FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Teaching Multi-Core Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister

im Rahmen des Studiums

Informatikmanagement

eingereicht von

DI Matthias Wenzl
Matrikelnummer 0425388

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ass.Prof. Dr. Monika DiAngelo

Wien, 12.11.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Teaching Multi-Core Programming

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Magister

in

Computer Science Management

by

DI Matthias Wenzl
Registration Number 0425388

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ass.Prof. Dr. Monika DiAngelo

Vienna, 12.11.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

DI Matthias Wenzl
Elisabethallee 39/8, 1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First, I would like to thank Ass.Prof. Dr. Monika Di Angelo for her valuable feedback and remarks. Moreover, I would like to thank my wife for her cherished support and understanding.

This thesis has been partly supported by the City of Vienna, MA23 under the project grant number MA23-Projekt 10-09.

Abstract

During the last decades processor manufactures mainly increased performance by shrinking their production size and increasing the processor's clock speed. However, due to physical limitations this approach is not efficient any more. As a consequence, the computer industry increased the number of processor cores on a single die, thus creating multi-core processors. Up to today multi-core processors are used in all kinds of computer systems ranging from super computers to mobile phones. However, programming multi-core systems is a challenging task, which requires software engineers to restructure the way they think about computer programs.

Therefore, this thesis presents a best practice based approach, introducing two course concepts on parallel programming. The first course is a more guided one, where all students are obliged to focus on the same assessment types within the course. Moreover, this course has a strong practical focus, thus being aimed at a university of applied science context. The second course offers more theory and a greater freedom to the students as they are free to choose a certain topic within the field of multi-core systems. This course is aimed at a university context.

The best-practice course concepts are obtained through a thorough analysis of numerous research articles with the following focus:

- The identification and proposition of solutions on the issues and pitfalls when teaching multi-core programming.
- The recommendation of suitable assessment methods when dealing with this challenging field.
- A set of related courses already teaching parallel programming in various aspects.

As a consequence two best-practice based course concepts tailored to the needs of a university and a university of applied sciences are derived. Based on these best-practice examples, two actual courses are designed. This includes a detailed course schedule, learning outcome, assessment concepts, and a set of recommended research articles on the theory of concurrent systems.

Kurzfassung

Eine Kombination von Erhöhung der Taktrate und Verringerung der Strukturbreiten war in den letzten Jahrzehnten ein gängiges Mittel zur Geschwindigkeitssteigerung von Mikroporzessoren. Aufgrund physikalischer Beschränkungen ist derzeit mit diesem Ansatz jedoch kein nennenswerter Geschwindigkeitszuwachs mehr zu erzielen. Eine naheliegende Lösung war die Entwicklung sogenannter Mehrkernprozessorsysteme, welche mehrere Prozessorkerne auf einem Chip zur Verfügung stellen und somit wieder zu einer Rechenleistungssteigerung führten. Heute sind Mehrkernprozessorsysteme bereits allgegenwärtig und in diversen Computersystemen, vom Supercomputer bis zum Mobiltelefon, im Einsatz. Allerdings ist die Programmierung von Mehrkernprozessorsystemen eine herausfordernde Aufgabe, welche von angehenden Ingenieuren verlangt, ihre Betrachtungs- und Herangehensweise im Bereich der Softwareentwicklung grundlegend zu erweitern. Aus diesem Grund stellt diese Diplomarbeit zwei Lehrveranstaltungsdesigns basierend auf einer ”Best-Practice”- Evaluierung vor. Der erste Kurs wird hierbei an die Anforderungen eines Universitätskurses angepasst, während der zweite Kurs für die Anforderungen an einer Fachhochschule optimiert ist. Die Vorgehensweise zur Erreichung dieser Ziele ist hierbei wie folgt:

- Die Identifikation von Herausforderungen und deren Lösungen im Unterrichten von parallelen Programmiertechniken und deren zugrunde liegender Theorie.
- Eine Analyse geeigneter Methoden zur Leistungsprüfung im Hinblick auf die unterschiedlichen Kursausprägungen.
- Eine Gegenüberstellung diverser Kurse im Bereich parallelen Rechnens auf Basis ihrer Lehrziele.

Aufbauend auf diesen Erkenntnisse und den daraus resultierenden ”Best-Practcie”- Empfehlungen werden die tatsächlichen Lehrveranstaltungen für ein universitäres, sowie für ein Fachhochschulumfeld abgeleitet. Die praktischen Kursbeschreibungen beinhalten einen detaillierten Zeitplan, die entsprechend ihrem Unterrichtsumfeld angepassten Lehrziele, Konzepte zur Leistungsüberprüfung sowie eine Auswahl an empfohlenen wissenschaftlichen Artikeln zu den behandelten Themen.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU Arithmetic Logic Unit

AMP Asymmetric Multi-Processor

CMP Chip Multi-Processor

CMT Chip Multi-Threading

CPU Central Processing Unit

EDA Electronic Design Automation

EU European Union

FPGA Field Programmable Gate Array

FPU Floating Point Unit

GPU Graphic Processing Unit

GUI Graphical User Interface

HT Hyper Threading

ILP Instruction Level Parallelism

IO Input/Output

IPC Inter Process Communication

ISA Instruction Set Architecture

LRU Least Recently Used

MESI Modified Exclusive Shared Invalid

MISD Multiple Instruction Single Data

MIMD	Multiple Instruction Multiple Data
MPI	Message Passing Interface
NUMA	Non Uniform Memory Access
OpenMP	Open Multi Programming
PC	Personal Computer
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multi-Processor
SMT	Simultaneous Multithreading
TLP	Thread Level Parallelism
UMA	Uniform Memory Access
UNESCO	United Nations Educational, Scientific and Cultural Organization

CHAPTER 1

Introduction

During the last decades processor manufactures mainly increased performance by shrinking their production size and increasing the processor's clock speed. However, due to physical limitations this approach is not efficient any more. As a consequence, the computer industry increased the number of processor cores on a single die, thus creating multi-core processors [104]. Up to today multi-core processors are used in all kinds of computer systems ranging from super computers to mobile phones. However, programming multi-core systems is a challenging task, which requires software engineers to restructure the way they think about computer programs [33, 59, 64].

Therefore, this thesis presents a best practice based approach, introducing two course concepts on parallel programming. The first course is a more guided one, where all students are obliged to focus on the same assessment types within the course. Moreover, this course has a strong practical focus, thus being aimed at a university of applied science context. The second course offers greater freedom to the students as they are free to choose a certain topic within the field of multi-core systems. As a consequence, this course is aimed at a university context. Nevertheless, the presented course concepts base on a thorough analysis of existing multi-core programming courses in order to derive best practices considering assessment, assignments, and general course structures with respect to desired learning outcomes. Eventually, both course designs will be applied in the summer term 2013 at the Vienna University of Applied Sciences and the Vienna University of Technology.

The remainder of this thesis is structured as follows: Chapter 2 provides a hardware and software oriented introduction on multi-core programming in general. The Chapter consists of a short overview on the history of parallel computing and a categorization of parallel computing systems according to Flynn [29]. Afterwards, hardware parallelism is examined, investigating processor architectures, memory consistency, and processor synchronization. Subsequently, Chapter 2 addresses software parallelism, introducing Amdahl's Law [6] and focusing on topics like categorizing software parallelism. Moreover, this Chapter investigates synchronization and communication issues and discusses software libraries aiding in developing parallel software.

The subsequent Chapter 3, is divided into 4 consecutive parts:

1. First, it is shown that the comparison of single courses within degree programs cannot be done on basis of the alumni qualification profile, but must be done with respect to the so called learning outcome.
2. Subsequently, numerous contributions identifying issues and challenges in teaching parallel programming in a more specific manner are reviewed.
3. Moreover, a set of dedicated parallel programming courses is evaluated with respect to their learning outcome definitions as well as their actual teaching content.
4. The result of this evaluation serves as a basis to derive two best practice courses. One is aimed at a university environment, the other one is designed to fit in a university of applied science context. These best practice course concepts will be used to design two courses in Chapter 4.

As a consequence, Chapter 4 presents the actual instances of the best practice course templates developed in the previous Chapter. The first course presented is composed for a university context. It is designed as a conference simulation which requires participants to write a seminar paper, perform a peer review, and do a final presentation with a subsequent discussion. The freedom of choice is implemented by letting students choose their general topic out of a topic pool. The second course is dedicated to a university of applied sciences context, thus offering more industry focused content. This includes covering dedicated programming libraries for all participants. Although students are required to read and understand certain research papers on the theory of parallel computing as well, the main focus of the course is the communication of the application of parallel programming techniques.

Finally, Chapter 5 concludes the thesis. Moreover, this Chapter presents the results of an evaluation on the university of applied science course that has been held in a somewhat related version in the summer term of 2013.

CHAPTER 2

Basics of Multi-Core Systems

The first computer featuring real hardware parallelism was built in 1968. At that time, Barnes et. al. [8] proposed the *ILLIAC IV* computer. It consists of 256 parallel computing elements, a unified address, and Input/Output (IO) space. The main purpose of this architecture was to aid in matrix calculations and the handling of large multi-dimensional data structures. Aside of this early attempt to increase computing power by adding extra hardware, it was not before November 2000, that chip multi-processors (CMP) were available to the public with the introduction of Intel's Pentium 4 Hyper Threading (HT) processor family [104]. By today, multi processor systems have also spread into the embedded market and are available to many people of the industrialized parts of the world. However, since 1968 numerous technologies and methods regarding parallel computing have been introduced. Therefore this chapter provides an overview of the used taxonomy in parallel computing.

2.1 Categorizing Multi-Processor Systems

The first and still most common categorization of parallel processors has been proposed by Flynn in 1972 [29]. Flynn developed a hierarchical model to represent an architecture's structure from a macroscopic point of view. To allow an abstract categorization of computer systems, Flynn used the *stream* concept, which denotes a sequence of data or instructions [30]. His approach is known as Flynn's taxonomy today. It reveals the following processor architecture classes:

Single Instruction Single Data (SISD) This kind of processor architecture consists of a single processing unit that operates with a single instruction stream on a single data stream. It does not provide any hardware parallelism at all, regarding truly parallel execution. This class refers to classic single processor architectures systems like older Personal Computers (PCs), or old main frames.

Single Instruction Multiple Data (SIMD) Systems utilizing these architecture use a single instruction stream to operate on multiple data streams. This very common type of parallelism can be found in Graphic Processing Units (GPUs), or array processors, for example. The idea behind this concept is, that it is possible to operate on one large data structure consisting of several independent sub-structures concurrently. Thus being able to finish the task faster [48].

Multiple Instruction Single Data (MISD) The third classification type by Flynn describes an architecture where multiple instruction streams, thus multiple processing units operate on a single data stream. An example for this kind of architecture is an application specific processor for pattern matching developed by Olaf René [11]. However, no commercially available processor using this kind of architecture has been built until today [40, 97].

Multiple Instruction Multiple Data (MIMD) Modern general purpose multi processor systems implement this kind of architecture where multiple instruction streams operate on multiple data streams. Examples for such systems implementing the MIMD architecture in the field of embedded systems are: [31, 32, 75, 80, 88, 96, 100–102]. Moreover, all modern server and desktop processors from Intel[1] and AMD[2] belong to this category. Furthermore, this category features clusters and super computers as well.

In order to be more specific it is also possible to break down the MIMD architecture into two sub categories [97], loosely and tightly coupled systems.

[1]www.intel.com
[2]www.amd.com

- The first sub category describes so-called *"loosely coupled"* MIMD systems. Such architectures describe systems, where the sum of available processing units does not share a common memory. They are also called *distributed memory* systems and are applied in super-computers as well as in embedded systems.

- The second disposition of systems implementing the MIMD architecture are so-called *"tightly coupled"* systems. In this case all processing units in a processor share a common accessible memory in general. Moreover, master/slave systems do also belong to this category. In a master/slave system a single master processor performs the required task scheduling, like starting and stopping jobs on a set of slave processors. Such architectures are mainly found in the field of embedded systems today. An example is the TI AM1710 platform from Texas Instruments [100].

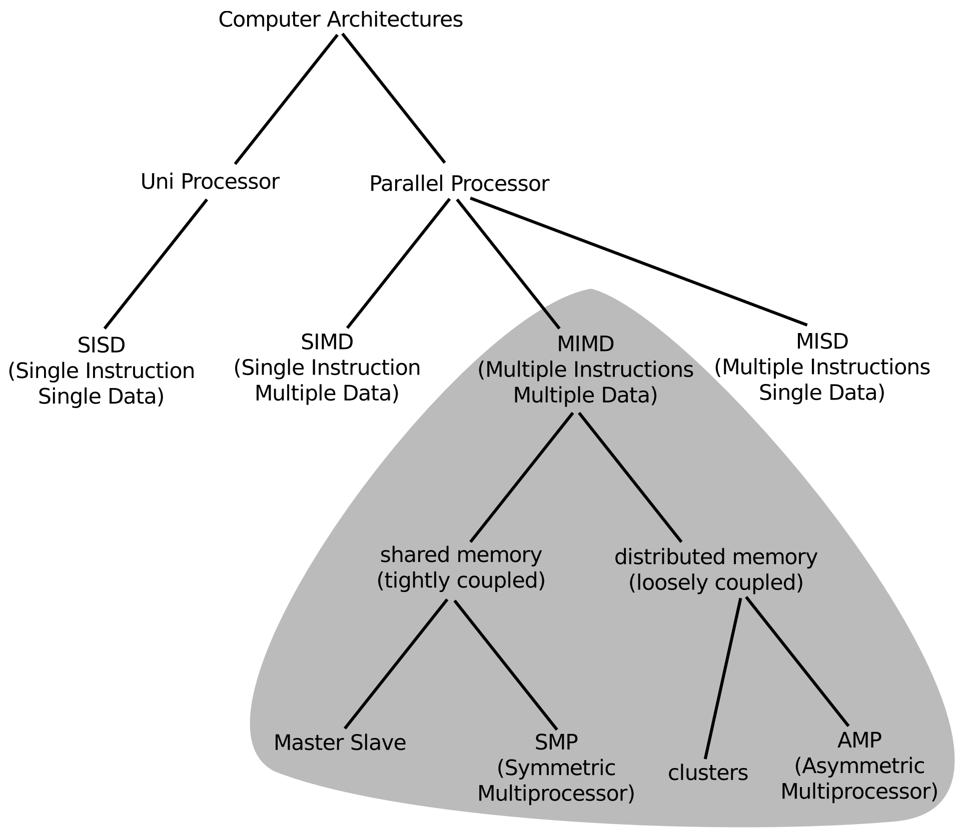

Computer Architectures
Uni Processor
Parallel Processor
SISD (Single Instruction Single Data)
SIMD (Single Instruction Multiple Data)
MIMD (Multiple Instructions Multiple Data)
MISD (Multiple Instructions Single Data)
shared memory (tightly coupled)
distributed memory (loosely coupled)
Master Slave
SMP (Symmetric Multiprocessor)
clusters
AMP (Asymmetric Multiprocessor)

Figure 2.1: Flynn's Taxonomy [97]

Furthermore, there exist another two ways to categorize parallel computing systems. On the one hand, it is possible to distinguish between Asymmetric Multi-Processor (AMP) and Symmetric Multi-Processor (SMP) systems. An SMP system implements a MIMD architecture, where all of the available processor cores utilize the same Instruction Set Architecture (ISA) [90]. A system consisting of several processors from at least two different ISAs are called AMP.

On the other hand, there exists a classification into *homogeneous* and *heterogeneous* systems. From a hardware point of view, these two terms can be used synonymously with the definition of SMP and AMP [90]. However, from a software perspective, the term *heterogeneous* describes a system executing at least two different operating systems on a given hardware platform, which may implement an AMP or a SMP. Considering *homogeneous* systems, only one operating system may be used on an AMP or SMP system from the software point of view [41, 90, 104]. In this thesis, the terms *homogeneous* and *heterogeneous* are used from a software point of view, if not stated otherwise. Figure 2.1 depicts the categorization introduced in order to provide an overview on the parallel computing taxonomy. Considering Figure 2.1, SMP and AMP are not bound to *tightly*, respectively *loosely-coupled* memory architectures, but these are their most common implementations [90]. Moreover, the sub-tree discussed in this thesis shown in Figure 2.1, as enclosed gray area. Furthermore, there exist different techniques implementing parallelism in computer systems. Therefore, the subsequent Sections 2.2 and 2.3 provide an introduction of hardware and software related techniques to realize parallel systems.

2.2 Hardware Parallelism

This section discusses techniques to implement parallelism at a hardware level, thus supplying computing systems with the components necessary to work on at least two *tasks* simultaneously. The term *task* refers to a wide variety of granularity, reaching from parallel execution of instructions to parallel computation of whole programs.

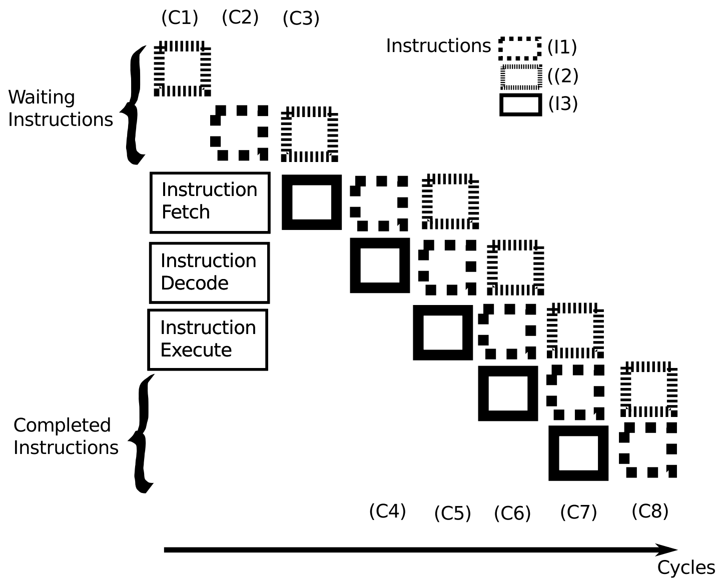


Figure 2.2: Filling and Emptying of a Three Stage Pipeline

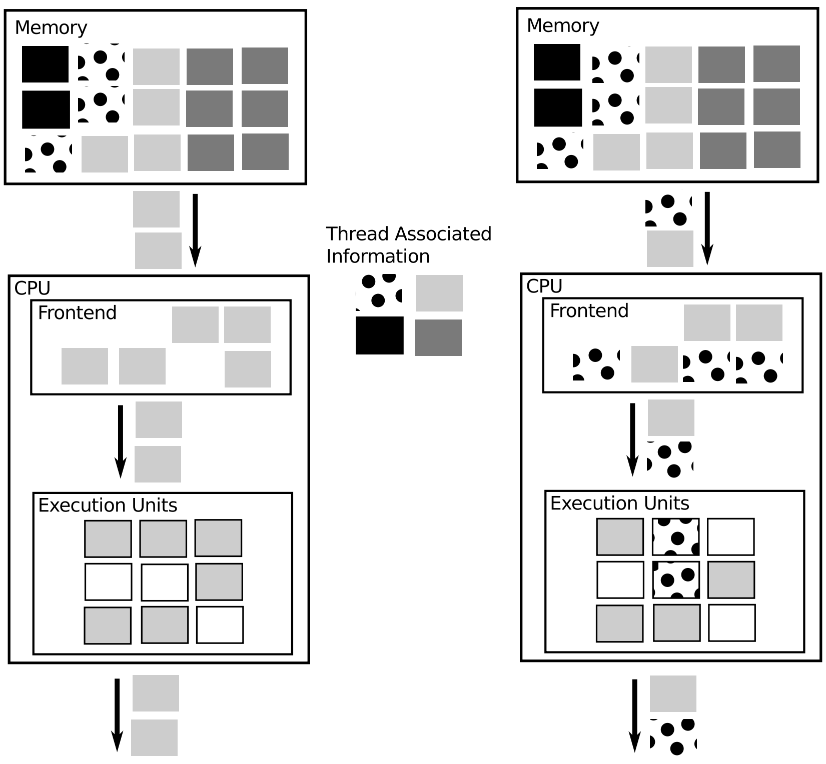
Instruction Level Parallelism

The first method described to implement parallelism in a computer system is called Instruction Level Parallelism (ILP) [82]. In fact the processors pipeline realizes the parallel processing of data or instructions by interleaving. In Figure 2.2 a three stage pipeline[3] executing three independent instructions in parallel using ILP is shown. As it can be seen it takes three cycles *(C3)* - *(C5)* until the pipeline is full (*(C5)*). Subsequently, it takes another three cycles *(C6)* - *(C8)* until it is emptied again. However, in cycle *(C5)* the pipeline is executing three instructions *(I1)* - *(I3)* concurrently. In order to exploit ILP the following conditions must be fulfilled:

(1) No data dependencies: A data dependency exists if the result of instruction i depends on a parameter of instruction j. Moreover, an instruction j is data dependent on instruction k, whereas instruction k is data dependent on instruction i [82].

