We have prepared a rich and advanced curriculum for extended master program in concurrent and distributed programming. The core of the program can be taken in four semesters; the purpose of the program is however to train experts that will be able to cope with challenging problems where efficiency, performance and safety is important, so we expect that a large part of students will have a prior knowledge of some of the courses (which we offer to our bachelor students) or/and will continue as Ph.D. students. While the program was prepared as an extension of the existing curriculum, it was enriched by taking into account the advice of our alumni working as architects and programmers in software companies. We have also tried to incorporate the best ideas and most recent trends in the curricula offered by the best CS departments. The program will be officially launched in September 2011. Some of the classes will be given by experts from the industry, who have already coped with problems of efficiency of multi-core systems.

*The curriculum was designed by Krystian Baclawski, Marcin Bieśkowski, Leszek Pacholski, and Pawel Rychlikowski. Some details have been consulted with Dariusz Biernacki, Tomasz Jurdiński, Emanuel Kieroński, Andrzej Łukaszewski, Marek Piotrów, Marcin Skórzewski, Piotr Wieczorek, and Tomasz Wierzbicki.
Contents

1 Introduction  3

2 Overview  4

3 Curriculum: Obligatory Courses  6
   3.1 Concurrent Programming  6
   3.2 Distributed Systems  8
   3.3 Concurrent Data Structures  10

4 Curriculum: Additional Courses  12
   4.1 Distributed Algorithms  12
   4.2 Stream Processing  14
   4.3 Architectures of Contemporary Processors  15
   4.4 Multi-core Programming Tools for C++  18
   4.5 Distributed Programming in Erlang  20
   4.6 Seminar: Cache-Aware Algorithms and Data Structures  22
   4.7 Seminar: Algorithms for Massive Data Sets  23
1 Introduction

In the concurrent and distributed data processing, many computations are carried out simultaneously, using different hardware resources such as processors and cores. While creating an efficient sequential application can be usually achieved by employing appropriate data structures, harnessing the power of parallelism is a skill that has been mastered by few. It requires splitting a task into many dependent pieces that are executed in parallel. The interaction between these pieces easily raises synchronization issues, i.e., situations where many computations wait for other ones to finish. Excessive waiting for other processes or for an access to shared resources (e.g., memory) is the main obstacle to creation of efficient and scalable parallel applications that use the potential of multiple computational units. Furthermore, the checking the correctness of a parallel program is far from being trivial, because modern computer systems are asynchronous to a large extent.

One may ask the question whether — given the obstacles — such a shift to parallelism is a sensible move at all. However, it appears that such a trend is inevitable. First, there are problems of large-scale data processing that no single modern processor is able to cope with in reasonable time. Second, more importantly, due to physical constraints, increasing CPU frequencies is much harder and expensive than it was in former decades, and the (quite advanced) hardware optimizations, allowing the execution of several processor instructions at once, are already in place. Hence, to pack more processing power, hardware vendors tend to put multiple processing units (so called cores) into a single chip and/or multiple chips on a single mainboard. However, using this power requires software (operating system and applications) that run in parallel. This trends continues as the massively parallel solutions, e.g., graphic cards composed of hundreds of simplified cores, become more and more common.

The multi-core and many-core technologies are already in place, but what the market lacks is the intellectual potential to use this technology. While students at decent universities gain the solid algorithmic and programming background on single-core systems, teaching the development of concurrent applications is still in its infancy. Surely, many graduates know the basics of concurrency, such as thread and process management, but their experience in that matter is usually restricted to single-processor, single-core applications, where many efficiency issues simply do not exists and synchronization errors are less likely to occur. For example, providing an efficient access to a single data structure (e.g. a queue) is a trivial task there, because only one process is executed at a time. On the multi-core systems, this issue, called memory contention minimization, is one of the most vital issues the programmer has to deal with.

This master studies program offers a systematic approach to the concurrency-related issues of the modern computing. As mentioned above, multi-core systems require understanding of new computational models, algorithms, and programming tools. Students’ competences are developed in three related areas.

- **Concurrent programming on multi-core machines.** This part addresses the dominant hardware architecture of modern computers. Algorithmic toolbox essential in creating effective software (such as concurrent, lock-free data structures and stream processing paradigms) is complemented with in-depth experience about the state-of-the-art tools assisting the software creation.

- **Low-level hardware details.** For sequential programming, skills related to algorithms and data structures are usually sufficient. However, for the multi-core and distributed processing, understanding the details of microprocessor architectures (such as the cache hier-
archies or internal algorithms for assuring memory coherence) is needed for any programmer. Developing low-level code, such as operating systems or compilers, requires even more advanced hardware knowledge, such as details of instruction flow in superscalar processors.

- **Distributed systems.** This part describes techniques and paradigms of distributed data processing, used everywhere where single computers are not sufficient. Solid algorithmic background for asynchronous processes that communicate by passing messages is presented. This part also encompasses the tools necessary for creating a medium-scale distributed systems. As in the first part, solid algorithmic background is accompanied by implementation details.

Although the combination of the teaching material is unique, some of the courses have equivalents in the top world computer science institutes, such as the Stanford University, UC Berkeley or Massachusetts Institute of Technology; references to appropriate materials are given in the description of the courses. The presented program covers all concurrency-related topics of the Computer Science Curriculum 2008 composed by the joint task force of the ACM and IEEE Computer Society. Information about the knowledge units covered is listed in the courses' description, wherever applicable.

