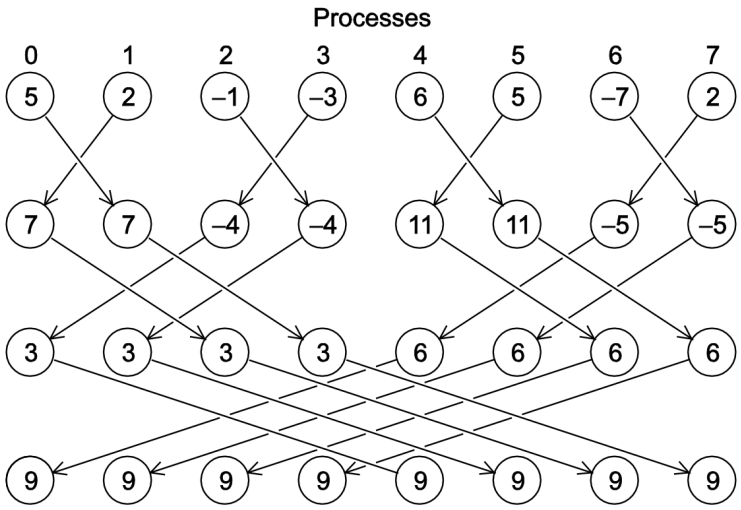


Figure 19: Allreduce Butterfly

17.2.7 Allgather

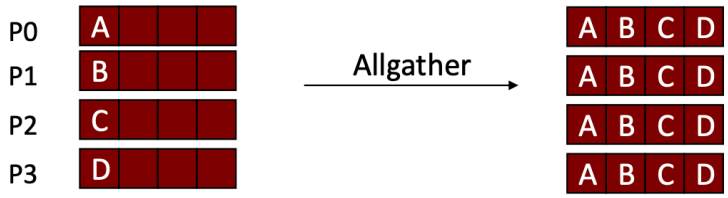


Figure 20: Allgather

17.2.8 Alltoall

Figure 21: AlltoAll

17.2.9 Barrier

...

17.3 Beispiele

17.3.1 Parallel Sorting

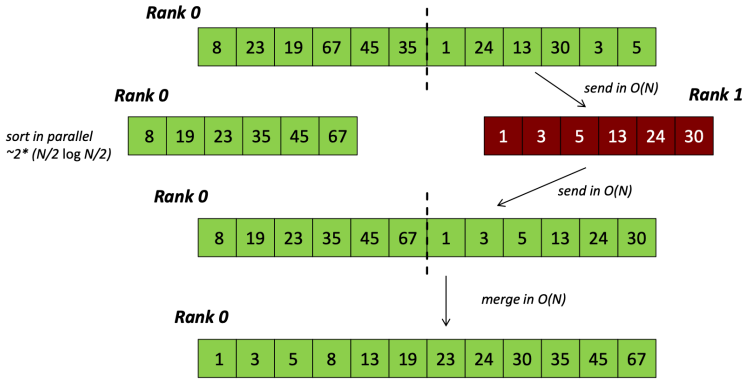


Figure 22: Parallel Sorting mit MPI send/receive

17.3.2 Pi berechnen

Um Pi zu approximieren kann folgende Formel genutzt werden: $\pi \approx h \sum_{i=0}^{N-1} \frac{4}{1+(h(i+\frac{1}{2}))^2}$.

Für eine gewisse Approximierung ergibt sich sequentiell:

```
1 for(int i=0; i<numSteps; i++) {
2   double x = (i + 0.5) * h;
3   sum += 4.0/(1.0 + x*x);
4 }
5 double pi = h * sum;
```

Mit MPI ergibt sich folgende Optimierung:

```
1  MPI.Init(args);
2  ... // declare and initialize variables (sum=0 etc.)
3  int size = MPI.COMM_WORLD.Size();
4  int rank = MPI.COMM_WORLD.Rank();
5
6  for(int i=rank; i<numSteps; i=i+size) {
7    double x=(i + 0.5) * h;
8    sum += 4.0/(1.0 + x*x);
9  }
10
11 if (rank != 0) {
12   double [] sendBuf = new double []{sum};
13   // 1-element array containing sum
14   MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 10);
15 } else { // rank == 0
16   double [] recvBuf = new double [1] ;
17   for (int src=1 ; src<P; src++) {
18     MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 10);
19     sum += recvBuf[0];
20   }
21 }
22 double pi = h * sum; // output pi at rank 0 only!
23 MPI.Finalize();
```

17.3.3 Matrix-Vector-Multiply

Diest berechnet beispielhaft $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$.

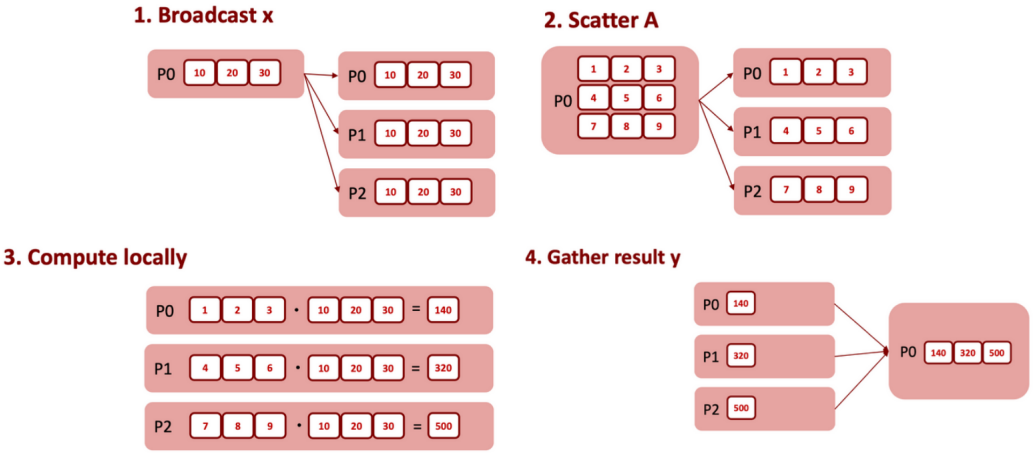


Figure 23: Matrix-Fector-Multiply