(2) No name dependencies: Name dependencies refer to a case when at least two instructions use the same register or memory location to operate on.

[3]It consists of the three basic steps: Instruction*Fetch*, Instruction *Decode*, and Instruction *Execute*



(a) Superscalar Architecture (b) Symmetric Multi-Threading Architecture

Figure 2.3: Processor Architecture Overview

(3) No data hazards: Data hazards denote two instructions that would alter the results of their operands when being executed in parallel. The distance between two instructions that are allowed to depend on each other is bound by the pipeline length [40].

(4) No control dependencies: A control dependency occures when ordering of instructions are determined by a branch. Therefore, this condition states that the order of instructions determined by branches must not be changed [40].

Finally, ILP is used in all processors including a pipelined architecture. This is true for every processor since about the second half of the 1950's [93].

A generalization of the ILP approach is the so called *Superscalar* processor architecture [93]. Here, a processor is capable of processing two or more instructions in parallel. This is possible since the most important resources in the pipeline are accessible separately, or are available multiple times (e.g. separate store/load units, separate but multiple available integer execution units). Nevertheless, a processor implementing this architecture can only execute instructions associated with a single thread/process in parallel. But, due to the fact that a series of instructions may be independent from each other, they can be executed concurrently because it is possible to utilize several of the available execution units at once. Examples of processors implementing a *Superscalar* architecture are the *MIPS R10000*, and the first *Pentium 4* processors [77]. Figure 2.3a gives an overview on this kind of architecture.

Therefore, to further exploit the parallel potential of certain software components, the next logical step was to increase the number of software threads being executed without the necessity of task switches by the operating system. This has been realized by the implementation of truly Multi-Threaded processor architectures.

Hardware Multi-Threading

Multi-Threading describes the concurrent execution of software components (threads, processes) on a given processor [94]. The first attempts of multi-threading were performed in software, where an operating systems scheduler has to issue context switches[4] to execute threads/processes in a pseudo parallel way. The term pseudo parallel refers to the fact that SISD architectures could only execute threads/processes by interleaving. This is also true for processor architectures implementing ILP, or the *Superscalar* architecture.

To further push the performance increase of *Superscalar* processor architectures a hardware support for task switching has been proposed by the introduction of Simultaneous Multithreading (SMT), which is also called HT by Intel [40]. The intended performance gained on multithreaded software is achieved by at least doubling the number of available processor registers, stack pointers, and program counters. Moreover, the processor provides resources for thread context sensitive pipeline flushes, subroutine return predictions and trapping [83]. The multiple availability of these resources causes the operating system to perform task switches in a faster way, since all that has to be done is to switch to the next thread context by changing to the next available regis-

[4]Context switches oblige a lot of costly operations like saving processor registers, stack pointer, and the program status word to the stack.

ter bank. Nevertheless, threads and processes are still executed in a pseudo parallel scheme [40,94]. Figure 2.3b shows the operation principle of a processor implementing the SMT architecture.

However, the further increase of parallel execution units and mirrored registers sets does not scale linearly with the estimated performance [77]. This is merely caused by the fact that the administrative overhead considering queuing and multi-port register files results in a limit on the performance return of an SMT architecture [77]. For example, when upgrading a four issue[5] machine, to an eight issue machine, the performance increase will only be about 20% [77]. Therefore, the next step in the development of processors being able to execute threads and processes in a truly parallel way was the proposal of *Chip Multi-Processor* architectures.

Chip Multi-Processor (CMP)

According to Olukotun et al. [77], the consequences of the limitations of *Superscalar* processors led to decentralized microprocessor designs in order to achieve anticipated performance gains. Moreover, the idea of implementing several comparatively small processor cores on a die, instead of one complex multi-issue design supports the processor's flexibility and scalability considering software execution [53]. This has been observed by Wall [107]. In his study, the author identified two general classes of programs executed on a processor.

- The first class describes programs with low to moderate inherent parallelism. These programs usually implement a large number of integer operations. However, they are not able to be fully exploited by a given multi-issue *Superscalar* architecture, due to their lack of larger amounts of independently executable code. Due to the fact that CMP processors implement independently operating processor cores on one chip, this class of programs can be executed in a more efficient way. This argument is also supported by [77], who showed that programs falling into this class scale worse on a four issue *Superscalar* processor than on a two

[5]The term *issue* denotes the number of parallel execution units in a *Superscalar* or *Symmetric Multi-threading* processor.

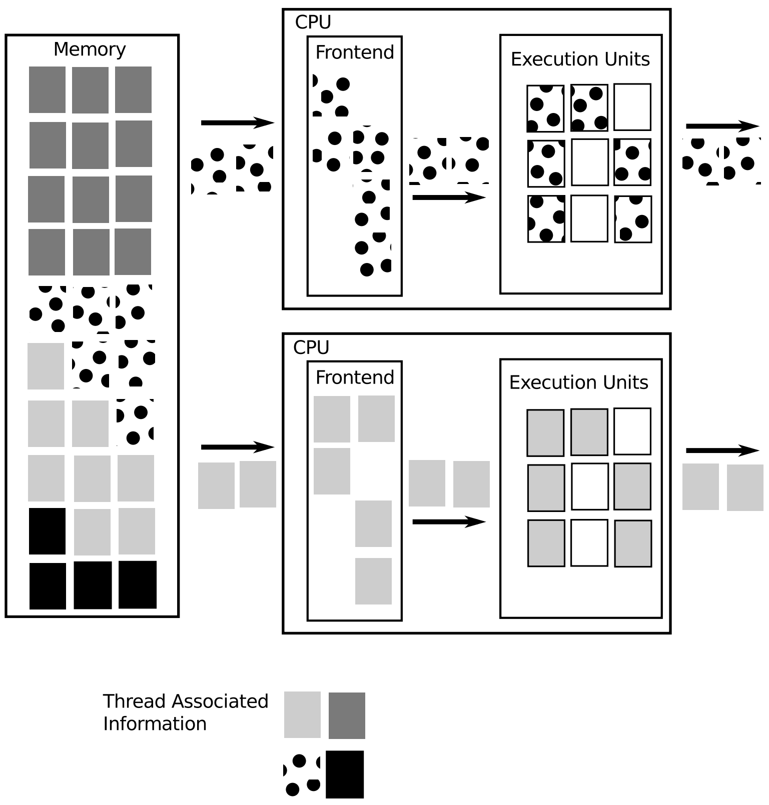


Figure 2.4: Chip Multi-Processor Architecture

issue *Superscalar* processor, both operating at the same clock frequency and implementing the same instruction set base.

- The second class represent programs containing a high amount of parallelism, while implementing floating point operations mostly. Although these kinds of programs would perform well on a *Superscalar* processor, they would not perform worse on a CMP, since several available processor cores could be used to exploit the available parallelism capability.

Therefore, CMP systems implement execution models being able to cope with both, low to moderate, and high parallelism capabilities [77]. Figure 2.4 shows a CMP system implementing two processor cores. Nevertheless, there exist three general types of application areas inducing architecture variants of CMP systems according to Kumar et al. [53].

Server & High Performance Computing This group is generally used in business or scientific computing. Thus machines in this field have to process a high amount of independent data using commercially available software (e.g. database servers). In the field of high performance computing, highly specialized software components are used to exploit a processor's architecture in an effective way. Therefore, CMP systems housing a large number of simple, but yet powerful processor cores can be used for these scenarios. Due to cost effectiveness reasons computers featuring these processor architectures have to operate on a high workload most of the time.

Desktop Computing In contrast to high performance computing systems, the average user has to master only a small number of tasks in parallel. However, in order to use software developed for SISD processors in a convenient way, it is highly desirable to have a smaller, but more powerful number of cores in a system belonging to this requirement group.

Embedded Computing Embedded systems do not have unified performance requirements. Their concern is rather application specific. An example for opposing system requirements in the field of embedded systems is the modern cell phone. On the one hand, the phone is obliged to have a long operational and standby availability, thus consuming as few power as possible. On the other hand, a modern phone must service its user with a lot of additional features like a music player or 3D graphics support. These media features need a lot of processor power in contrast to a phone's main tasks. Therefore, a heterogeneous CMP solution can often be found in these cases.

Chip Multi-Threading (CMT)

Due to the ever decreasing feature size and the resulting increase of transistors on a chip it is possible to implement hardware SMT features in a CMP system, as it is shown in Figure 2.5 [94]. This so called *Chip-Multi-Threading* is implemented by the high end models of Intel's *Core I7* and *Core I5* processors, for example. The motivation for implementing such systems is located in the field of highly parallel software, such as data base applications. Usually, such kind of software serves several hundred of users, where

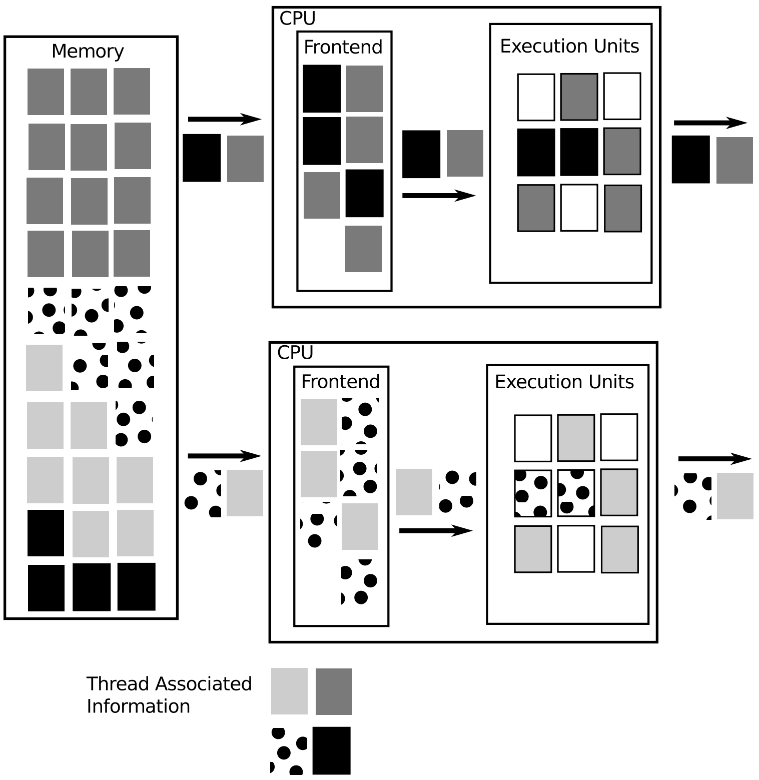


Figure 2.5: Chip Multi-Threading Architecture

each user is associated with a process, consisting of several threads in order to hide disk access latencies [94]. Therefore, a processor design implementing truly parallel execution of processes in combination with fast task switching capabilities is very well suited for an environment realizing a high rate of Thread Level Parallelism (TLP). However, the design of CMT capable processors is not done by simply instantiating whole processor cores multiple times on one chip. Instead there exist a lot of shared resources which are only available in a fraction of the physically available cores, such as Floating Point Units (FPUs), for example. Therefore, there exist special requirements when exploiting the capabilities of CMP and CMT systems, regarding interconnect technologies [94].

Special Issues regarding truly Parallel Executing Processors

Besides the fact that CMP and CMT processors provide solutions to the problems on how to deal with parallel programs in an efficient way, there still exist various challenges

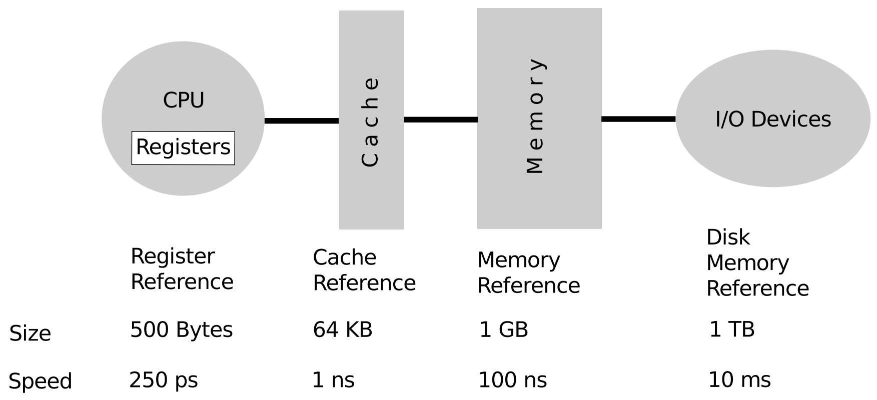


Figure 2.6: Memory Hierarchy [40]

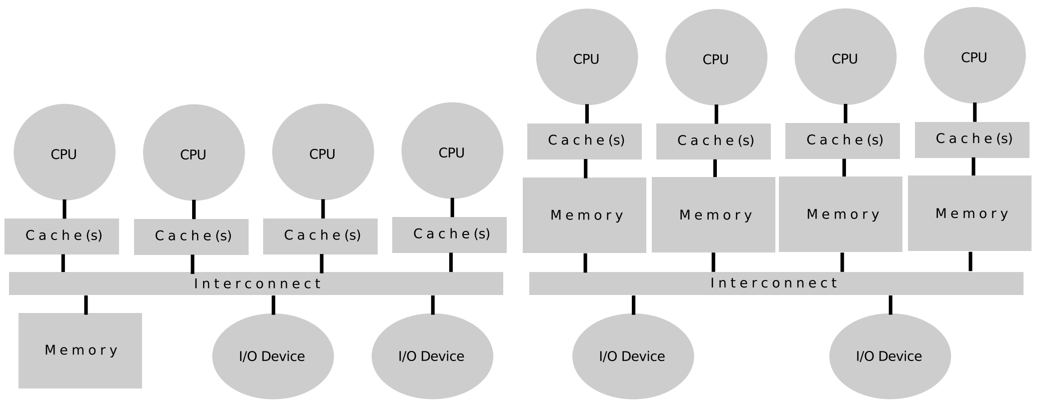
in exploiting these capabilities which are discussed in the following Subsections.

Memory Hierarchies

The design of memory hierarchies has to cope with two contradictory conditions. On the one hand, programmers want to have access to an infinite space of fast memory. On the other hand, the faster the memory, the more expensive it is. In order to find a balanced tradeoff between the costs and the performance constraints, a computer's memory is organized in a set of layers, as it can be seen in Figure 2.6. In the first layer, a set of registers expose a small amount of storage to the Central Processing Unit (CPU). This kind of memory is implemented inside the processor core and can be accessed within a few clock cycles. The next layer represents the so called cache memory. The cache can be seen as a memory, exploiting the principle of locality in an aggressive way. The adjacent layer is called main memory in general and is located off chip, thus it is necessary to let the processor communicate with a dedicated memory controller[6] to access the main memory, thus decreasing the access speed. The last layer includes mass storage devices, like a hard disk or a flash memory [40].

Furthermore, considering CMP and CMT systems it is possible to distinguish between *shared memory*[7] and *distributed memory*[8] architectures, as explained in Sec-

[6]Sometimes, these memory controllers reside on-chip as well.
[7]This architecture is also known as Uniform Memory Access (UMA)
[8]This architecture is also known as Non Uniform Memory Access (NUMA)



(a) Shared Memory Hierarchy (b) Distributed Memory Hierarchy

Figure 2.7: Memory Hierarchies of Multi Processor Systems

tion 2.1. As shown in Figure 2.7, both memory hierarchy models consist of memories only local to a specific processor. Therefore, the necessity for memory synchronization and consistency arises [40].

Nevertheless, synchronization of concurrent processes was important in uni processor systems[9] as well, but could be realized without the need of additional hardware support apart from the common *Test & Set* instructions necessary for implementing synchronization primitives, such as semaphores or spin locks [22]. This was possible since concurrent processes in a uni processor systems were actually executed in a pseudo parallel way [94]. However, when dealing with multi processor systems there has to be additional hardware support for synchronizing the access to shared memory regions, as well as keeping them in a consistent state.

Synchronization

To gain synchronized access to a central resource, the mutual exclusion problem has to be solved [20]. Basically, the problem states, that when multiple processes compete over a single resource it must be assured that one process has exclusive access to that resource. Furthermore, it must be guaranteed that a process has only access to the resource for a finite amount of time. Moreover, it must be asserted that also other

[9]Including Superscalar and SMT architectures

processes can enter the shared resource once the process possessing it has left.

Since synchronization is a common problem in both, uni processor and multi processor systems, there exist dedicated instructions to perform these tasks. They are usually subsumed under the term *Test & Set* instructions [22]. However, there exist major differences in the implementation of *Test & Set* instructions between uni processor and CMP systems. In order to illustrate these difference, the synchronization instructions of the ARMv5 and ARMv6 ISAs are compared [1]. Here, the ARMv5 ISA represents a uni processor instruction set, whereas the ARMv6 symbolizes a multi processor capable ISA.

Listing 2.1: ARMv5 compatible Spinlock Implementation [61]

```
static inline void __spin_lock_nds(nds_spinlock_t *lock)
{
  unsigned long tmp;

  __asm__ __volatile__(
  "spin:  ldr %[tmp], [%[lock]]\n"
  " cmp %[tmp], 0\n"
  " bne spin\n"
  " ldr %[tmp], 1\n"
  " swp %[tmp], %[tmp], [%[lock]]\n"  /* atomic */
  " cmp %[tmp], 0\n"
  " bne spin\n"
  :
  : [lock] "r" (&lock->lock), [tmp] "r" (tmp)
  );
}
```

Synchronizing pseudo parallel executing processes using an ARMv5 ISA can be done with the help of a single instruction, called **swp**. This instruction performs a combined load and store operation, making it possible to exchange the value of a variable in an atomic way. This is necessary, since an occurring interrupt request would intercept a non atomic load and store operation, thus resulting in the possibility that two processes will access the shared resource, due to a context switch. The read back value of the atomic **swp** instruction is used to determine the actual state of a lock, guarding the entrance to a shared resource region, for example. A possible implementation of such a guarding routine can be seen in Listing 2.1. This procedure works as follows: The variable *lock* represents a guarding variable and is accessed by all processes that want to utilize the shared resource. In order to do so, the variable *lock* must be set to zero. As long as *lock* ist not set to zero, it is assumed that another process has access to the shared resource, thus a process wishing to use it as well has to wait. This is done in lines 6 to 8 of Listing 2.1, where the instruction **bne** denotes that the program flow shall branch to the label **spin**, if the preceding compare operation did not return *true*. Line 9 loads a value indicating an occupied resource in the variable **tmp**. Subsequently the instruction **swp** is executed performing the *Test & Set* operation. If the read back value is equal to zero, the shared resource has not been used by another process. Therefore the process attempting to occupy it succeeds (line 11). Otherwise, the CPU will branch back to the label **spin** in line 6).

Although the mentioned **swp** instructions are available in the ARMv6 ISA as well, they are not suited for synchronization in multi processor environments, since **swp** instructions do not perform write buffer[10] flushes [1,3]. Therefore, the instructions **ldrex**, and **strex** are available in processors implementing the ARMv6, and above architectures. These two instructions implement system wide exclusive load and store operations. This is done by marking an address accessed by using **ldrex** exclusively for a specific processor. Subsequently, **strex** can be used to store a date exclusively on the memory location marked by **ldrex**, thus making it possible to implement synchronization primitives in multi processor environments. In order to get a better overview on how synchronized access on a shared resource can be done in a multi processor environment, Listing 2.2 presents an example implementation of a Spinlock for an ARMv6 processor.

