Introduction

0.1 Moore's Law

Moore's Law says that the number of transistors on a CPU is doubling every 1-2 years. The problem nowadays is that the clockspeed has reached its limit at approximately 5GHz and the curve is flattening. Because the clockspeed of one core cannot be increased any further the CPU manufacturers started to increase the number of cores on one processor. These multicore processors increase the CPU performance despite the following *walls*:

- MEMORY WALL:
  Describes the gap betwenn the CPU clock speed and the memory acces time.
- ILP WALL:
  There are not enough instruction-level parallelism to keep the CPU busy.
- POWER WALL:
  The higher the clock speed the higher the power consumption is.

Therefore the performance of any single-threaded machine is not improving anymore - the new standard is multicore.

0.2 Multicores

Multicore processors are nowadays used in every desktop PC, laptops and servers.
But the existence of multicore does not lead to the expected speedup, because the usage requires a lot care for parallelization and synchronization which affects the speedup negatively. To programm a correct and efficient application we need to exploit parallelism. Furthermore we need synchronization to prevent race-conditions, deadlocks, and so on and ensure correctness but still must have an eye on the performance.

0.3 Parallelization

0.3.1 Data Parallelism

The data can be split into partitions of which every thread processes one of them. After finishing the work they join the results. This approach is used especially in image processing.
The proble is that good data partitioning is difficult, i.e. that every thread is computing approximately the same amount of time. Also synchronization is required for joining the results and the load balancing during the process. Another issue could be if some results are dependant on other results which had to be computed before.

0.3.2 Pipeline Parallelism

The approach here is to split the work into different phases and every thread works on jobs in one of the phases. The problem is that not every program has or cannot be divide in such phases and some phases can be much longer than others. Therefore a high amount of synchronization is required because one phase's output becomes the next phase's input which also makes the access pattern really complex.

0.3.3 Speculative/Optimistic Parallelism

In this approach we assume that most jobs do not conflict and therefore just execute them. Therefore partitioning or phases are not neede but can be implemented as well. But in the applicaiton there must be any kind of synchronization because every access can potentially interfere with another thread. To get the most out of the application there must ba as little contention on shared data structures as possible.

0.4 Shared Memory Synchronization

The cores share a main memory, which contain some atomic hardware instructions like *increase, decrease, CompareAndChange*. Also the load and store instructions are usually atomic, but not necessary ordered - not sequential consistent. Therefore memory barriers must be used to enforce the "right" order.

1 Basics

1.1 Formalizing a Problem

There are two types of formal properties in asynchronous computation:

- SAFETY PROPERTY
  Nothing bad ever happens (for example mutual exclusion is a safety property).
- LIVENESS PROPERTY
  Something good happens eventually (for example no deadlocks is a liveness property).

1.2 Amdahl's Law

The speedup of any computation given n CPUs instead of 1 can be comuputed with the following formula:

$$Speedup = \frac{OldExecutionTime}{NewExecutionTime}$$

Furthermore we can normalize this formula and implement the number of processes and the concurrent and non-concurrent part of the code to compute the expected speedup:

$$Speedup = \frac{1}{1 - p + \frac{p}{n}}$$

,where p is the procentual share of the concurrent code and n is the number of processes used.

When analyzing the the graph when we scale the percentage share of the sequential code and slowly decrease it the expected speedup does not increase linearly but increases really slowly. Therefore when programming for multicore usage the most important thing is to find ways to effectively parallelize the code by miimizing the sequential parts and reduce any wait time in which threads are idle. All in all the percentage which cannot or only hardly make concurrent will have a great impact on the overall speedup.

2 Mutex

2.1 Partial Orders

- IRREFLEXIVE
  It is never true that $A \to A$
- ANTISYMMETRIC
  If $A \to B$ then it is not true that $B \to A$
- TRANSITIVE
  If $A \to B$ and $B \to C$ then $A \to C$

2.2 Total Orders

- same as **Partial Orders**
- ADDITIONAL: For every distinct A, B
  Either $A \to B$ or $B \to A$

2.3 Different Implementations

2.3.1 Peterson's Algorithm

In the Peterson's Algorithm a thread that wants to acquire a lock first announces its interest via a flag or something similar and sets itself as a victim. As long as other threads are interested and it is the victim, it will wait. When releasing the lock it will set its flag to false to announce that it is not interested anymore.
This algorithm fulfills mutual exclusion such that only one thread can be inside the critical section. Furthermore it is deadlock free because a thread is only blocked when in the *while*-loop which happens when it is the victim. But because one or another thread must not be the victim the scenario of both waiting can be excluded.

2.3.2 Filter Algorithm

In the Filter Algorithm there are $n - 1$ *waiting rooms* called levels. Each level can be entered by at least one thread and at least one is blocked if man tried to enter. In the end only one thread makes it through the filter and enters the critical section.
First a thread announces its intention to enter a distinct level L. Itself gives priority to anyone else by setting its status to be the victim. As long someone else is at the same or any higher level and it is still the victim the thread waits. When the loop resolves it enters the level L. When unlocking the lock its level is reset to the default value 0.

2.3.3 Bakery Algorithm

The Bakery Algorithm provides a First-Come-First-Serve behaviour by giving each thread a number, like in a queue at the supermarket, and waits until each thread with a lower number has been served.
This algorithm provides property as no-deadlock, mutual exclusion and so on, but it is not practical because every time n distinct variables must be read which leads to a significant loss of performance.

3 Linearizability

3.1 Sequential vs. Concurrent