Listing 2.2: ARMv6 compatible Spinlock Implementation [61]

```
static inline void __spin_lock_nds(nds_spinlock_t *lock)
{
  unsigned long tmp;

  __asm__ __volatile__(
  "spin:  ldrex %0, [%1]\n"
  " teq %0, #0\n"
```

[10]Using write buffers is a common technique to bypass memory latencies [3].

```
 8      " strexeq %0, %2, [%1]\n"
 9      " teqeq %0, #0\n"
10      " bne spin"
11      : "=&r" (tmp)
12      : "r" (&lock->lock), "r" (1)
13      : "cc");
14
15      smp_mb();
16  }
```

In line 6 an exclusive load operation is performed, to retrieve the current state of the variable *lock*. Subsequently, the value is compared against zero (line 7). If the comparison **teq** returns true, the instruction **strex** is issued (line 8). The **eq** post-fix denotes that the instruction **ldrex** must only be executed if a preceding comparison evaluated to true. Otherwise, a *no-operation* instruction is executed. Eventually, the *lock* is occupied once **strex** was executed successfully[11]. Eventually, a system synchronization instruction encapsulated in the function *smp_mb()* is called in line 15.

Memory Consistency

From a uni processor system's point of view memory consistency is not a big problem, since read and write operations affect memory in the order they were issued. This is also true for processors being able to perform memory operations in an out of order[12] manner if the following rules are obeyed.

(1a) It must be assured that two instructions do not reference the same data.

(2a) It must be guaranteed that one instruction does not control the execution of the other (e.g. an IF clause).

Once these two conditions are fulfilled it is possible to perform memory operations out of order. In essence these two conditions enable compiler optimizations like register allocation, code motion, loop transformations and several hardware optimizations such

[11]This is indicated by writing a zero into the return register (First parameter of the instruction).

[12]Out of order means that a certain processor can execute instruction k before instruction i, although i is located before k in the program flow, if both instructions are independent from each other. [40]

as ILP, for example [3]. Therefore, the topic of memory consistency describes models that allows the programmer to have a consistent view on all memory operations, such that they behave in the same way as the program execution order depicts it. Thus, one expects a read operation to return the value of the last recent write operation to the very same memory location.

However, when it comes to multi processor systems it is not sufficient to fulfill conditions **(1a)** and **(2a)** only in order to maintain memory consistency. In order to have a consistent memory view in a shared memory multi processor environment Lamport [54] proposed the *Sequential Consistency* model in 1979. The author identifies two major aspects in keeping memory operations sequentially consistent:

(1b) Maintaining program order among operations from individual processors and

(2b) maintaining a single sequential order among operations from all processors.

The second aspect results in a presentation of memory operations that appear to be *atomic* regarding memory operations from another processors point of view. Beside Lamport's conditions to retain sequential consistent memory operations in multi processor environments, Adve and Gharachorloo [3] examined numerous multi processor systems with focus on their memory hierarchies, and read/write operation implementations. Eventually, the authors derived the following constraints on sequential consistency:

- A processor must ensure that its previous memory operation completes before it issues a new one in program order. This requirement is called *Program Order*. Moreover, a write operation must generate invalidate or update messages for all cached copies.
- If a system utilizes caches it must assure that writes to the same location are serialized. Thus, the value of a write must not be returned by a read until all other processors have updated their caches. This requirement is called *Write Atomicity*.

Finally, due to the fact that the sequential consistent memory model introduces several bottle necks because of its firm requirements, several models relaxing the sequential consistency model have been proposed [3, 40]. To categorize the relaxed memory order models we use the formal definition of [40], which corresponds to the two requirements *Program Order* and *Write Atomicity* as proposed in [3]. A rule has the following syntax: $X \rightarrow Y$. Its semantic is, that operation X must complete before operation Y. The *sequentially consistent memory model* can be described with the following rules: $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, $W \rightarrow W$. The following relaxing memory models are named after the rules they relax in the *sequentially consistent model*.

- A model known as *total store ordering* or *processor consistency* relaxes the $W \rightarrow R$ requirement. This model retains ordering during write operations. It is commonly used by programs operating under the sequential consistency model, without the need for additional synchronization.
- Relaxing requirement $W \rightarrow W$ results in a model called *partial order store*, which is implemented in certain SPARC processors, for example.
- The third memory model relaxation omits the requirements $R \rightarrow W$ and $R \rightarrow R$ and realizes a variety of models known as *weak ordering*. This kind of model is used in the PowerPC consistency model, for example.

After introducing the theoretical background of memory consistency models in multi processors environments, the subsequent Subsection will present an overview on techniques realizing these models.

Enforcing Consistency

The two major methods to coerce memory consistency in shared memory multi processor systems are represented by cache coherency protocols and memory barriers [66]. As described in the Subsection ”Memory Hierarchies” in Section 2.2, caches can be described as some kind of short term memory to a processor. Moreover, the presence of a cache results in a performance increase since the processors pipeline would have to be stalled each time the memory would have to be accessed without having a cache. Furthermore, when a multi processor system is deployed as shared memory architecture each processor might have its own private cache, as it has been presented in Figure 2.7a. Thus to compel a consistent view on memory, it is necessary to let the caches share their information by utilizing cache coherence protocols.

Nevertheless, in order to understand the operation of cache coherence protocols, the basic functionality of a cache is explained in these paragraphs. First, we will explain the terms of definitions used when dealing with caches. Subsequently, the functionality of the example cache shown in Figure 2.8 is explained.

The smallest accessible amount of memory that a cache can exchange with main memory is called *cache line*. These lines are typically between 16 and 256 bytes wide. In Figure 2.8, a cache line is represented by a cell in the table (E.g. The cell which states $0x12345000$). In our case, the cache line is 256 bytes wide. Thus each cache line represents a 256 byte wide consecutive memory block. This makes our cache rather large. Caches of this size are usually organized as hash tables with fixed sized buckets, which are named *sets* from a hardware developer's point of view. A *set* of *lines* is called *way*. Because the location of the *line* in the cache is determined by a simple hash function looking at four address bits at the start address of the *line*, the cache can be referred to as *associative*. Therefore, the cache depicted in Figure 2.8 implements a *two way set associative cache*.

The cache operates in the following way. Once a CPU is reset, and the cache has been activated by software, all memory transfers using cache able memory regions must pass through the cache. Due to the fact that caches are initially empty, the first access to the cache results in a so called *miss*. Assume that this request aimed at a word stored at memory address $0x12345308$. Since the requested word is located in the address range between $0x12345300$ and $0x123453FF$, the whole memory block starting at address $0x12345300$ with a length of 256 bytes is transferred into the cache, and indexed at *way* 0, line $0x12345300$[13]. The memory block index is returned by the cache's hash function. Therefore, it is also possible that two words from distinct address ranges evaluate to the same index position in the cache (See address index $0xE$ in Figure 2.8). Here, a memory block from the base address $0x42310E00$ has been stored at *way* 1, because *way* 0 has already been occupied by another memory block belonging to the base address $0x12345E00$. However, since caches have no infinite capacity, there exist replacement strategies, like Least Recently Used (LRU)[14] in order to exchange data

[13]This makes sense because of the principle of locality, which states, that it is very likely that once data is accessed at a specific location, there will be more accesses to addresses nearby [40].

[14]The cache replacement strategy LRU, drops the least common accessed memory block after a certain amount of cache hits. [40].

stored in the cache with data located in main memory when necessary [40, 66].

When it comes to SMP systems, the following question arises with the use of private processor caches. How does CPU 0 know about a change of a variable located in CPU 1's private cache?

The answer to this question are so called coherence protocols. Their key feature is to track the state of any shared data block, as well as keeping data up to date [40]. In general there exist two different classes of cache coherence protocols.

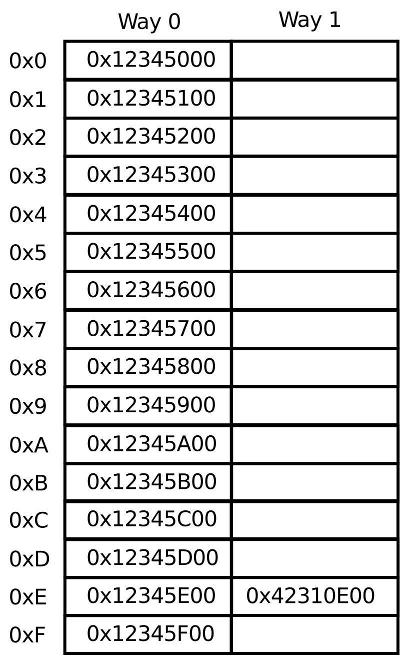
Directory based This kind of protocols provides a central book keeping mechanism to track the state of shared data. Although directory based protocols have a higher implementation overhead they are better suited for large scale multi processor systems [40].

Snoop based A distributed approach on keeping memory in a consistent state is followed by snoop based cache coherence protocols. Here, each cache controller listens on a shared bus in order to gain knowledge on which memory blocks must be synchronized in order to keep a system in a consistent state. This technique is widely used in today's desktop and server processors with few processing cores [40].

Basically, both protocol types consist of several states indicating the current condition of each cache *line*. Each of these states may send messages to each other if an appropriate action, like a write operation to a cached memory location that is also present in another processors cache, is induced by one of the CPUs participating in the SMP[15].

Since modern processors are able to execute instructions out of order, or issue read operations before an earlier write due to the anticipation of write buffers, so called *memory barriers* were introduced to guarantee an in order access to memory and IO devices. *Memory barriers* are interventions that instruct the compiler to retain the order of instruction execution as implemented by the developer, thus avoiding to apply some kind of optimization techniques in order to execute the given code faster [66]. There exist several types of *memory barriers*:

[15]Therefore, it is sometimes mentioned that shared memory systems base upon message passing system [66]



	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x42310E00
0xF	0x12345F00	

Figure 2.8: A 2-Way Set Associative Cache [3]

General Barriers This kind of barrier enforces that both, memory read and memory write operations are finished, before any of the memory read, memory write operations after the barrier are executed.

Read Barriers If this barrier is used it is guaranteed, that all read operations are located before the barrier are finished before any of the read operations after the barrier are executed. This does not apply for memory write operations.

Write Barriers If this barrier is used it is guaranteed, that all write operations before the barrier are finished before any of the write operations, after the barrier are executed. This does not apply for memory read operations.

Data Dependency Barriers This kind of barrier implements a weaker version of the read barrier. Its usage is restricted to data dependencies only, thus it could be applied if a second read operation directly depends upon a first read operation. This could be the case if the first read operation loads an address, which is used to load data within the second read operation.

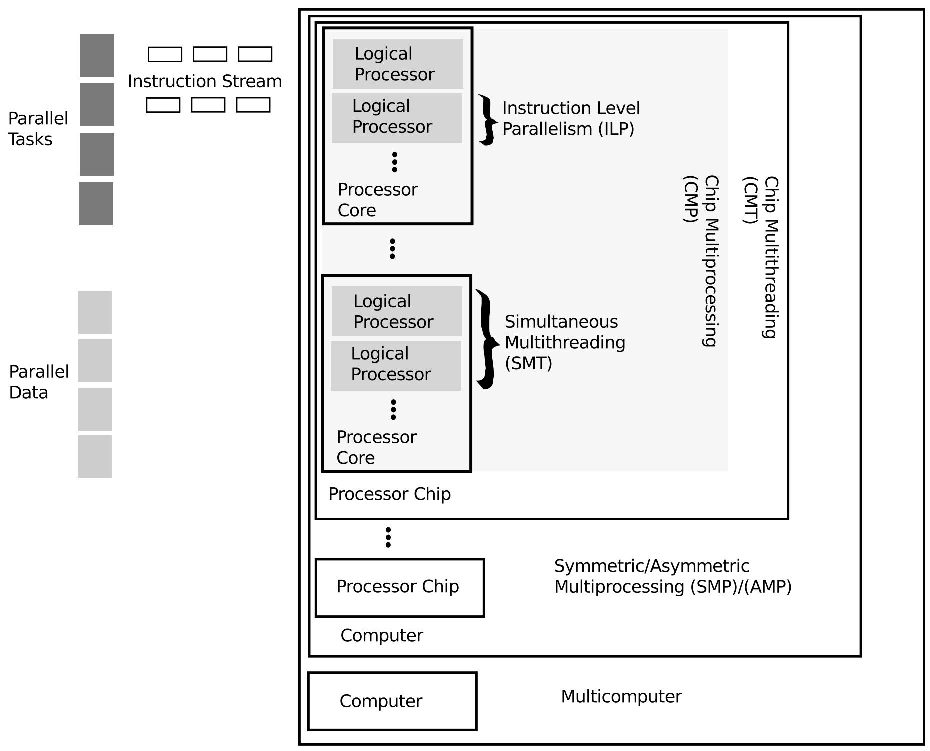


Figure 2.9: Categorizing Hardware Parallelism [104]

Eventually, memory barriers in conjunction with cache coherency protocols realize a consistent and ordered view on memory transaction in multi-processor systems.

Section 2.2 gave an introduction on parallelism from a hardware point of view. First, several methods of hardware parallelism from instruction level parallelism to chip multithreading were discussed. Second, special issues regarding parallel executing hardware were introduced. This includes the discussion of memory hierarchies, memory consistency in general, as well as techniques to enforce a consistent view on memory transaction during the execution of truly parallel symmetric multi processor systems. Finally, Figure 2.9 presents a hierarchically structured overview on the introduced techniques to implement parallel computing systems. The figure is structured in the following way. The mangled data and instruction streams as described in Section 2.1 are depicted on the outer left side of Figure 2.9. The *Logical Processor* blocks represent a single issue pipeline, capable of implementing Instruction Level Parallelism (ILP). Since it is possible to have several components of such a pipeline available multiple times, the next hierarchical step represents Simultaneous Multithreading (SMT) architectures subsuming Superscalar systems as well. Systems implementing several processor cores of the just mentioned feature are usually called Chip Multi-Processor (CMP) systems. However, CMP architectures implement Superscalar processor cores only. Therefore the term Chip Multi-Threading (CMT) takes SMT capable Central Processing Units (CPUs) into account. From a hardware point of view, the use of the same Instruction Set Architectures (ISAs) on one chip, or on several chips realizing a parallel computing systems can be described as a Symmetric Multi-Processor (SMP) system. Despite, the use of several distinct ISAs would be called an Asymmetric Multi-Processor (AMP) system. This fact is shown in the third layer of Figure 2.9. Finally, a system consisting of several microcomputers is called a multicomputer. This kind of architectures represent distributed memory systems. However, also CMT systems can be realized as distributed memory architectures for example. The subsequent section will introduce software based approaches to exploit hardware provided parallelism.

2.3 Software Parallelism

The previous Section presented numerous methods to potentially accelerate the execution of parallel software by increasing the number of hardware components computing on parallel tasks. Unfortunately, a linear increase of processors on a program with a given percentage of parallel executable code does not result in a linear performance gain. This circumstance is known as Amdahl's law [6], and can be calculated as seen in Equation 2.1, where $S(N)$ is the achieved speedup by using N processors on a program consisting of a parallel portion of P. As shown in Figure 2.10, there exist speedup limits, even when using highly parallel code (P=95%) and more than 1000 processors trying to exploit the available parallelism. However, there exists a relaxation of Amdahl's law which has been proposed by Gustafson [38], stating that it is always possible to achieve speedup by choosing bigger problems. This essentially means that due to increasing computing power (using more and faster processors), it is possible to solve bigger problems in the same time as smaller ones, therefore leading to an overall computational speedup. Nevertheless, Amdahls and Gustafsons law are still applying to today's multi-core system according to Hill et al. [42].

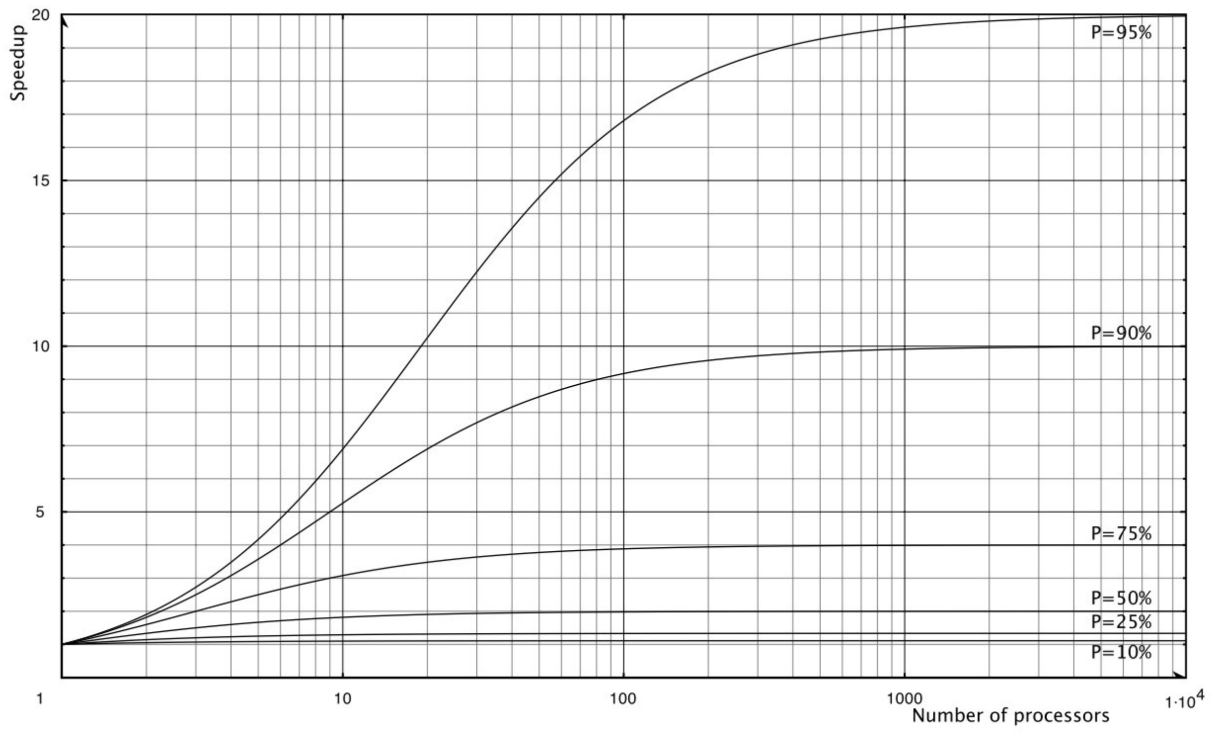


Figure 2.10: Amdahl's Law: Speedup using a program with a parallel portion P [104]

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

The remainder of this Section is structured as follows: First, a categorization of available approaches in supporting the development of parallel software is given. This categorization is twofold, and presents a tool and a model based taxonomy. Afterwards, the topics synchronization and communication are reviewed from a software point of view. Finally, this Section addresses special issues regarding the truly parallel execution of software. Finally, a short summary concludes Section 2.3.

Categorizing Software Parallelization

Before addressing special issues in truly parallel executing programs, it is necessary to introduce two techniques in parallel software categorization made by Tröger [104] and Skillicorn [92]. The first method presented by Tröger, will describe ways to develop parallel software at different supportive levels regarding libraries, languages, or tools. The second categorization approach made by Skillicorn focuses on a model centric view of parallel programming languages. According to Tröger [104] and Popovici [95], there exist various techniques to support programmers in developing parallel software.

Compiler Optimization In this approach, it is the compiler's responsibility to make use of the available processors in the appropriate way. However, it has been shown by Blume et al. [12] that this is very hard to achieve, since the compiler would have to take all possible side effects into account as well. Therefore, this kind of parallelism support is merely limited to loop parallelism, which describes an approach of mapping certain loop constructs to parallel threads, thus reducing the amount of required serial loop iterations.

Operating System The basic concepts of exploiting parallel hardware are provided by operating systems and are known as processes and threads. Hereby, a process can be viewed as a program which is executed in a protected memory area within an operating system environment. Since operating systems are capable of executing several processes in parallel, there arises the need for inter-process communication. However, inter-process communication, as well as processes communicating with operating system components, such as hardware drivers for example, require a certain amount of waiting time, causing these processes to be suspended and woke up by the operating system on demand [97]. Nevertheless, suspending and rescheduling processes results in a throughput decrease of parallel software due to the rather expensive administrative actions[16] that must be performed by an operating system to reschedule a task, for example [92]. Therefore, the concept of threads has been introduced in operating systems [92]. A thread does not have its own operating system protected memory region. Thus context switches within threads associated with a process are faster, and the number one choice for parallel software implementation on operating systems [97]. This argument is supported by the fact that a large number of parallel programming libraries, virtualization environments, and languages, actually use threads in order to glue their concepts of parallelism to the operating system. Tröger [104] calls this the *hour-glass* of parallel software since a lot of parallel software solutions encompass threads to

[16]Saving context of currently running process and restoring context of to be run process [97].

utilize a larger number of hardware components providing parallelization support (See Section 2.2).

Application Level The application level is formed by immanent parallel programming languages, extension to sequential languages, as well as domain specific languages encompassing parallelism in high-level language constructs [104]. However, since most of today's programs are written in C,C++ or Java, the most common parallel programs are actually implemented using language extension such as Open Multi Programming (OpenMP) [89], Message Passing Interface (MPI) [35], Cilk[17], or rely directly on one of the available threading libraries such as PThread[18], for example. Despite the large success of parallel extensions for sequential languages, languages implementing a parallel programming model by design are not very widespread. Instead, only few of the many parallel languages, which have been proposed remained. Amongst them are substitutes like High Performance Fotran [62], and OCCAM [71]. In contrast to general purpose languages like C or Java, domain specific languages focus on specific problem domains, like formal verification (Promela[19]) or database queries (SQL). Due to their high grade of abstraction for their specific problem field, some domain specific languages are not Turing complete [99].

Regardless of the used parallelism approach, software engineers have to decide on the degree of control they want to have when writing parallel software [104]. In order to be able to choose the right granularity of control for each task, Skillicorn et al. [92] identified five key concepts a programming model might hide before its developer, in order to aid her producing productive code. These concepts are:

- Concurrency or parallelism of the software,
- Decomposition of the software into parallel threads,
- Mapping of threads to processors,

[17] `http://supertech.csail.mit.edu/cilk/` - Last visit: 04/12/2012

[18] `http://www.gnu.org/software/pth/` - Last visit: 04/12/2012

[19] `http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html` - Last visit: 04/13/2012

- Communication among threads,
- Synchronization among threads.

Constituitively, Skillicorn defined six abstraction levels on base of his five key concepts. These levels of abstraction are presented in reverse order of abstraction. Thus the last mentioned abstraction level provides the greatest degree of freedom, but leaves almost all parallel programming challenges to the programmer.

1. The most abstract parallel programs are formed by languages implementing models that abstract parallelism completely. Therefore, the programmer has only to consider the meaning of the program, rather than how their program will execute in parallel. An example for languages implementing this model is Crystal [18].
2. The second kind of programming models is implemented by languages in which parallelism is made explicit, but decompositions of programs into threads, communication and synchronization is not. Therefore, programmers must only be aware of the fact, that their algorithm will be executed in parallel, but not how this is done. Such models oblige programs to express the maximal amount of parallelism, however the amount of used parallelism depends on the architecture the programs eventually operates on. An example for languages implementing this model is OpenMP [89].
3. The third abstraction level describes models where it is necessary to cope with parallelism and decomposition explicitly. However, thread mapping, synchronization and communication are done by the language implementing the model. An example for languages implementing this model is the specification language SDL [15], which is used by the ITU-T to model telecommunication protocols.
4. This model demands that software developers have to consider parallelism, decomposition, and mapping explicitly. However, the tasks synchronization and communication is done by the language realizing the model. When using such languages, the developer is required to have knowledge on the underlying architecture, since his mapping decisions will inevitably influence the software's performance on this architecture. Therefore, it is a challenge to design software

using this model in a portable way. An example for languages implementing this model is CORBA [36].

5. The fifth level of abstraction demands that the programmer is able to deal with the tasks parallelism, decomposition, mapping, and communication explicitely. Synchronization is done implicit by the language. An example for languages implementing this model is the Kahn Process Language [50].
6. This model offers the greatest degree of freedom to the programmer, as he has to consider all five concepts explicitly. However, according to [92, 95] it is *"extremely difficult to build software using such models, because both correctness and performance can only be achieved by attention to vast numbers of details"*. An example for languages implementing this model is the specification language OpenMPI [35].

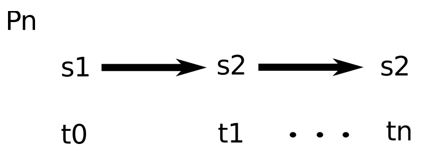
Beside the taxonomies of parallel software, there exist special issues that have to be taken into account when executing parallel programs on parallel hardware.

Special Issues regarding Software executed on truly Parallel Executing Processors

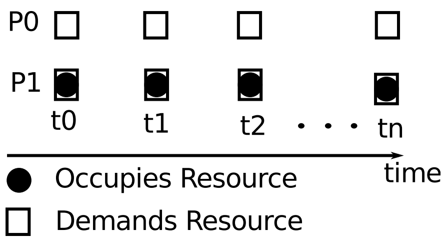
Subsection *"Special Issues regarding truly Parallel Executing Processors"* on page 14 mentioned two issues regarding parallel hardware, namely, consistency, and synchronization. This Subsection will investigate these two points from a software point of view.

Synchronization & Communication

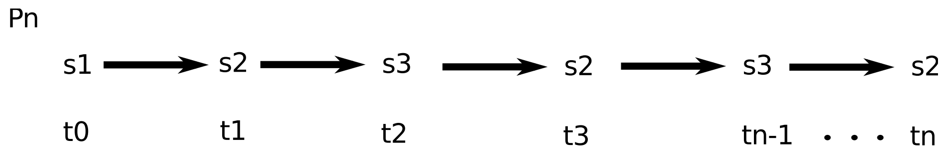
In Subsection *"Special Issues regarding truly Parallel Executing Processors"* on page 14 synchronization upon a central resource has been contemplated from a processor's instruction sets point of view. Usually, when developing parallel software on behalf of an operating system, none of these issues are of concern, except the issues arising when using the different types of synchronization primitives. The terms and techniques introduced in this thesis, apply for uni-core and multi-core systems. This is true since on unicore systems it is also possible to execute processes in a parallel manner. Nevertheless,



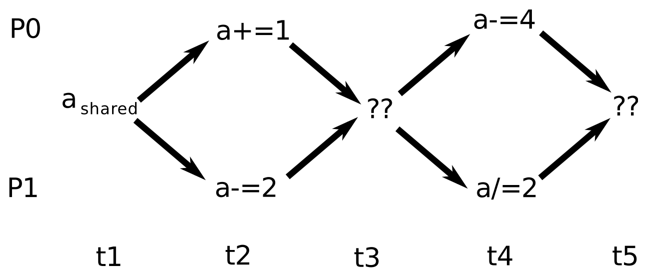
(a) Deadlock



(b) Starvation



(c) Livelock



(d) Race Condition

Figure 2.11: Special Issues in parallel programs

one has to synchronize processes and threads when trying to access a mutual exclusive resource. Since this is a very common problem in software engineering so called Inter Process Communication (IPC) methods have been defined [97]. These methods are usually provided, and exported to the programmer by the operating system. Amongst them are Locks, Semaphores, Sockets, Signals and Message. However, these synchronization tools may lead to the following issues in a program when not used properly [97]:

Race Condition This term describes a particular error occurring in parallel programs. It happens when the result of a concurrent program depends on the order the participating processes are scheduled. This kind of errors are usually hard to detect, since it is not easy to reproduce them in a deterministic way. However, race conditions result from synchronization errors. An example race condition is shown in Figure 2.11d. Here, two processes $P0$ and P operate on a shared variable a. However, $P0$ and $P1$ do not synchronize upon the access on a. Therefore, the outcome of the parallel executed equations at time $t3$ and $t5$ is arbitrary since the read and write order is not deterministic.

Deadlock If the advance of at least two processes depend on each other in a circular way resulting in a state where none of the processes can make an advance, this is called a deadlock. The situation might occur when process 0 has occupied resource A, and needs to occupy resource B as well, thus waiting for B to be freed. However, process 1 has already occupied resource B and needs to occupy resource A as well, thus waiting for resource A. Therefore both processes wait for the release of their missing resource without releasing their already occupied one. Thus leads to the state where process 0 and process 1 will wait forever, thus forming a deadlock. This example can also be viewed from a state transition perspective, as it is done in Figure 2.11a. Here, the progress of a program is reflected by a state change from sx to $sx+1$, thus an ongoing state change from sx to sx would reflect a deadlock, since it would show now progress in the program execution. As a consequence, process Pn in Figure 2.11a shows a deadlock since from time $t1$ to time tn the process stays in state $s2$.

Livelock Basically a livelock is the same as a deadlock, except for the fact that processors can still advance in circles inside a livelock, but cannot leave the circle, thus spinning there forever. This instant is reflected by Figure 2.11c, where a process Pn enters a loop from time $t3$ to time tn.

Starvation When a set of processes want to access a mutual exclusive resource like, a network interface card for example, but at least on of these processes is never allowed to interact with the resource (e.g. because its priority is too low), then this process suffers from starvation. This parallel programming issue is depicted by Figure 2.11b. Here, the processes $P0$ and $P1$ compete for a mutual exclusive resource. However, although both processes demand that resource, only process $P1$ gets access granted to it, thus leading to a starvation of process $P0$.

Convoying Assume a set of truly parallel executing processes operating on several mutual exclusive resource protected by a synchronization primitives like locks. If the order of access to these shared resource is of matter, it is possible that all processes competing for the resource end up accessing the mutual exclusive resource in a serial way. This is called convoying, like a convoy of trucks driving serially on a high-way [69].

Priority Inversion This phenomenon can occur when a high priority process cannot access a shared resource, because a process with a lower priority has occupied the lock and is currently preempted. Therefore, the lower priority process causes the high priority one to wait, thus inverting the access priority. [69]

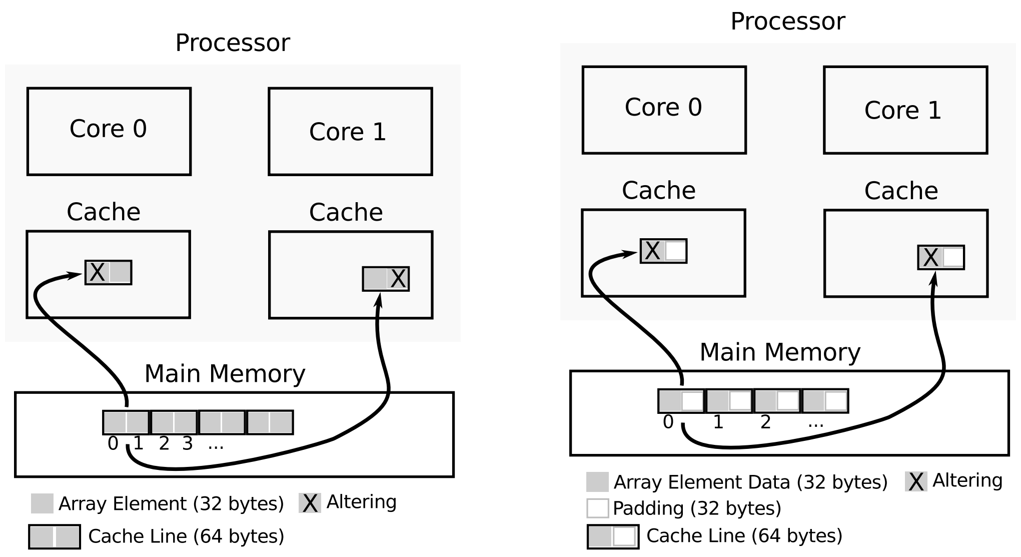
Finally, it is the programmer's responsibility to be aware of these side effects regarding parallel programming. The following subsection discusses memory consistency from a software point of view.

Consistency

From a software developer's point of view, memory consistency is guaranteed when she obeys the rules of synchronization and avoids race conditions. However, when developing parallel software on shared memory systems there exists certain potential to introduce performance bottle necks by ignoring the processor's cache architecture. Consider the code example depicted on Listing 2.3 with respect to a cache with a line size of 64 bytes. The used processor is a two core 32-bit CPU, with an integer size of 4 byte. In this example, an array *sh_array* is used to house 128 elements of type *struct container*, whereas each element requires 32 bytes of storage. The two functions *thread_operate_on_even* and *thread_operate_on_odd* will be executed concurrently, where each thread is going to be placed on a dedicated core of the processor. As it can be seen there is no thread synchronization required since thread 1 will only operate on even array indices, and thread 2 will manipulate odd array elements only. Therefore, it may be assumed that this kind of task partitioning will result in a speedup of 100%. However, this assumption is shown to be incorrect due to false sharing [14].

Listing 2.3: Example code to evocate False Sharing

```
struct container {
```



(a) False Sharing (b) Avoiding False Sharing

Figure 2.12: False Sharing and its avoidance

```
  unsigned int id;
  unsigned char data[28];
};

struct container sh_array[128];/*will be shared*/

/*thread 1*/
void *thread_operate_on_even(void *arg) {
  for(i = 0;i <128;i = i+2)
    sh_array[i] = calc_on_even(sh_array[i]);
}

/*thread 2*/
void *thread_operate_on_odd(void *arg) {
  for(i = 1;i <128;i = i+2)
    sh_array[i] = calc_on_odd(sh_array[i]);
}
```

False sharing happens when the software developer does not consider the line size of its processors cache. In our case, a cache line is 64 bytes large. However, an item of the container array occupies 32 bytes only, therefore, two consecutive array elements are stored within a 64 byte long memory block. Since thread[20] 1, running on processor core 0, operates on even, and thread 2, running on processor core 1, manipulates odd array elements each processor core invalidates the whole cache line, once it writes to its memory location. As a consequence the cache coherence protocol is triggered to update the cache line in memory, thus leading to a response time increase due to memory latency implied by a forced cache miss. This results in crippling the performance of the two threads operating on mutual memory regions within a shared cache line. This instance is also reflected in Figure 2.12a.

A straight forward solution to avoid false sharing is the introduction of padding bytes as it can be seen in Listing 2.4, line 5. Here the array called *padding* has no productive purpose, it is just available to occupy another 32 bytes of memory to achieve cache line alignment. However, it is not guaranteed that the array will start at a 64 byte address boundary. Therefore it is necessary to align the beginning of the structure to the desired address boundary manually. This is done in line 6. The compiler attribute is dedicated to be used with the GNU C compiler[21] only. Nevertheless, on systems operating Microsoft Windows, one can use *__declspec(align(x))*[22] as prefix before the variable declaration to achieve an alignment on an address boundary of x bytes. As a result, the cache coherence protocol will not invalidate the cache line when parallel executed processes on mutual processors access two adjacent, but independent array elements.

Listing 2.4: Example code to avoid False Sharing

```
1 struct container {
2  unsigned int id;
3  unsigned char data[28];
4  /*padding bytes*/
5  unsigned char padding[32];
6 } __attribute__ ((aligned(64)));/*gcc-4.4.5*/;
```

Section 2.3 gave an introduction on parallelism from a software point of view, starting with Amdahl's and Gustafson's law, showing that it is not possible to achieve infinite performance gain by simply adding more parallel hardware. Afterwards, a cate-

[20]This does also apply to processes.

[21]`http://gcc.gnu.org/onlinedocs/` - last access 5/9/2012

[22]`http://msdn.microsoft.com/en-us/library/83ythb65.aspx` - last access 5/9/2012

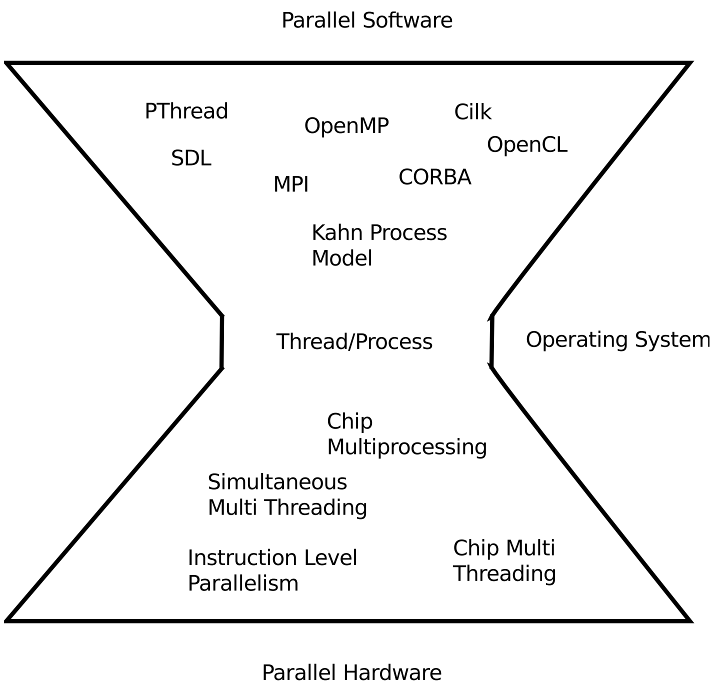


Figure 2.13: Hour Glass of Concurrent Computing [104].

gorization of several types of abstraction regarding programmer support in developing concurrent software. The presented models reached from languages providing a thorough abstraction of parallelism, to libraries exporting the full complexity of parallel programming to a developer. Subsequently, the topics synchronization & communication between parallel processes as well as memory consistency from Section 2.2 were re-examined from a software point of view.

2.4 Conclusion

Chapter 2 provided an introduction on parallel computing systems from a hardware and a software point of view. After a short historical abstract, the most important taxonomies regarding parallel hardware and parallel software have been introduced. It has been shown that there exist a vast number of methods to make hardware faster by using techniques like instruction level parallelism, multi issue pipelines, simultaneous multithreading, and chip multi processors, amongst others. From software perspective, there exist numerous methods of abstraction to aid programmers in writing software, exploiting the parallel computing power provided by hardware. However, almost all of these techniques end up by mapping the proposed technologies to a thread or a process. This kind of containing leads to the so called hour glass of parallelism [104], as it can be seen in Figure 2.13. This concludes the introduction on Multi-Core basics. The subsequent chapter presents a survey of challenges identified by researches regarding the teaching of concurrent programming.

CHAPTER 3

Challenges in Teaching Concurrent Programming

The previous Chapter 2 provided an introduction on the technical background of parallel computing with a focus on Multiple Instruction Multiple Data (MIMD) systems on a single computer. Furthermore, numerous singularities considering parallel computing were pointed out. These peculiarities have also been recognized by people teaching computer science courses. As a consequence this Chapter features a two-fold examination of a set of university and university of applied science courses in the context of parallel programming.

Since this thesis is written with a focus on course design in the field of multi-core programming a comprehensive overview on the field of university didactics is given in this paragraph. University didactics is part of the field empirical education research, which focus lies on topics regarding student learning behavior, their motivation, as well as their social environment and its impact on them [13]. Generally, it is possible to distinguish between the following sub fields of university didactics:

- One sub-field focuses on academic education in order to optimize a student's learning progress while taking classes [7].
- Another sub-field focuses on the qualification level of university lecturers themselves and deals with the question "What makes a teacher a good teacher?", thus

trying to find new didactical concepts and ideas to transfer knowledge from lecturer to student [86].

- A rather new sub-field in university didactic concentrates on universities as institutions of education, focusing on possibilities to optimize education as a processe in universities [87], [70], [68].

The examination of the university of applied science context multi core programming course and the university context multi-core programming course designs is divided into several sections. First, it is shown that the comparison of single courses within degree programs cannot be done on basis of the alumni *qualification profile*, but must be done on behalf the so called *learning outcome*. Subsequently, numerous contributions identifying issues and challenges in teaching parallel programming in a more specific manner are reviewed. Moreover, a set of dedicated parallel programming courses is evaluated with respect to their *learning outcome* definitions, as well as their actual teaching content. Finally, the evaluation results are presented in Table 3.1. They serve as a basis to derive best practice concepts leading to two generalized course descriptions presented in Section 3.2. These best practice course designs will be used as templates to propose a course in a university context and a course in a university of applied sciences context in the subsequent Chapter.

3.1 Qualification Profiles and Learning Outcomes

Due to the ever increasing diversity of higher education facilities and programs, the United Nations Educational, Scientific and Cultural Organization (UNESCO) approved an international standard classification of education (ISCED) [106], at their 29^{th} general conference in November 1997. The general idea of this standard is to create a

framework for the statistical description of national education systems, as well as a set of variables which are of high interest for policy makers in international education comparisons. Moreover it is the aim of this program to establish a

methodology that converts national education programs into a representation which makes them internationally comparable with respect to their levels of education and their fields of education.

Therefore, one of the UNESCO recommendations' aims is to improve student and alumni exchangeability [16]. As a consequence, the European Union (EU) and Australia, amongst others, have established comparable qualification level programs [4,23], which intend to implement the UNESCOs recommendations. However, in order to compare specific courses the concept of qualification profiles cannot be used. This fact is due to the the following:

- Qualification profiles are used to organize national education systems in a way to compare them internationally, as stated above [85].
- The so called *qualification descriptor* is used to establish a set of minimum requirements in order to reach a certain level within a qualification profile [46]. These levels are mapped to the respective degrees awarded after accomplishing a certain education. This includes traineeships as well as university degrees.
- The term pointing to to the smallest available organizational item within the UNESCO's framework is the *learning outcome* [45]. It describes the quantity and quality obliged to be communicated within a single course.

The idea of the *learning outcome* is to perform a paradigm shift from what the staff members teach to what the students should be able to accomplish after finishing this course. The process of designing a course with respect to the *learning outcome* can be viewed as an infinity loop and looks like the following:

1. The first step is to identify the aim of the course.
2. Second, the actual *learning outcome* is formalized.
3. The next step involves the creation of assessments and choosing the right assessment method. It is important to note, that the assessment method must be chosen after the *learning outcome* has been defined.
4. Afterwards, the threshold assessment criteria, should be defined.
5. Subsequently, the teaching concept and learning strategy is to be developed and checked against the *learning outcome* and assessment criteria.

6. The last step executes the course and triggers its re-evaluation after each run, starting at step *1*.

Therefore, after an identification of common parallel programming pitfalls in Section 3.2, the relevant courses will be compared on the basis of their defined *learning outcome* in Section 3.3.

3.2 Issues and Pitfalls in Parallel Programming

Parallel programming is a challenging task [33, 64]. Therefore, this Section reviews a set of contributions identifying and presenting solutions on several issues and pitfalls regarding concurrent thinking, and parallel programming. The insights used in this Section will be used in Section 3.6 in conjunction with Section 3.3, and Section 3.4 to derive two abstract course designs.

In [9], Ben-Ari et al. presented an overview on four parallel modeling frameworks in order to foster their student's understanding of certain concurrency concepts. The described tools are implemented in Java and feature distinct aims within teaching concurrent programming, mostly. *JBACI*, the first tool, is primarily aimed at teaching concurrent programming within a framework accepting *PASCAL* and *C* like dialects. The advantage of using a tool like *JBACI* lies in abstraction, as it does not overwhelm the parallel programming novice with operating system specific details. The second tool analyzed by the author is called *DAJ*. It covers basic understanding of distributed algorithms by visualizing private and global data structures. Additionally, Ben-Air et al. present a novice friendly model-checker called *CPV*, which allows students to get an introduction into formal verification of concurrent and distributed algorithms. Consecutively, the authors mention *JSPIN* as a Graphical User Interface (GUI) for the model checker *SPIN*, which can be used for the verification of distributed message passing systems. The latter three software packages are aimed at more abstract course designs dealing with distributed systems. However, the authors argue that these tools might help in lowering the entrance hurdle on parallel and distributed programming, due to their focus on the essential issues in parallel programming.

Identifying best practices and avoiding pitfalls while teaching parallel programming is the main aim of the article written by Joiner et al. [49]. In essence, the authors presented their experiences and insights in teaching parallel programming at various curricula at the courses offered by their faculty. Although most courses focused on high performance computing the authors derived a set of tips useful for every kind of courses dealing with concurrent computing. Since one argument for parallel programming is the ability to exploit computational power of parallel processors it is absolutely vital to show student a significant speedup when using parallel programs instead of serially implemented ones, so Joiner et al. . Nevertheless, in order to show speedup in a notable way, appropriate problem sizes must be addressed, as the usual small *getting started* problems would not suffice. Moreover, the authors argue that it is also important how speedup is represented. This may be especially important when it comes to novices in parallel programming. Here the authors suggest to use a visual representation rather than a textual one. Furthermore, the authors evaluated that using real world examples in teaching concurrent programming is far more motivating for students, than using purely academic examples like an efficient calculation of π. In conclusion, Joiner and his colleagues argued that the approaches presented in this paper could be verified by adapting numerous courses in the parallel programming context.

Robbins [84] presents an indirect approach to teach concurrent programming. He focuses on the visual representation of parallel behavior by offering a logging tool to his students. The logger is capable of tracing the timely occurrence of messages, events, and synchronizing tasks within a POSIX thread based framework. The tool is also capable of logging the communication in distributed environments. Robbins argues, that on the one hand, the logger should be used as a visualization tool for students, and an assignment checking tool for teachers. But on the other hand, the logger program shall be examined and partly changed by the students to foster their knowledge on parallel programming. Thus it serves as a teaching example. It is Robbins' intention to let the students learn and understand concurrency with the very same software they are using to test their solutions in order to understand possible interdependencies in a better way.

Considering the overall method of teaching concurrency Lamport [59] pointed out that most available lectures tend to confront students with programming languages, thereby distracting participants from the actual problem of understanding concurrency. Lamport argues that it is not necessary to actually implement a parallel program in an executable language in order to understand the properties and implications of concurrency, rather it is obligatory for students to really understand concurrency. According to the author this can be done best by teaching ongoing computer scientists and engineers to understand *"the most important concept in concurrency"*: invariance[1]. Therefore, Lamport suggests that a course on parallel programming is highly recommended to include a lecture devoted to the mathematical basics of computation.

3.3 Parallel programming courses

This Section provides a two-fold overview on a set of concurrent programming courses at universities and universities of applied sciences. Each course evaluation is structured in three paragraphs. The first paragraph introduces the general idea and topic of the course. Subsequently, the focus of the course is presented. The third paragraph describes the learning outcome of the currently observed course, where available. Information related to the course' learning outcome is denoted by black bullets(•).

Course 1: The first course examined is held as a long-term joint teaching experiment at the University of Illinois and the University of Maryland [51]. One the one hand, the course features an introduction of the shared memory programming library OpenMP and the message passing library MPI. On the other hand, the parallel programming framework XMT is presented to the students. XMT is primarily used to develop software for massive parallel multi threading processors and therefore provides several supporting tools to design and test parallel software. During the course the students were obliged to develop parallel algorithms in OpenMP, MPI, or XMT. Finally, the authors concluded that XMT was the best suited toolset for their purpose, because all students were able to achieve performance gains on a specific algorithm they implemented with XMT. On the opposite none of the students were able to improve the very same algorithm's performance when using OpenMP.

According to the course website[2], this course is held at an undergraduate level, therefore has a stronger focus on practical programming skills, than on conveying theoretical

[1]An invariant is a predicate to an argument that is true before its computations and is also true after its computation [59].

[2]`http://www.umiacs.umd.edu/users/vishkin/TEACHING/enee459p-f10.html`

information. The topics of the course are merely formed around the parallelism of searching and sorting algorithms.

The course's prerequisite list states the students have to have

1. basic knowledge about operating systems,
2. as well as data structures and programming.

The main aim of this course, and therefore its learning outcome is to

- raise the students' awareness of the parallel programming potential,
- as well as to teach them basic knowledge in certain programming environments (MPI, OpenMP, XMT).

Course 2: In their contribution in the field of teaching multi-core programming, Hansson et al. [39] present an embedded systems design course were students are obliged to perform a hardware/software codesign of a JPEG decoder within one semester. At the beginning of the course, participants receive a sequential implementation of the JPEG decoder algorithm. Moreover, they receive an introduction on a hardware/software codesign tool called HIVE. This tool provides aid in design space exploration, simulation and system composition. HIVE does also provide a backend for certain Electronic Design Automation (EDA) design tools like Xilinx[3] ISE. It is the students' responsibility to perform the hardware/software codesign and present their solution to class at the end of the semester. The used target architecture is a larger Xilinx Virtex4 Field Programmable Gate Array (FPGA) capable of serving two processors.

The aim of this course is to

- foster the participants' skills in embedded systems design, where the basic knowledge about hardware, software, and system development has been taught in preceding courses.

Moreover, the students are confronted

- with a larger engineering task, closely related to real-life engineering challenges.

[3]`http://www.xilinx.com`

Thereby, they

- simulate a real embedded systems project.

Besides the challenge of facing a larger project, the participants are obliged to reflect upon their design decisions when it comes to partitioning the algorithm on several processors and hardware accelerators.

Hannson et al. state that their course is the last disposition in a series of four related to hardware, software and embedded systems design. It is located in a later semester in the master curriculum of embedded systems engineering at the University of Eindhoven. Since this master program is a joint study of computer science and electrical engineering, the authors state that a course obliging the students to combine their knowledge of both fields seems to be the right challenge to test the knowledge taught in this master program.

Course 3: A more general approach is presented by Wolffe, Greg, and Treftz [108]. Here, the authors present a teaching framework for undergraduate computer science students who have never encountered parallel programming in their curriculum before. First an overview on several parallel programming paradigms is given, as well as a categorization of who might profit from a course featuring the respective content. Moreover, Wolffe et al. present several types of assignments as well as a set of hints for pitfalls novice parallel programmer might encounter when facing concurrent programming the first time.

The presented course framework shows that teaching parallel programming can be taught in many ways. Starting from using Graphic Processing Units (GPUs) as co-processors over developing parallel algorithms for scientific applications for the use in super computers, to exploiting the features of distributed databases by designing optimized queries.

Since Wolffe et al. provide a course framework, rather than a description for a dedicated course in a specific curriculum, the learning outcome is omitted.

Course 4: A completely different approach has been chosen by Ozturk [78]. The author describes a course using the commercially available virtual platform environment Simsic[4] to teach embedded multi-core programming. However, Ozturk's course focuses

[4]`www.windriver.com`

on hardware design considering cache hierarchies of symmetric multi-core systems, rather than on the software oriented challenges of embedded multi-core programming.

Nevertheless, Ozturk argued that a

- foster understanding of hardware related details in the field of multi-core programming offers students a
- different perspective on certain problems they might encounter when programming multi-core processors.

The course is offered to undergraduate and graduate students optionally. Nevertheless, participans have to have certificates in courses teaching hardware design and basic algorithm and data structures, as well as a theoretical background on parallel systems.

Course 5: In [79], Qingsong et al. present a remote laboratory for their multi-core programming curriculum. The authors provide two high-performance computers to their students which can be programmed using Intel's[5] multi-core programming tools. The computers can be accessed over the internet, where a dedicated computer is assigned to the student by a reservation system. The provided computers feature a dual and a quad core processor in order to let student experience behavioral differences, caused by the distinct architectures, in their parallel applications.

The aim of the distance multi-core laboratory is to provide students a possibility

- to foster their insights on parallel programming gained during class.

However, the course offers vendor specific tools and insights only.

Considering the course attendees learning outcome, no statement is made by the authors. However, regarding the used tools and vague descriptions in this paper it may be assumed that the students are at an undergraduate level, since it is mentioned that the course is located shortly after introduction courses dealing with programming and data structures.

[5] `http://www.intel.com`

Course 6: In their paper *"Some resources for teaching"* [34], the authors present a course focusing on dynamic formal verification[6] of concurrent programs and their properties. Moreover, a thorough introduction on posix threads and MPI software development is given.

The general idea of this course is to confront ongoing engineers with a thorough verification method for parallel systems, as conventional software debugging methods can hardly be applied for massive parallel software applications. By using dynamic formal verification, attending students learn techniques applicable in real world software engineering. Moreover, the students' reservation regarding formal verification shall be reduced by utilizing easy to use tools actually capable of verifying POSIX thread and MPI based C applications.

Considering the learning outcome, the authors intend to introduce their students into

- the basics of dynamic formal verification
- in conjunction with MPI and POSIX thread programming.

Since, this course is held at a graduate level, participants are obliged to understand the theoretical concepts and apply them to the concurrent programing libraries mentioned above.

Course 7: In 1992, McDonald presented a course about teaching parallel programming to undergraduate students without using truly parallel computing systems [65]. At that time parallel hardware was expensive and rare. Moreover, according to the author, compiler being able to construct inherently parallel code were costly goods. Nevertheless, the paper presents experiences in teaching parallel programming using the languages Linda and Joyce. However, Linda and Joyce do not require parallel hardware to be executed, rather they are simulating concurrency by using their own time base. In order to visualize concurrent behavior a graphical component showing the parallel execution of tasks is available.

[6] *"Dynamic verification checks invariants at runtime and can thus detect runtime physical errors. It differs from static verification, which checks that an implementation satisfies its design specification. Static verification can detect design bugs but, by definition, cannot detect runtime physical errors. The two approaches are complementary"* [67].

In 1992 parallel computation was mostly a scientific or large company application. Therefore, the learning outcome of this course was designed

- to provide a basic understanding of the properties and implications upon parallel programming decisions.

The course was placed in an undergraduate curriculum in the first semester.

Course 8: Rague depicts in [81], that many undergraduate courses teaching parallel programming focus on synchronization and communication concepts, but disregard the necessity of procuring parallel thinking. Therefore, the author adapts a computer science course in a first semester undergraduate curriculum to place the concept of concurrent thinking as early as possible in his students' minds. Due to the fact, that fully featured parallel programming libraries, or dynamic formal verification tools, may overburden programming novices, Rague introduces the graphical parallel analysis tool (PAT). PAT does not require any specific lexical programming skills, but uses a petri net based representation of a program flow. According to the author, this way of visualizing program flows showed to be effective when teaching novice computer science students.

Rague describes his approach as a way to offer students a course of parallel thinking at a very early age in their computer science education. The course's learning outcome is defined to teach

- basic programming concepts, as it is an introduction course in computer science,
- as well as to provide participants with a basic understanding of parallel thinking.

Course 9: In [52], Kessler argued that most textbooks regarding algorithms and data structures for second year computer science students focus on sequential algorithms, but lack the discussion of parallel ones. Nevertheless, there exist textbooks dealing with both topics, but due to the complexity of sequential and parallel systems in general, there existed no single textbook covering both topics sufficiently in 2009. Therefore, the author presents an approach using a *"simple parallel programming model"*. Kessler's approach describes a course using the bulk synchronous parallel model (BSP), which is a message passing based model, to teach parallel thinking and programming within a special framework called NestStep. However, the author omits the very common shared memory model due to its simplicity. Nevertheless Kessler mentions that he uses the shared memory model PRAM for teaching concurrent programming in a graduate course.

Kessler presents a course aimed at second year computer science students with sequential programming experience. The learning outcome of the course is to teach

- students the fundamentals of concurrent concepts
- as well as their implications using a framework specially designed for parallel program investigations within message passing systems.

Course 10: Introducing interested high-school students to parallel and distributed programming is Ben-Ari's [10] aim. The course is split up into mandatory and optional sections. The mandatory parts cover basic programming, algorithms, and data structures. Afterwards, the students may decide between sections dealing with topics assembly programming, logic programming, and concurrent programming. In order to introduce concurrency issues like synchronization and model building, Ben-Ari lets the students dramatize a synchronization task of two robots obliged to sweeten a glass of juice. This is done in several steps. First, he lets the students find out whether the given algorithm works correctly or not. Later the author asks the students to find a solution to this problem and test their findings by dramatizing them in class.

The aim of Ben-Ari's course is not to teach parallel programming to high-school students, but to make them aware on parallel problems. Moreover, the learning outcome of the course formulates that students should be able to

- identify systems with parallel properties, as well as
- discuss solutions to this problems in teams.

Course 11: In [17], Bynum and Camp present a concurrent programming course using Ben-Ari's concurrent interpreter (BACI). BACI descends from PASCAL-S, a dialect of the programming language PASCAL, supporting parallel constructs. Basically, the course offers a standard introduction into parallel programming using several versions of semaphores and a monitor to solve synchronizations tasks. Nevertheless, BACI is said to have an easy to learn syntax and offers a lot of synchronization techniques to be taught.

The leaning outcome of this course has been identified to provide students with a

- fairly complete overview and introduction on synchronization methods for parallel programming.

Moreover, the didactic concept of the course states that at the beginning students receive a lot of help by documents and hints to solve their assignments. Consequently, this aid is reduced during the course, where students shall be able to solve a parallel problem on their own in the end. Therefore, stating that another learning outcome of Bynum's lecture is provide participants with

- the knowledge to solve synchronization tasks upon their own.

3.4 Assessment Concepts

As assessments are a part of each reasonable course, this Section presents a set of assessment methods obliging the student to perform constantly, rather than using their short term memory while learning for a final exam. Basically, the presented assessment techniques implement automated assessment, student peer review, a project and a conference like approach.

Since teaching large groups of students imply that a large number of assignments have to be assessed the need for automated assessment methods in the field of computer science arose [21]. Despite of the positive economic perspective when using automated assessment methods teachers face the fact that today's students are creative in using today's technology to cooperate in their home assignments, although this might not be intended by the lecturer. Therefore, automated assessment might not be the only technique one may want to use when assessing programming homeworks. Thus, it is recommended to pick random students which should explain their solution thoroughly to a lecturer.

In order to provide students with new insights when participating in a programming course, Sitthiworachart et al. present an approach where students are able to assign marks to their colleagues [91]. This works as follows: Each programming assignment is awarded with a number of points. Fifty percent of the overall achievable points are assigned by an automated assessment tool, essentially testing whether the actual output matches the desired one. The remaining fifty percent of the grade are awarded by a randomly chosen fellow student. To execute this approach, there exists a firm grading scheme the students have to follow. Moreover, the students grading is done anonymously. In essence th idea behind this assessment method is to foster the students understanding of the programming task by reading their colleagues code.

In [63] the authors present an approach for using active assessment methods in a course dedicated to algorithms and data structures. Although, the course is focused on a different topic, the principle assessment strategies may also be of value in another context. Malmik et al. suggest to use a combined assessment method consisting of the following components to determine a participants final grade.

Peer Review This assessment technique is used to let the students review their colleagues solutions. The correctness of the review is not accounted for the final grade. However, the review's quality in terms of thoroughness is evaluated by a teaching assistant. The author's intention to introduce peer-reviewing was to give students insights into the approaches their colleagues have made in order to foster their understanding of the taught material. Peer Review is carried out as a single job.

Revision The method revision is used to check whether a student has engaged herself with the home assignment. Here at the beginning of class a number of students are obliged to defend their solutions against the arguments of a lecturer. This assessment method is included in the final grade. Revision is carried out as a single job. Nevertheless, only a subset of the participating students will be assessed within a revision.

Final Examination The final examination is obligatory by every student and covers the topics covered in class in a written exam.

In order to assess, whether their student were able to achieve the learning outcome of their course, Normak et al. [73] introduce an examination concept called *Mini Project*

Programming Examination. Basically, the concept bases upon two phases, from a students perspective. First, small sized programming examples which take about 48 hours to implement for each member of a small student groups assigned. In this phase the course participants are obliged to solve a real world problem of reasonable size on their own as a group. In the second phase the students have to participate in a review held by a lecturer. Here the group must present their solution on the one hand. On the other hand, each member of the group has to answer questions dedicated to herself. According to the authors, the idea of examination through projects has the following advantages:

- Students work on motivating real world example covering the topics taught in the course.
- Participants are obliged to do team-work, which is a highly recommended skill in general.
- Although participants will present their project as a team, each member is required to know the whole program in detail, therefore, it is possible to check if all students have reached the learning outcome of a course using this approach.

3.5 Evaluation Results

This Section provides the results of the evaluation of a number of parallel programming courses with respect to their learning outcome, as described in Section 3.3. The overall evaluation results can be found in Table 3.1. The table consists of the following fields with their respective meanings:

Ref. # This column references the respective course described in Section 3.3. Thus **Course 4** in Section 3.3 points to Ref.# 4 in Table 3.1.

Level This field describes whether a course is held at a graduate level (**G**), or an undergraduate level (**U**). Nevertheless, there exists the optional modifier (**?**), denoting that it is assumed that the course it taught at that level, although the cited paper did not state it explicitly.

Context The context field shows the course's focus. Generally, a course can have two context descriptors and an optional modifier. The context descriptor **PS** shows that this course has a focus on programming skills, where **TC** denotes that the course tries to communicate the theoretical background of concurrency in a stronger way. In case a course teaches a theoretical and a practical focus, the modifier **+** may indicate an emphasis on this particular focus. The modifier **-** may indicate that this focus is disregarded, respectively.

Used Tools This column lists the libraries, special programming languages, compiler extensions, simulation tools, etc. that are used throughout the respective courses. See the actual course in Section 3.3 for a short description of each tool.

Topics Here, the topics covered by the respective courses are listed. **App** denotes that parallel software development is taught with a focus in general application programming for CMP, or parallel systems in general. The acronym **HPC** means high performance computing, referring to programs and algorithms used in a scientific environment (e.g. climate model simulations on super computers). Finally, the shortcut **E** covers embedded applications of parallel systems including hardware related topics (e.g. multi-core mobile phone platforms)

Learning Outcome This column provides a short overview of the learning outcome defined in the course' description with respect to their context, used tools, and taught topics. A more detailed explanation may be found in the respective course overview in Section 3.3.

As it can be seen in Table 3.1, 8 out of 11 reviewed courses are aimed at undergraduate curricula solely. One course is taught to graduate and undergraduate students, where each participating student had to attend in fundamental courses about concurrent systems. Although 8 of 11 courses had a focus on the theoretical concepts of parallel systems, only 3 courses placed their spotlight at parallel systems after the general topic introduction. The most common tools throughout the examined courses were MPI, PThreads, and custom tools made for teaching parallel systems theory and programming. Nevertheless, 5 of 11 tools were custom ones. Finally, 9 parallel application courses cover the field of high performance computing. Only 4 courses tend to cover embedded applications. However, only 1 course actually teaches embedded system in the parallel programming context.

Nevertheless, it is interesting to see that more than 70% of the examined courses where held at an undergraduate curriculum. A supportive argument for teaching concurrent programming to lower level undergraduate students has been made by Feldman [24]. Here, the authors provide an empirical study on the feasibility of teaching multi-core programming to lower-level undergraduate students, which is also supported by Ben-Ari [10] teaching concurrency at a high-school level. In conclusion, both author groups argue that confronting students with parallel thinking early and often in their curriculum results in a better understanding of the general concepts of parallel systems.

3.6 Best Practices

The preceding Sections, Section 3.2 to Section 3.5 provided an overview on the challenges and solution concepts arising when teaching parallel programming. It is the purpose of this Section to provide two generalized course descriptions on the basis of best practices examined in former parts of this Chapter. The descriptions will be structured as follows. First, an abstract structure of the methods used to construct an introduction and motivation lecture is given in **(1)**. Next, the used teaching framework is presented in **(2)**. Finally, the applied assessment method is described in **(3)**. Finally, a generalized learning outcome for both of the courses is given. Each outcome is marked with a black bullet (●).

In order to identify both general course descriptions within this Section, the first course is denoted **(A)**, where the second one is identified with **(B)**. Both courses will anticipate concepts that oblige the students to primarily work on their own, however, Course **(B)** will offer a greater degree of freedom to its participants. Since both courses incorporate an introduction, the first paragraph is valid for courses **(A)** and **(B)**. A compact summary on the structure, concepts and assessment methods propagated to form the two course concepts in this Section is given in Table 3.2

(A1/B1): Since Ben-Ari [10], Hansson [39], and Robbins [84] argue that an introduction tailored to the expectations and already gathered experiences with certain subjects lead to a greater acceptance of the taught material from a students perspective, the introduction can be considered as one of the most important parts of a lecture. Therefore, it is recommended to provide an introduction that uses striking examples in combination with information students may already know from their daily life, in order to make a course appear more interesting to them in the first place. In the case of parallel programming a pure speedup factor of a parallel algorithms' run time compared to a sequential implementation may be to cumbersome from a students point of view. Instead a more figurative example like a video stream being decoded slowly in a sequential implementation, but being shown fluently in a concurrent one, may be a good example. Moreover, it is advised to use real world examples, students are familiar with when introducing a new topic. In the case of concurrent systems, a recent mobile phone may be used as an application for parallel hardware for example. Parallel software could be exemplified with a recent graphics engine, or an acceleration method for video codecs. Nevertheless, according to Lamport [59], it is absolutely vital to introduce students to the theoretical concepts behind concurrency. Therefore, it is recommended to show students the basics of these concepts [20], as well as the abstraction of computation in states [43].

(A2): Considering the actual course content, [34, 79, 108], and [51] endorse the usage of industrially relevant parallel programming libraries in order to let students work with languages they are familiar with. This is especially important in course designs having a strong focus on theory lectures and practical home assignments. Nevertheless, Joiner [49] supports the use of real world programming examples as home assignments, or projects, rather than academic ones, where the students cannot directly benefit from the actual problem they solved. Furthermore, it is obliged that side effects like false sharing, must be repeatable within the laboratory by the students. Otherwise, possible real pitfalls might be labeled as unimportant by course attendees [79].

(B2): Targeting course designs with a large student contribution, for example courses with a seminar like character, Joiner [49], and Qingsong [79] recommend a thorough set of getting started documents and tool documentation. Moreover, the authors advise to use distinct lectures to propagate the projects students may pick to work on throughout the course. Additionally, it is recommended to offer a heterogenous set of topics. This should be done to let the students get a wider insight into the field of concurrent systems [78, 108].

(A3): When teaching firmly organized courses with dedicated home assignments, Normak [73] recommends examination through mini-projects. This approach is also applied by Ozturk [78], and Wolffe [108]. Thereby, mini-project examination works like the following. The students are divided into groups, where programming tasks are assigned to each of the groups as a home assignment. Usually, such a home assignment is of smaller size. The examination takes place in such a way that the students have to explain and defend their solution on behalf of a lecturer. Every student is obliged to know the whole program, thus it is possible to ask every student a random detail of the program. Therefore, every group member has to contribute in the solution, or at least be able to explain it.

(B3): When teaching courses in a seminar like setting one could hand out the topics to the students and wait until they return their seminar papers at the end of semester. However, this approach is not encouraged by Malmik et al. [63]. Instead the authors propose to simulate the execution of a conference within the seminar. This would include the composition of a seminar paper to a specific topic with the course's field. In the next step participants must hand in their papers in an anonymized form, thus omitting authorship. Now lecturers assign these papers back to other students to perform a review by means of a structured review form, similar to those used in conferences. Afterward, the reviews are checked for thoroughness by the lecturers. In the following, the feedback given by students must be considered by the paper's authors. Eventually, each seminar paper is presented and discussed in class.

As mentioned in Section 3.1 it is very much recommended to provide a learning outcome for each course description in order to conserve comparability. In the case of the general course concept based on best practices presented in this Section, the learning outcome for Course **(A)** may look like the following: Students are able to

- understand and describe the theoretical concept of concurrent systems.
- name and describe the most common parallel hardware architectures, as well as being familiar with a set of software libraries exploiting their capabilities best.
- port a small sized sequential program to a parallel one where applicable.

Considering course **(B)**, the learning outcome can be defined as follows: Students are able to

- understand and describe the theoretical concepts of concurrent systems.
- explain a formal verification concept of parallel systems.
- name and describe the most common parallel hardware architectures, as well as show that they are being familiar with a set of software libraries exploiting their capabilities best.
- In addition, participants have engaged themselves thoroughly with a specific topic in the field of the course, and are able to explain and understand the key concepts of their course contribution.

3.7 Summary

Chapter 3 served as the foundation for providing the necessary theoretical and practical background to derive two generalized course concepts in the field of concurrent computing systems. First, it has been shown that course programs must be compared at the *learning outcome* level, rather than the qualification profile, which denotes the *learning outcome* of the whole curriculum so to speak. This has been done in Section 3.1. Afterwards, a set of contributions identifying issues and pitfalls when teaching concurrent programming is presented in Section 3.2. Subsequently, 11 courses dealing concurrent programming in various ways have been examined and compared with respect to their *learning outcome* in Section 3.3. A summary of this evaluation can be seen in Table 3.1. As a consequence numerous assessment concepts have been presented in Section 3.4. As a conclusion regarding the previous Sections, two generalized course designs, based on identified best practices are proposed in Section 3.6.

Table 3.1: Results of Mult-Core Programming Course Examintaion

Ref.#	Level	Context	Used Tools	Topics	Learning Outcome
1	U	PS	OpenMP, MPI, XMT	App, HPC	Basic programming skills in used libraries; Raise awareness of parallel systems and their capabilities.
2	G	PS	Hive, From Scratch	E	Being able to partition and design a realworld application; Use and combine knowledge of predecessing courses about parallel computing theory and practice.
3	U	TC-,PS	GPU, MPI, OpenMP, PThreads	E, HPC, App	Provides a course framework; Learning outcome differs. In general the courses provide a focus on practical skills (e.g. CPU/GPU programming) and theoretical concepts (e.g. understanding and evaluation of distributed database algorithms).
4	U,G	TC-,PS	Hardware	E	Understand and use different cache architectures in distinct multi-core designs for different problem descriptions.
5	U?	PS	Intel TBB	HPC, App	Use the Intel TBB library to develop parallel programs.
6	G	TC+,PS-	PThreads, MPI	App, HPC	Understand and use dynamic formal verification to verify message passing and threaded applications. Develop MPI and PThread based software.
7	U	TC	Linda	HPC	Understand the basic principles of parallel hardware and software, as well as their implications.
8	U	TC	PAT	App, HPC	Understand the basic concepts of parallel thinking.
9	U	TC+,PS-	NestStep	App, HPC	Understand the concepts of concurrency, as well as their implications when used in a computer program.
10	U	TC,PS-	Discussion, Dramatizing	HPC, App, E	Raise awareness for concurrency and concurrent processes in general. Detect and describe situations where synchronization upon a critical section is obligatory.
11	U	TC+,PS	BACI	App, HPC	Understand, describe, and use synchronization methods common in concurrent programs. Use the most appropriate synchronization method for a given problem.

Table 3.2: Abstract Course Overview

	Course A	Course B
Learning Outcome	Students understand and can describe the theoretical concepts of concurrent systems. The participants are able to describe the most common parallel hardware architectures and are able to utilize one concurrent programming library to adapt a small sequential program to a parallel one, where applicable.	Students understand and can describe the theoretical concepts of concurrent systems. They should know the key principles of a formal verification technique in the filed of concurrent systems. The participants are able to formulate the key concepts and principles of their seminar paper coping with a thorough altercation of their topic within the field of parallel systems.
Introduction & Motivation	Tailored to the knowledge base and expectations in the first place. Use striking examples in conjunction with knowledge students are already familiar with (Smartphones, multi threade gaming engines,...).	
General Setup	Small real world problems are likely to be faced with more interest than possibly exciting academic problems. Therefore, mini projects on rewriting small, but useful sequential programs, into parallel ones seems promising.	In this concept the execution of a conference shall be simulated, where each student picks his topic of interest within the course field to write a seminar paper. After a reviewing process (done by course participants), the papers are presented and discussed in class.
Assessment	Each project group is examined by a lecturer. Each member of the group should know the whole program in detail, thus being able to answer question about specific code constructs from an architectural and programming technique point of view.	The assessment consists of two parts. First, the quality of the peer reviews is taken in to account. Moreover, the thoroughness of the review is addressed. Second, the students presentation and their seminar papers are taken into account.

CHAPTER 4

Two Approaches for Teaching Multi-Core Programming

Based on the results of Chapter 3, Chapter 4 will present the actual implementation of the abstract course descriptions **(A)** and **(B)**, which were presented in Section 3.6. The first course described in Section 4.1 will be held in a university context, whereas the second approach is described in Section 4.2. Although both courses intend to provide an overview on programming modern multi-core systems, they differ significantly in the way this should be achieved. The first approach discusses an approach in a university context.

4.1 Multi-Core Programming in a University Context

This Section provides a detailed description of course **(B)** from Section 4.1. The course will be held as a seminar, thus having only a small number of lessons where participants are obliged to attend. In essence there will be three sessions with the following content:

1. The first session will consist of the following three corner stones. First, participating students shall be made aware of the timeliness of the topic. This will be done by a talk on the current use cases of parallel computing systems and discussion in class, where each student is invited to present his personal experiences and

views on concurrent systems. Afterwards, the organization of the lecture, including assessment and conference like format will be presented. Finally, a theoretical overview on parallel systems from a hardware, software and formal point of view will be given.

2. The second lecture will present a set of sub-topics to the students. The topics will range from practical programming tasks to theoretical elaboration. At the end of the lesson students will be assigned a given topic (in groups of two, depending on the actual number of participants). In case there are too many participants the group size will be increased. As a consequence the thoroughness of the task to be solved will be raised as well.

3. The last session will give each group the opportunity to present the results of their findings. Every group member has to present something. This may include live demonstrations of programs. The presentation and the subsequent discussion will be taken into account for the final grade.

Each of the described lectures will have a length of 90 minutes, except for the last one which shall be scheduled for at least 180 due to the presentations. A participant's final grade consists of a review of his seminar paper, the presentation of his seminar paper and the quality of his peer review of a fellow student's seminar paper. Moreover, the peer review her received (see Section 4.1) is taken into account. The course's key facts can be seen in Table 4.1.

Table 4.1: University Course Key Facts

Structure	# of Sessions	Attendance	Length of Session	Assessment
seminar	3	compulsory	90 (180) minutes	seminar paper, presentation, review quality & received reviews

Although the course has only 3 sessions of attendance there exist 3 additional deadlines. A deadline miss results in a grade reduction of one level.

- The first deadline refers to the hand in of a student group's seminar paper for peer review. This hand in should be done anonymously. The seminar papers are immediately redistributed to fellow students, where each student shall review 2 seminar papers when possible (depending on the number of participants).
- The next deadline refers to return the seminar paper reviews to the lecturers. Subsequently, the reviews are redistributed to the respective paper authors.
- The last deadline requires the course attendees to hand in the final camera ready versions of their seminar papers. This final version will be used for grading.

As students might not be familiar with the concept of peer reviewing, as well as the general structure of such a review, an example peer review form is supplied by the lecturer. The form can be seen in subsection *"Peer Review Form"* on page 65. It is based on a form for a student conference at the Umeå University [105] in Sweden. As it can be seen, the review form consists of a header requiring the reviewing student to fill in the title of the paper he is investigating, as well as his name and student ID. The latter information are collected since the review form is part of the final grade. Right after the personal information table, there exists a paragraph explaining the organization of the review form, as well as the meaning of the points that have to be assigned to each of the subsequent rating parts. These rating parts are subsidized into three categories:

PART I - Formal criteria This part covers a review with respect to writing style, paper organization and transitions between sections, subsections and paragraphs. The purpose of this part is to show students that the organization of an article is just as important as its content.

PART II - Qualitative criteria In this part, the reviewer is obliged to examine the paper from a qualitative point of view. Here, facts like thoroughness, proper citations, and critical evaluation of found facts and results have to be taken into account. The intention of this part is to animate reviewing students to examine and evaluate their colleagues work carefully, thus having a general idea of their fellow students work during the final presentation phase in the last sesssion.

PART III - Suggestions for improvements Here, participants should provide a short summary of the paper's content with their own words. Moreover, they should give hints for improvements based on their review results of parts I and II. Moreover, there exists a box dedicated to special comments that are only assigned to the lecturers and are not forwarded to the paper's author. This box can be used to point out suspicions about plagiarism for example.

Peer Review Form

Title of reviewed paper:	
Name of reviewer:	
Student ID:	

The purpose of this review form is to aid and guide during the peer-review process. It consists of three parts. The first part is used to check whether the reviewed paper has considered the formal criteria, like proper paper organization, writing style and transitions. The second part takes qualitative aspects into account. This includes thoroughness of the analysis and the line of argumentation for example. Here you have to assign between 1 to 5 points to each of the listed criteria:

- 1 denotes bad coverage.
- 3 refers to moderate coverage.
- Finally, 5 points are rewarded if the coverage is excellent.

You may reward 2, or 4 points if you feel the coverage is in between. The third part will hold your personal comments and suggestions for improvements. Moreover, a short summary on the reviewed paper is obliged in order to see whether you have read and understood the article in principle.

PART I - Formal criteria

Writing Style (The paper has only few spelling mistakes, no blown up phrasing, no slang. Furthermore, all terms and acronyms are explained properly)	
Paper organization (The article is well structured into sections and subsections. Moreover, it uses tables and figures where appropriate)	
Transitions (The author made smooth transitions between paragraphs and sections.)	

PART II - Qualitative criteria

Contribution (The idea is to teach people something new, thus summarizing only a single paper is insufficient.)	
Citation (There exists a reasonable number of different, trustworthy and correct citations. (NO WIKIPEDIA))	
Thoroughness (The text is not simple minded. Instead it approaches the problem from different points of view.)	
Critical evaluation (The author does not simply accept existing work. He identifies and discusses possible weakness.)	
Correctness (The presented work is correct, own ideas and third party ideas are clearly separated an well cited.)	

PART III - Suggestions for improvements

Summary: (A short summary of the paper, written in your own words.)

Suggestions for improvements: (Give hints on how the author could improve his paper.)

Additional comments (not forwarded to the author): (This field may contain special remarks on the paper that are not suited for the author. (e.g. suspect to plagiarism))

By now, the overall structure of the multi-core programming course in a university context has been explained in general. The following Subsections will provide a detailed overview of the course, starting with an overview time table (see Table 4.2). This table reflects a possible implementation of the course within one term from a student's point of view. In the first column the name of the respective lecture, or deadline is shown. Column two provides a summary of the lecture's content. The last column reveals a timely offset from the previous event. Thus, if the column says *7 Days* it means that this lecture/deadline is intended to be scheduled seven days after the prior one. In addition to Table 4.2, Table 4.3 presents the course' schedule from a lecturer's point of view. In general the columns in Table 4.3 have the same meaning as in Table 4.2, except for column one which also represents administrative tasks a lecturer has to perform during the course. In order to get a better overview on the timely organization of the course see Figure 4.1. The length of brackets indicating passed time are not to scale.

Table 4.2: University Course Time Table from a student's point of view

Lecture/Deadline name	**Content**	**Offset**
Lecture 1	Motivating examples on the importance of parallel computing; Theoretical introduction based on ex-cathedra; Explanation of organizational structure; Handout of fundamental papers aiding in choosing a topic of interest	
Lecture 2	Talk about the available topics; Students have to get together in groups of two or more (depending on the participants to topic ratio) and pick a topic	7 Days
Deadline 1	Students have to hand in their anonymized seminar paper on their chosen topic;	5 weeks before end of the course
Deadline 2	Participants have to hand in the filled in review forms.	7 Days
Deadline 3	Students have to hand in the camera ready version of their paper reflecting the suggestions for improvements imposed by their colleague's reviews.	14 Days
Lecture 3	Participants present their papers in last lecture; They are encouraged to discuss their findings with their colleagues and lecturers after each presentation.	14 Days

Table 4.3: University Course Time Table from a lecturer's point of view

Lecture/ Deadline/ Task name	**Content**	**Offset**
Lecture 1	Motivating examples on the importance of parallel computing; Theoretical introduction based on ex-cathedra; Explanation of organizational structure; Handout of fundamental papers aiding in choosing a topic of interest	0 Days
Lecture 2	Talk about the available topics; Students have to get together in groups of two or more (depending on the participants to topic ratio) and pick a topic	7 Days
Deadline 1	Students have to hand in their anonymized seminar paper on their chosen topic;	5 weeks before end of the lecture
Distribute seminar papers	Assign assign at least two seminar papers to each student to start the reviewing process. Send the properly anonymized paper and and the review form.	24 hours
Deadline 2	Participants have to hand in the filled in review forms.	7 Days
Redestribute Reviews	Short review of student reviews. Redistribute reviews.	1 Day
Review reviews	Review the received reviews and cross check with handed in papers. Grade reviews.	13 Days
Deadline 3	Students have to hand in the camera ready version of their paper including the suggestions for improvements imposed by their colleague's reviews.	14 Days

Continued on next page

Table 4.3: University Course Time Table from a lecturer's point of view

Lecture/ Deadline/ Task name	**Content**	**Offset**
Review camera ready version	Check whether the handed in camera ready version has considered the reviewer's suggestions for improvement, if they were useful. Grade the seminar paper.	14 Days
Lecture 3	Participants present their papers in last lecture; They are encouraged to discuss their findings with their colleagues and lecturers after each presentation. Grade the presentation	14 Days
Final grading	Take all accomplishments into account (seminar paper, review, presentation) and compute the final grade A detailed weighting of each grade component can be found in Table 4.4.	1 Day

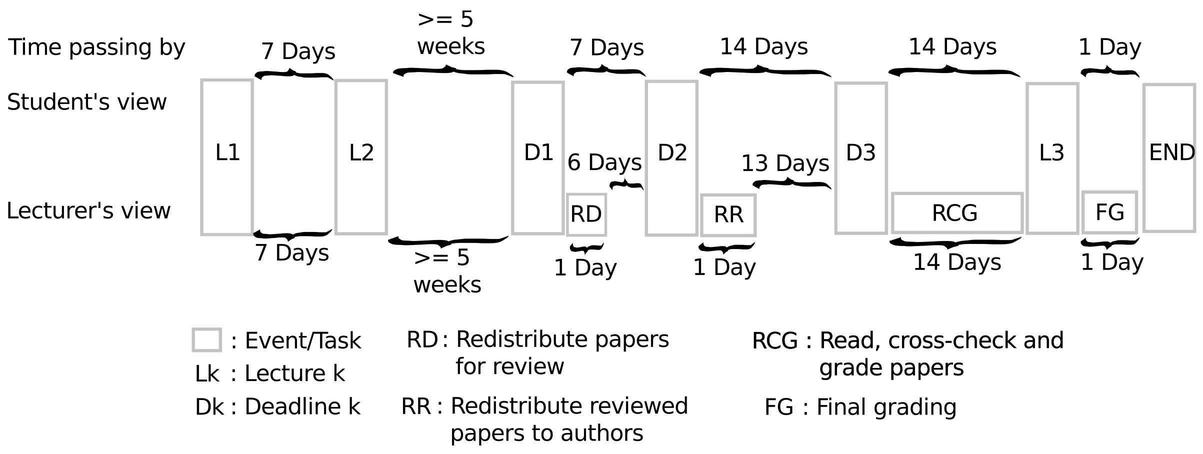


Figure 4.1: Multi-Core programming in a university context - detailed course timeline

Table 4.4: Weighted grade components

Component	Weighting in %
Seminar paper	40
Review quality	30
Final presentation	30

The course's learning outcome is based on the learning outcome description of Course **(B)** in Section 3.6 and is precised in the following way:

- Students understand and can describe the theoretical concepts of concurrent systems.
- Participants gain further insights on a topic of their choice within the field of multi-core programming.
- Furthermore, participants are able to name and describe the most common parallel hardware architectures, as well as phenomenons like false-sharing for example.

This concludes the structural and organizational overview of the university context multi-core programming course. Nevertheless, in order to have a complete course description the subsequent Section 4.1 will present a number of fundamental papers dealing with concurrency and parallel systems. These articles are recommended to provide students with a straight forward introduction on the topic in general. Based on these papers, Section 4.1 introduces a set of possible assignments, that might be offered to students participating in this course.

A selection of recommended papers on parallel systems

This Section presents a set of articles that might be offered to students in order to prepare themselves for their actual assignments. Some of these papers are recommended to be read compulsory, others can be interesting if a group of students chose a specific topic for their seminar paper. Each paper is described upon the categories *Author*, *Title* and

Summary. At first we will present those papers which are of interest for every course attendant:

P(1): Dijkstra, Solution of a problem in concurrent programming control [20] This paper can be considered as the beginning of research on parallel system in general. The author thoroughly identified every necessary property to guarantee a mutual exclusive synchronization of an arbitrary number of processes/processors on a shared memory region. Moreover, Dijkstra provided an algorithm satisfying his identified properties, as well as proper correctness proofs. The article has been written in 1965.

P(2): Amdahl, Validity of the single processor approach to achieving large scale computing capabilities [6] Gene M. Amdahl showed in 1967 that a linear increase of the number of used processors does not result in the expected linear speedup. However, the motivation for his paper was the notion that single processor machines would soon be to slow to cope with their intended workload, thus multi processor computers were to be the future of fast computation.

P(3): Flynn, Some Computer Organizations and Their Effectiveness [29] In 1975 Michael J. Flynn presented a categorization of processor architectures in order to organize the various proposed and existing processor architecture types at that time. His organization takes only instruction and data streams into account, thus being generic enough to be used throughout time. Today it is known as Flynn's Taxanonmy.

P(4): Skillicorn, Models and languages for parallel computation [92] This paper introduces a model to categorize parallel programming languages, libraries an optimizing compilers. This categorization may aid in choosing the right tool for the right task.

P(5): Adve et al., Memory models: a case for rethinking parallel languages and hardware [2] This article presents an overview of the memory models of industrially relevant programming languages and their feasibility for parallel programming with respect to hardware memory models. Moreover, the languages are also examined with respect to their synchronization capabilities. Amongst the covered languages are Java, C and C++.

P(6): Bolosky et al., False sharing and its effect on shared memory performance [14] Since false sharing is the cause for a common performance bottle neck when programming shared memory multi-core systems this article shows the reasons for this phenomenon and provides practical solutions to it.

P(7): Adve et el., Shared memory consistency models: a tutorial [3] Due to to fact that a reasonable part of today's processor performance relies upon cache memory hierarchies, this article provides a tutorial on cache memory architectures from a software engineer's point of view.

P(8): Lee, The problem with threads [60] Threads are a well known technique to implement parallelism in software engineering. However, the largest issue with using threads is non-determinism. In this paper the author argues why threads should be replaced by other design pattern implementing parallel software and programming languages.

Based on these introductory articles in the field of parallel systems, the following contributions are categorized by the possible field of interest a course participant might have. The first group of articles belongs to theoretically interested students who want to learn more about the design, specification and reasoning of concurrent systems in a state based computation model.

P(9): Lamport, Verification and specification of concurrent programs [57] This paper provides a historical overview on state based concurrent program verification, as well as an overview on the methodology in general. It may serve as an entry point to the topic.

P(10): Lamport, The mutual exclusion problem: part I - a theory of inter process communication [55] This article describes the formal problem of mutual exclusion, as well as a model providing a formalism to reason about these problems. It is the first part in a series of two.

P(11): Lamport, The mutual exclusion problem: partII - statement and solutions [56] This is the second part of Lamport's series on the mutual exclusion problem. It deals presents a rigorous problem statement as well as a set of solutions for solving the mutual exclusion problem in a distributed memory model multi-core system.

Students who are more interested in practical topics may address the following papers providing an overview of libraries, languages, and hardware support to master the parallel computing power of today's multi-core processors.

P(12): Sato, OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors [89] This article gives an overview on the parallel programming library OpenMP. OpenMP has been designed to aid in adding loop parallelism to sequential programs. This is done by source code annotations. The compile does the mapping to threads in the end.

P(13): Graham et al. Open MPI: A Flexible High Performance MPI [35] MPI is an abbreviation for Message Passing interface, thus it provides a library for synchronization and communication in distributed memory architectures. This paper provides an introduction to this library.

P(14): Croix et al., Introduction to GPU programming for EDA [19] Since graphic adapters are actually massive parallel multi-core computer systems they do a lot more than just performing graphic computations. This paper provides an overview on the computation capabilities of GPUs.

P(15): Nickolls et al., Scalable Parallel Programming with CUDA [72] CUDA can be used to tell graphic adapter from NVIDIA that they should calculate other things than pictures. This article shows how this can be done in principle.

P(16): Nottingham et al., GPU packet classification using OpenCL: a consideration of viable classification methods [74] OpenCL is similar to CUDA. It serves as an interface to use a GPU as a coprocessor, thus having a heterogeneous computing system. Nevertheless, OpenCL is an open standard. This article shows how to design and implement a network packet filter by utilizing a GPU.

Participants who are interested in the application of multi-core systems in hardware oriented scenarios may be interested to the following articles in the field of embedded systems.

P(17): Grant, Overview of the MPSoC design challenge [64] This paper provides an overview of the multi-core design challenge when building embedded systems on chip (A whole computer system including CPUs, GPU, network controllers, etc. on one die.).

P(18): Gschwind et al., Synergistic Processing in Cell's Multicore Architecture [37] The Cell processor is an industrially used example for a heterogeneous multi-core processor. It is used in the Sony Playstation 3. This article shows that it can be used for more than gaming.

P(19): Aguiar et al., Embedded systems' virtualization: The next challenge? [5] Virtualization is a common technique to consolidate server hardware or operate several operating system in parallel. However, when it comes to heterogeneous architectures in embedded system virtualization might help in providing a unified interface for different kinds of hardware. This paper provides an overview of this topic.

P(20): Stoif et al., Hardware synchronization for embedded multi-core processors [98] Synchronizing upon a shared memory region may be a time consuming task in the field of embedded system. Therefore this paper presents an approach to implement synchronization techniques in hardware. The approach is evaluated using an FPGA housing two PowerPC processors.

P(21): Holt et al., Software Standards for the Multicore Era [44] Libraries like OpenMP and MPI are well suited for personal computers and server machines. However, embedded system do not always provide a lot of computing power or main memory. Therefore, this paper presents a lightweight multi-core communication API especially designed for embedded system. This library makes no assumptions on the used memory architecture.

The description of numerous papers in the field of embedded systems concludes this Subsection. The following Subsection will present a set of possible topics students may choose to compose their seminar papers on.

A selection of possible topics for student contributions

This Section provides an overview of five possible assignments given to students participating in the university oriented multi-core programming course.

(1) Investigate TLA+ a language to specify concurrent systems

Students who are especially interested in the formal specification and proofing of parallel systems may choose this topic. Here an investigation of Lamports TLA+ language and toolset including a demonstration is the desired outcome of the seminar paper, therefore having a small tutorial on the most crucial features of the toolbox. The software can be obtained from `http://www.tlaplus.net/`. Moreover, there exists a book [58] and a set of publications introducing this topic. The desired structure of the seminar paper should consist of a section dedicated to background information, a section on the toolbox itself and finally a demonstration and showcase of a real application, that is also implemented in real hardware/software.

(2) Parallel programming languages - an overview and capability demonstration

Students choosing this assignment should investigate and compare parallel programming languages on behalf of the following criteria:

- Purpose of the given language (Academic, Industrial)
- Features and capabilities of the language
- Programming model of the language (How is parallelism enforced)
- Tool support and availability of the language
- Working program demonstration when possible

This seminar paper has a theoretical focus, however there is also room to play with the investigated languages. A team of 2 should at least investigate 4-6 different languages. The investigation shall begin with a descriptive part, where each of the investigated languages is presented. Subsequently, the languages shall be compared against each other on behalf of the named criteria. The last Section is used for example code and experiment results.

(3) Parallel programming in Java

Java is one of the most industrially relevant programming languages [103]. Thus it would be nice to know how Java can be used to develop parallel software. This seminar paper shall investigate and demonstrate the parallel programming capabilities of Java. Moreover, students choosing this topic should investigate the language on possible issues when developing concurrent software. As Java is a commonly known programming language, a lengthy Java introduction is not required. Rather, students are obliged to focus in the mechanisms Java provides to enforce its parallel capabilities.

(4) GZIP parallelization

The widely used compression/decompression program *GZIP* bases upon a sequential algorithm developed by Ziv and Lempel [109]. An improvement of their initial work has been done in [110] by the same authors. It is the purpose of this seminar topic to change this the program *GZIP* in a way that it utilizes the computing power of more than just one processor core on a modern computer. The seminar paper shall consist of the design decisions, their impact on the software, as well as a section on pitfalls and problems. The source code package will be distributed by the lecturer.

(5) Finger print verification parallelization

Utilizing finger prints in jurisdiction and access control is a common task [76]. The aim of this seminar topic is to change a given open-source finger-print verification software in such a way that it utilizes the available computing power of a modern multi-core processor. The source code package will be distributed by the lecturer. Moreover a large set of over 50 sample finger prints will be distributed with the software package in order to test the thoroughness of the solution. The seminar paper shall consist of the design decisions, their impact on the software, as well as a section on pitfalls and problems.

Course Description Summary

This Section provided a detailed description of a multi-core programming course in a university context. The course is designed as a seminar, simulating a conference including a seminar paper presentation at the end. Thus each seminar paper has to undergo a peer review process. This peer reviewing is done by students. Therefore, the overall grade consists of the three components *review quality*, *seminar paper quality* and *seminar paper presentation quality*. The topics for the seminar papers can be chosen from a pool. Each paper is to be handled by a group of students of at least two. Because the course has a seminar like structure, only three lectures with compulsory attendance are held, two of them at the beginning. The third one is used to let the students present their work. This course has been developed on basis of the best practices identified in Section 3.6. The subsequent Section 4.2 will give a detailed introduction on a multi-core programming course in the context of a university of applied science.

4.2 Multi-Core Programming in a University of Applied Sciences Context

In contrast to universities, were most compulsory courses last for a whole term and are held in full-time curricula, universities of applied science tend to offer a more heterogeneous curriculum portfolio. In general there exist full time and extra-occupational curricula in parallel. Considering the extra-occupational curricula they are often organized in such a way, that only two or at most three days of the week are covered with lectures [25–28]. However, this leads to the issue that not all courses can be held throughout a whole term, thus introducing compact lectures only lasting half a term. Therefore, the course described in the following is structured to fit in the most strict curriculum framework, thus being an extra-occupational course lasting only a half term.

Nevertheless, it bases on course **(A)** in Section 3.6. The course is designed to fulfill the following organizational constraints:

- The course must be held within a half term. This includes a written exam at the end.
- Since, the curriculum is extra-occupational all lectures must be held in the evening and have to consist of two units, 45 minutes each.
- The course consists of 8 lectures which are held weekly.

However, these constraints will not restrict the applicability of the intended learning outcome as described in Section 3.6. The general structure of the course may look as follows:

1. The first two lessons will be used to provide students with a theoretic introduction on the topic. This covers an overview on the history of parallel computing and a comprehensive introduction of the theoretical fundamentals of this field.
2. The ascending two lessons are used to introduce students to the general usage of parallel programming design templates and libraries exploiting the parallel capabilities of modern multi-core processors. These two lessons conclude the theoretical part of the lecture.
3. Afterwards, the participants will get together in groups of two to four, where each group will receive a distinct programming task that has to be solved until the next three lessons. The chosen tasks have to be solved at home, since the ascending two lectures will be used for discussion, questions, and intermediate result presentations. However, since it might be the case that students tend work on their tasks just before the deadline, intermediate milestones which have to be presented in class will be issued to the participants. These milestones consist of a small presentation with discussion in each of the two mentioned lectures.
4. The seventh lecture will be used for a final presentation, where all students are obliged to present their home assignment solution. In the eighth lecture the final written exam will take place.

Moreover, since there is only a half term available to teach participating students the basic principles of multi-core programming, the lessons two to five start with a small test checking whether the articles to be read handed out in the prior class have been read and understood. The course' key facts are presented in Table 4.6 concluding this overview. Furthermore, Figure 4.2 shows the course's timeline from a student's point of view.

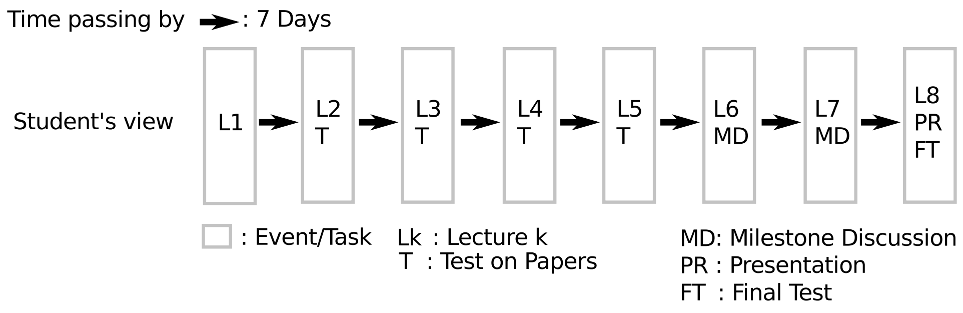


Figure 4.2: Multi-Core prgramming in a university of applied sciences context - detailed course timeline

Table 4.5: University of Applied Science Course Key Facts

Structure	# of Lessons	Attendance	Length of Lesson	Assessment
lecture & exercise	8	compulsory	90 minutes net	small tests on read articles, assignment, written exam

The course's learning outcome is based on the learning outcome description of Course **(A)** in Section 3.6 and is precised in the following way:

- Students are able to understand and describe the theoretical concept of concurrent systems. This includes phenomenons like false-sharing and implications on cache sizes.

- Moreover, they are able to name and describe the most common parallel hardware architectures, as well as being familiar with a set of software libraries exploiting their capabilities best, including OpenMP and Pthread.
- Additionally participants are able to port a small sized sequential program to a parallel one where applicable.

Applied methods of assessment

This Subsection presents the assessment methods used for this course. As it has been mentioned in Section 3.6 Course **(A)**, the main assessment methods are recommended to be hands on oriented, thus a oral examination on the details of the designed and implemented program may be used. However, since this course is only held in a half term, thus leading to less complex programming examples, additional assessment methods are recommended. At first a final examination will be introduced at the end of the course. This written test will cover the following topics:

- Basic understanding of the theoretic fundamentals of parallel computing.
- Application of the theoretic fundamentals on small examples.
- Explanation of several features, strengths and weaknesses of the covered parallel programming libraries.
- Understanding of the key concept covers by the articles read for class.

Additionally, in order to motivate students consecutively learn the topics discussed in class two additional assessment methods are applied.

Concurrent Tests Since the lecture is short on time when lasting only half a term, some topics have to be outsourced to the attending students. Therefore, participants must write a small test on the papers they were told to read until the next time. Each test will not last longer than $10 - 15$ minutes and will cover the key statements of the read articles.

Graded Milestone Discussion Since the second part of the course will cover a practical assignment, these graded milestone discussions will be used to check whether the students have worked on their assignments or not. Thereby the assignment will be divided into several sub-tasks, where each completed sub-task will be covered by a milestone. Each milestone is presented to a lecturer and is discussed in class. A milestone-discussion will be graded according to the thoroughness of the student's defense of his point of view.

A detailed course description

This Subsection provides a detailed course description including a chronological overview and a presentation and justification of the research papers to be read. Furthermore, a set of possible assignments is described. The detailed structure of the course can be seen in Table 4.6. The table consists of the following columns with their respective meanings:

Lecture Reflects the number of the lecture within this term.

Paper Reference to a set of papers that students have to read until this lecture. The reference points to the articles listed in Section 4.1.

Content Topics covered in class.

Table 4.6: University of Applied Science Detailed Course Description

Lecture	Paper	Content
1	-	Ressourcing & Introduction and motivating examples on the importance of parallel computing; Organizational introduction; Short presentation of the history of concurrent computing; Introduction into parallel hardware and software concepts.
2	P(1), P(4), P(17)	Test 1; Theoretic background introduction based on ex-cathedra. Covered topics are: Fundamental concepts; Introduction into models and languages of parallel computation; Definition of the parallel computing design challenge.
3	P(5), P(6), P(7)	Test 2; Discussion of memory models and cache issues when dealing with parallel software.
4	P(19), P(20), P(21)	Test 3; Highlighting of embedded and future concepts in the field of multi-core systems. Set up of groups; Distribution of assignments.
5	P(12), P(13)	Test 4; Discussion of Milestone 1; Moreover, the technical report *Getting Started POSIX Threads* by Schuster, 2011, providing an introduction to PThreads should be read.
6	-	Discussion of Milestone 2
7	-	Each group presents their results to their colleagues and the lecturer.
8	-	Final written exam (90 minutes).

Since the multi-core programming course in the university of applied science context features three independently graded sub tasks, a student's final grade is calculated as it is shown in Table 4.7.

Table 4.7: Weighting of grade components

Component	Weighting in %
Tests on articles	30
Programming assignment	40
Final test	30

Considering parallel programming assignments, the following presented tasks **(1)** and **(2)** in the ascending list are reused from Section 4.1 on page 77 in a modified form:

(1) Finger print verification As mentioned in Section 4.1 on page 77, assignment 5 finger print verification is a common task within various fields. Therefore, it is considered to be an interesting assignment to work on by the author. Students choosing this task have analyze the source code of the program, detect the most time consuming computations and parallelize the source code of the program using OpenMP. This assignment is considered to be suitable for two to four students.

(2) Data compression Since data compression is an omnipresent task in the field of computer science much effort has been made to design an implement efficient compression algorithms. In this assignment two to four students are required to analyze a serial implementation of the zip compression/decompression algorithm, find its performance bottle necks and parallelize the application using OpenMP.

(3) Image processing: Sobel filter [47] This assignment can be taken by up to two students. I consists of an analysis and parallelization task of the Sobel edge detection algorithm used in image processing. Student picking this assignment must use PThread for their implementation. A serial implementation is provided to the participants.

(4) Image processing: Erode filter [47] The erode filter is used in the field of image processing to widen and enhance the brighter areas of an image. This task can be chosen by up to two students which have to use the PThread library to implement a parallel version of this algorithm. A serial implementation is provided to the participants.

(5) Image processing: Dilate filter [47] Dilate describes an mathematical operation in the file of image processing where wider and darker areas of an image are expanded. This assignment is suitable for up to two students which have to use the PThread library in order to transform a given serial implementation of the algorithm into a parallel one.

(6) Image processing: Laplace filter [47] The application of the Laplace filter is used for edge detection in images. This task can be solved by one to two students using the PThread library to implement a parallel version of this algorithm on base of a supplied serial implementation.

Course Description Summary

Section 4.2 presented an approach for a parallel programming course in a university of applied sciences context. The course is designed to fit in a strong constrained curriculum considering the amount of available time. In the case mentioned this would mean that the course's lectures have to take place within a half term, where each lecture has to last 90 minutes. These constraints may apply in an extra-occupational curriculum. Therefore, in order to be able to fulfill a university of applied science aims in providing a practical oriented education the course is provided with a concurrent assessment technique consisting of three parts:

1. Since the whole course has to be held in a half term, a set of research articles covering the topics of the subsequent lecture have to be read by the course participants. In order to verify that the main arguments in the articles have been understood, a small test of about 10 to 15 minutes has to be solve by every student at the beginning of a lecture.

2. To provide students a practical experience in multi-core programming a software engineering assignment has to be solved by each student. Since these assignments differ in size and complexity, the participants have to solve these tasks in groups of two to four, depending on their appointed task size.

3. Finally a written exam consisting of theoretical and practical tasks has to be solved by the participants.

The programming examples solved by the students are complete, or parts of, real-world applications which intend to show students the necessity and importance of the taught topics, as well as to increase their motivation in solving the assigned tasks thoroughly.

4.3 Summary

Chapter 4 discussed two possible approaches when teaching multi-core programming. Generally spoken, both course concepts try to impose the theoretical background on multi-core programming to participating students in combination with practical software engineering tasks. Moreover, both course concepts base upon the insights made in Chapter 3, where numerous parallel programming courses have been analyzed and best practices for the development of the courses described in Section 4.1 and Section 4.2 have been derived. A compact summary of both course concepts results in the following:

University course This course is designed to simulate a conference to participating students. It consists of three lectures with compulsory attendance. The first two lectures are held at the beginning of the course providing the necessary theoretical background, as well as an introduction on the possible assignments to the students. These assignments provide a framework to the students, they do not animate them to focus on a specific detail. The last lecture is used to let the participants present their findings. Since the course is designed like a conference students have to compose a seminar paper. This paper is then peer reviewed by the colleagues. Finally, the participants are obliged to create a camera-ready version on base of their colleague's reviews. The final grade is calculated on basis of the seminar paper, the quality of the peer review, and the final presentation. The main learning outcome of this course is that the participants have a general knowledge about multi-core computing systems and that they have deepened their insights on a predefined part of this field.

University of applied sciences course Since universities of applied sciences offer a very practical education to their students this course is a lot more guided than the university context course. The course consists of an introduction block of about four lectures at the beginning. Here the basic concepts in the field of multi-core programming are explained. Nevertheless, since the course is forced to be held in half a term, which is a realistic constraint in a university of applied sciences due to extra-occupational curricula, it is necessary that participants read and understand research articles on their own at home. Moreover, the participants need to solve homeworks in groups of two to four, depending on the complexity of their chosen task. The home assignments have to be presented in two milestone discussions, where each group has to present their intermediate solutions. Finally, the last lecture will be used for a written exam covering all topics discussed in class and read in the research articles. The final grade is composed on the small tests on the read papers, the milestone discussions, and the final examination. The main learning outcome of this course is to provide students with a solid understanding of the principles in multi-core programming and the applicability of the parallel programming libraries OpenMP and Pthread.

CHAPTER 5
Conclusion

Due to the wide availability of multi-core computer systems in all kinds of applications ranging from personal computer to mobile-phones, it is vital that ongoing engineers receive an education on how to cope with the challenges arising when mastering such systems. As a consequence this thesis presents two course concepts on teaching multi-core programming:

University context This course is designed to simulate a conference to participating students. Its intention is to offer students a greater degree of freedom in order to deepen their knowledge on a specific field in the domain of multi-core programming. Therefore it is necessary that participants are not complete novices in the field of software engineering. The course bases on the recommendations and best practices derived in Chapter 3. It consists of three lectures with compulsory attendance. The first two lectures are held at the beginning of the course providing the necessary theoretical background, as well as an introduction on the possible assignments to the students. The assignments are designed to provide a framework to students where they can explore the actual direction they want to be going. Thus realizing a greater amount of freedom and therefore possibly fostering the autonomy of the participant's work. The last lecture is used to let the participants present their findings. Since the course is designed like a conference students have to compose a seminar paper. This paper is then peer reviewed by the colleagues. Finally, the participants are obliged to create a camera-ready version on base of their colleague's reviews. The final grade is calculated based on the seminar paper, the quality of the peer review, and the final presentation. The main learning outcome of this course is that the participants have a general knowledge about multi-core computing systems and that they have deepened their insights on a predefined part of this field.

University of applied science context Since universities of applied sciences offer a very practical education to their students this course is a lot more guided than the university context course. As a consequence the learning outcome is defined more clearly, as well as the home assignments the students have to work on. The course consists of an introduction block of about four lectures at the beginning. Here the basic concepts in the field of multi-core programming are explained. Nevertheless, since the course is forced to be held in half a term, which is a realistic constraint in a university of applied sciences due to extra-occupational curricula, it is necessary that participants read and understand research articles on their own at home. Moreover, the participants need to solve home assignments in groups of two to four, depending on the complexity of the chosen task. The solved assignments have to be presented in two milestone discussions, where each group has to present their intermediate solutions. Finally, the last lecture will be used for a written exam covering all topics discussed in class and read in the research articles. The final grade is composed on the small tests on the read papers, the milestone discussions, and the final examination. The main learning outcome of this course is to provide students with a solid understanding of the principles in multi-core programming and the applicability of the parallel programming libraries OpenMP and Pthread.

Although none of the described courses has been taught in the described version, an indirect relative version of the university of applied science context course has been held in the summer term of 2012 at the Vienna university of applied sciences. Compared to the version described in this thesis the held course differed in the following way (considering the organizational constraints there were differences between both courses):

- The students were not obliged to read and understand research articles in the field of multi-core programming in advance to the lectures $L2$ to $L5$ as it can be seen in Figure 4.2.

- The given home assignments were of academic nature mostly, thus requiring the student to parallelize prime number calculation algorithms for example.
- There was only one written test covering the theoretical and practical parts of the lecture at the end of the half term.

At the end of the lecture the student's prosa feedback on the course was evaluated. The total number of students participating on the lecture was 22. Although more than two third (15) of the participants noted that the lecture's content was interesting in general, almost all students (19) commented that the home assignments were not very aspiring. Moreover, about half of the participants (9) let us know that although they understood the lecture's content they could not easily tell what they can do with the gained insights in their future live. This feedback in combination with the definitive need on a thorough multi-core programming education led to the course designs presented in this thesis. These course designs will be applied in the summer term 2013 at the Vienna University of Applied Sciences and the Vienna University of Technology.

Bibliography

[1] Advanced RISC Machines Ltd. 2012. Arm architecure reference manual. http://www.arm.com.

[2] Sarita V. Adve and Hans-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, August 2010.

[3] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66 –76, dec 1996.

[4] AEI-NOOSR. Country Education Profiles Australia. www.aei.gov.au/Services-And-Resources/Services/Country-Education-Profiles/About-CEP/Documents/Australia.pdf, 2011.

[5] A. Aguiar and F. Hessel. Embedded systems' virtualization: The next challenge? In *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, pages 1 –7, june 2010.

[6] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[7] Braun E. & Hannover B. Zum zusammenhang zwischen lehr-orientierung und lehr-gestaltung von hochschullehrenden und subjektivem kompetenzzuwachs bei studierenden. *Perspektiven der Didaktik (Zeit-chrift für Erziehungswissenschaft*, 9:277–291.

[8] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. The illiac ipv computer. *Computers, IEEE Transactions on*, C-17(8):746 – 757, aug. 1968.

[9] Mordechai Ben-Ari. A suite of tools for teaching concurrency. *SIGCSE Bull.*, 36:251–251, June 2004.

[10] Mordechai Ben-Ari and Yifat Ben-David Kolikant. Thinking parallel: the process of learning concurrency. *SIGCSE Bull.*, 31:13–16, June 1999.

[11] Olaf René Birkeland. *Searching large data volumes with MISD processing - PHD Thesis*. September 2008.

[12] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[13] BMBF - Bundesministerium für Bildung und Forschung. Rahmenprogramm zur Förderung der empirischen Bildungsforschung. `http://www.bmbf.de/pubRD/foerderung_der_empirischen_bildungsforschung.pdf`, 2007.

[14] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[15] Manfred Broy. Towards a formal foundation of the specification and description language sdl. *Formal Aspects of Computing*, 3:21–57, 1991.

[16] Markus Brunner. Didaktische Entwürfe für den Kompetenzbereich Industrielle Informationstechnik. March 2012. Master Thesis.

[17] Bill Bynum and Tracy Camp. After you, alfonse: a mutual exclusion toolkit. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, SIGCSE '96, pages 170–174, New York, NY, USA, 1996. ACM.

[18] Marina C. Chen. A parallel language and its compilation to multiprocessor machines or vlsi. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 131–139, New York, NY, USA, 1986. ACM.

[19] John F. Croix and Sunil P. Khatri. Introduction to gpu programming for eda. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 276–280, New York, NY, USA, 2009. ACM.

[20] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[21] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), September 2005.

[22] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9 –21, feb. 1988.

[23] European Comission - Education and Training. European Qualification Framework for Lifelong Learning (EQF). `http://ec.europa.eu/education/lifelong-learning-policy/eqf_en.htm`, 2008.

[24] Michael B. Feldman and Bruce D. Bachus. Concurrent programming can be introduced into the lower-level undergraduate curriculum. *SIGCSE Bull.*, 29:77–79, June 1997.

[25] FH Johanneum. Berufsbegleitendes studienangebot. `http://www.fh-joanneum.at/aw/home/studieninfo/Bachelor/Finde_dein_Studium/~cial/sti_bachelor/?lan=de`, last viewed: 3.9.2012.

[26] FH Kaernten. Berufsbegleitendes studienangebot. `http://www.fh-kaernten.at/studienangebot.html`, last viewed: 3.9.2012.

[27] FH Oberoesterreich. Berufsbegleitendes studienangebot. `http://www.fh-ooe.at/campus-hagenberg/studienangebot/`, last viewed: 3.9.2012.

[28] FH Technikum-Wien. Berufsbegleitendes studienangebot. `http://www.technikum-wien.at/studium/berufsbegleitend/`, last viewed: 3.9.2012.

[29] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972.

[30] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, dec. 1966.

[31] Freescale Inc. 2012. QORIQ Advanced Multi-Processor System. `http://www.freescale.com/webapp/sps/site/overview.jsp?code=QORIQ_AMP`.

[32] Freescale Inc. 212. PSC9131 Single Core Single DSP System. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=PSC9131`.

[33] Ronald Goodman and Scott Black. Design Challenges for Realization of the Advantages of Embedded Multi-Core Processors. *Autotestcon*, (September):8–11, 2008.

[34] Ganesh Gopalakrishnan, Yu Yang, Sarvani Vakkalanka, Anh Vo, Sriram Aananthakrishnan, Grzegorz Szubzda, Geof Sawaya, Jason Williams, Subodh Sharma, Michael DeLisi, and Simone Atzeni. Some resources for teaching concurrency. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 2:1–2:6, New York, NY, USA, 2009. ACM.

[35] Richard Graham, Timothy Woodall, and Jeffrey Squyres. Open mpi: A flexible high performance mpi. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 228–239. Springer Berlin / Heidelberg, 2006.

[36] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.3 ed.* Object Management Group, 1999.

[37] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *Micro, IEEE*, 26(2):10 –24, march-april 2006.

[38] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[39] Andreas Hansson, Benny Akesson, and Jef van Meerbergen. Multi-processor programming in the embedded system curriculum. *SIGBED Rev.*, 6:9:1–9:9, January 2009.

[40] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach - 4th Edition*. Morgan Kaufmann, 2006.

[41] Thomas Henzinger and Joseph Sifakis. The embedded systems design challenge. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2006.

[42] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, july 2008.

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[44] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister. Software standards for the multicore era. *Micro, IEEE*, 29(3):40 –51, may-june 2009.

[45] Gosling D. & Moon J. *How to Use Learning Outcomes & Assessment Criteria*. London SEEC, 2001.

[46] Moon J. *How to Use Level Descriptors*. London SEEC, 2002.

[47] Bernd Jaehne. *Digitale Bildverarbeitung. 6. überarbeitete und erweiterte Auflage*. Springer-Verlag, 2005.

[48] Maheshkumar P Jagtap. Era of Multi-Core Processors. *Science*, (March):87–94, 2009.

[49] David A Joiner, Paul Gray, Thomas Murphy, and Charles Peck. Teaching Parallel Computing to Science Faculty : Best Practices and Common Pitfalls. *PPoP06*, pages 239–246, 2006.

[50] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[51] F. Keceli, A. Tzannes, G.C. Caragea, R. Barua, and U. Vishkin. Toolchain for programming, simulating and studying the xmt many-core architecture. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1282 –1291, may 2011.

[52] Christoph Kessler. Teaching parallel programming early. In *Workshop on Developing Computer Science Education - How Can It Be Done?*, 2006.

[53] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32 – 38, nov. 2005.

[54] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690 –691, sept. 1979.

[55] Leslie Lamport. The mutual exclusion problem: part i -a theory of interprocess communication. *J. ACM*, 33(2):313–326, April 1986.

[56] Leslie Lamport. The mutual exclusion problem: partii -statement and solutions. *J. ACM*, 33(2):327–348, April 1986.

[57] Leslie Lamport. Verification and specification of concurrent programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374. Springer Berlin / Heidelberg, 1994.

[58] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[59] Leslie Lamport. Teaching Concurrency. *ACM SIGACT News*, 40(1):58–62, 2009.

[60] E.A. Lee. The problem with threads. *Computer*, 39(5):33 – 42, may 2006.

[61] Linux Kernel 2.6. Linux 2.6 arm machine support. `http://www.kernel.org`.

[62] D.B. Loveman. High performance fortran. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(1):25 –42, feb. 1993.

[63] Lauri Malmi and Ari Korhonen. Active learning and examination methods in a data structures and algorithms course. In *Reflections on the Teaching of Programming*, pages 210–227. 2008.

[64] G. Martin. Overview of the MPSoC design challenge. *2006 43rd ACM/IEEE Design Automation Conference*, pages 274–279, 2006.

[65] Chris McDonald. Teaching concurrency with joyce and linda. *SIGCSE Bull.*, 24:46–52, March 1992.

[66] Paul E. McKenney. Memory ordering in modern microprocessors, part i. *Linux Journal*, 136, June 2005.

[67] A. Meixner and D.J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *Dependable and Secure Computing, IEEE Transactions on*, 6(1):18 –31, jan.-march 2009.

[68] K. Metz-Göckel S. S. Auferkorte-Michaelis N. & Zimmermann. *Schneeflocken oder eigener Forschungstyp*. Bielefeld Universitätsverlag Webler, 2005.

[69] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[70] Auferkorte-Michaelis N. *Hochschule im Blick: Innerinstitutionelle Forschung zu Lehre und Studium an einer Universität.* Münster Lit-Verlag., 2005.

[71] J.R. Newport. An introduction to occam and the development of parallel software. *Software Engineering Journal*, 1(4):165 –169, july 1986.

[72] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008.

[73] Kurt Nørmark, Lone Leth Thomsen, and Kristian Torp. Mini project programming exams. In *Reflections on the Teaching of Programming*, pages 228–242. 2008.

[74] Alastair Nottingham and Barry Irwin. Gpu packet classification using opencl: a consideration of viable classification methods. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, pages 160–169, New York, NY, USA, 2009. ACM.

[75] Nvidia Inc. 2012. Nvidia APX. `http://www.nvidia.com/object/product_tegra_apx_us.html`.

[76] Lawrence O'Gorman. Fingerprint verification.

[77] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30:2–11, September 1996.

[78] Ozcan Ozturk. Multicore Education Through Simulation. *IEEE Transactions on Education*, 54(2):203–209, May 2011.

[79] Shi Qingsong, Chen Tianzhou, Hu Wei, Jolly Wang, and N. Bao. Online programming experience platform for multicore curriculum. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 5, pages 785 –788, dec. 2008.

[80] Qualcomm Inc. 2012. Qualcom Snapdragon 4. `http://www.qualcomm.com/media/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age`.

[81] Brian Rague. Teaching parallel thinking to the next generation of programmers. *Journal of Education, Informatics and Cybernetics*, 1:43–48, 2009.

[82] B. Ramakrishna Rau and Joseph A Fisher. Instruction-level parallel processing: History, overview, and perspective. In B. R. Rau and J. A. Fisher, editors, *Instruction-Level Parallelism*, volume 235 of *The Kluwer International Series in Engineering and Computer Science*, pages 9–50. Springer US, 1993.

[83] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. *SIGPLAN Not.*, 35:245–256, November 2000.

[84] Steven Robbins. Using remote logging for teaching concurrency. *SIGCSE Bull.*, 35:177–181, January 2003.

[85] Bloom Benjamin S. *Taxonomy of Educational Objectives Book 1: Cognitive Domain.* Addison Wesley Publishing Company, 2nd edition edition, October 1956.

[86] Dany S. *Start in die Lehre. Qualifizierung von Lehrenden für den Hochschulalltag.* Münster LIT-Verlag, 2007.

[87] Metz-Göckel S. *Hochschulforschung als Ko-Produktion von Erkenntnissen. Bemerkungen zum produktiven Verhältnis von Wissenschaft und Verwaltung.* Münster Lit-Verlag., 2008.

[88] Samsung Inc. 2012. Samsung Exynos. `http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=844&partnum=Exynos%204210`.

[89] M. Sato. Openmp: parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In *System Synthesis, 2002. 15th International Symposium on*, pages 109 –111, oct. 2002.

[90] Benaoumeur Senouci, Abdellah.M Kouadri.M, Fr Rousseau, and Fr Petrot. Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers. *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 41–47, June 2008.

[91] Jirarat Sitthiworachart and Mike Joy. Effective peer assessment for learning computer programming. *SIGCSE Bull.*, 36(3):122–126, June 2004.

[92] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[93] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609 –1624, dec 1995.

[94] L. Spracklen and S.G. Abraham. Chip multithreading: Opportunities and challenges. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 248 – 252, feb. 2005.

[95] Popovici Katalin Rousseau Frédéric Jerraya Ahmed A. Wolf Marilyn Springer. *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. Springer, 2010.

[96] ST Ericsson inc. 2012. ST Ericsson NovaThor. `http://www.stericsson.com/products/u9500-novathor.jsp`.

[97] William Stallings. *Operating Systems: Internals and Design Principles, 5/E*. Prentice Hall, July 2004.

[98] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase. Hardware synchronization for embedded multi-core processors. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pages 2557 –2560, may 2011.

[99] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[100] Texas Instruments Inc. 2012. TI OMAP 1710. `http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=11991&contentId=4670`.

[101] Texas Instruments inc. 2012. TI OMAP 5. `http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12863&contentId=103102`.

[102] Texas Instruments Inc. 2012. OMAP 4 Mobile appplications platform. `http://www.ti.com`, 2011.

[103] TIOBE Programming Community Index. Tiobe programming community index for july 2012. `http://www.tiobe.com/content/paperinfo/tpci/index.html`.

[104] Peter Tröger. The multi-core era - trends and challenges. *CoRR*, abs/0810.5439, 2008.

[105] Umea University, Sweden. Student conference paper review form. `http://www8.cs.umu.se/kurser/TDBD18/ReviewForm06.pdf`.

[106] UNESCO. International Standard Classification of Education I S C E D. `http://www.unesco.org/education/information/nfsunesco/doc/isced_1997.htm`, Nov. 1997.

[107] David W. Wall. Limits of instruction-level parallelism. *SIGPLAN Not.*, 26:176–188, April 1991.

[108] Greg Wolffe and Christian Trefftz. Teaching parallel computing: new possibilities. *J. Comput. Small Coll.*, 25:21–28, October 2009.

[109] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337 – 343, may 1977.

[110] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530 – 536, sep 1978.