Some of the courses that are part of this curriculum were already taught at the University of Wroclaw, but there are a couple that will be introduced shortly. Part of the material was already presented on other lectures, but has now been gathered and organized in order to create a coherent teaching program. The contents of particular courses were prepared by different specialists of the given fields, hence there might be some discrepancies in the level of detail used in the descriptions.

## 2 Overview

The presented Concurrent and Distributed Programming (CDP) program is a track in the master program offered at the Institute of Computer Science, University of Wroclaw. Therefore, the characteristic features of our master studies (such as ECTS-based credit system, freedom of constructing individual selection of courses, and the emphasis put on solid theoretical foundations) are also present in this program. This section covers the competence requirements for the program candidates and a simplified description of the rules of study. Next sections contain detailed descriptions of the courses of the CDP track.

### Competence requirements for candidates

A candidate for the master studies should hold a bachelor degree, or its equivalent. They should be able to use mathematical notions and tools from calculus, logic, algebra and discrete mathematics. They should be able to create and analyze algorithms, know different programming paradigms and languages, and be able to use this knowledge in practice. They should understood well the architectures of modern computers, operating systems, computer networks, databases, numerical methods, software engineering methods, and be able to take active part in programming projects.
General rules for obtaining a Master’s Degree

Most of the courses in the graduate studies program are optional. Each student can choose freely any of the available courses and it is her/his responsibility to decide if her/his knowledge is sufficient to take advantage of the course. While making their choices, students are encouraged to carefully read descriptions of the courses and their prerequisites.

To successfully complete the Concurrent and Distributed Programming (CDP) track, a student has to fulfill the following conditions:

(i) finish all three obligatory courses: Concurrent Programming, Distributed Systems and Concurrent Data Structures (see Section 3);

(ii) finish at least four additional courses of the CDP track (see Section 4);

(iii) finish courses whose total sum of ECTS points is at least 120.

The list of the available additional courses (see Section 4) is quite flexible. Each year some new courses may be introduced and some out-of-date ones removed. The purpose of this process is to keep in touch with current achievements of computer science and also with the requirements of employers. The students have an opportunity to express their opinion on this subject by voting for courses presented to them before the academic year starts.

An example of a schedule

An example of a schedule satisfying all requirements is presented below. This schedule contains all the courses “belonging” to the CDP track. They are complemented by other courses taught at the Institute.

<table>
<thead>
<tr>
<th>Semester 1</th>
<th>ECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent Programming</td>
<td>CDP obligatory</td>
</tr>
<tr>
<td>Distributed Systems</td>
<td>CDP obligatory</td>
</tr>
<tr>
<td>Computational Complexity</td>
<td></td>
</tr>
<tr>
<td>Advanced Java Course</td>
<td></td>
</tr>
<tr>
<td>Advanced Numerical Analysis</td>
<td></td>
</tr>
<tr>
<td>Semester 2</td>
<td></td>
</tr>
<tr>
<td>Concurrent Data Structures</td>
<td>CDP obligatory</td>
</tr>
<tr>
<td>Stream Processing</td>
<td>CDP additional</td>
</tr>
<tr>
<td>Distributed Programming in Erlang</td>
<td>CDP additional</td>
</tr>
<tr>
<td>Multi-core Programming Tools for C++</td>
<td>CDP additional</td>
</tr>
<tr>
<td>Large-Scale Databases</td>
<td></td>
</tr>
<tr>
<td>Semester 3</td>
<td></td>
</tr>
<tr>
<td>Distributed Algorithms</td>
<td>CDP additional</td>
</tr>
<tr>
<td>Architectures of Contemporary Processors</td>
<td>CDP additional</td>
</tr>
<tr>
<td>Seminar: Cache-Aware Algorithms and Data Structures</td>
<td>CDP seminar</td>
</tr>
<tr>
<td>Realistic Computer Graphics or Image Processing</td>
<td></td>
</tr>
<tr>
<td>Information Retrieval and Text Mining</td>
<td></td>
</tr>
<tr>
<td>On-line Algorithms or Approximation Algorithms</td>
<td></td>
</tr>
<tr>
<td>Semester 4</td>
<td></td>
</tr>
<tr>
<td>Seminar: Algorithms for Massive Data Sets</td>
<td>CDP seminar</td>
</tr>
<tr>
<td>Compiler Design</td>
<td></td>
</tr>
<tr>
<td>Master Thesis Preparation</td>
<td></td>
</tr>
</tbody>
</table>
3 Curriculum: Obligatory Courses

3.1 Concurrent Programming

The main objective of this course is to teach students developing correct concurrent programs, which utilize the power of current multi-core processors. The course covers shared memory and message passing models, as well as implicit parallelism. While the course concentrates on practical issues, it also gives a theoretical background (such as process calculi or temporal logic) needed for deeper understanding of concurrency in programming. Most of the concepts are illustrated with Java, but miscellaneous concurrent constructions are presented in other programming languages as well.

What you will learn and gain from this course:

- What are the main differences between concurrent and sequential programming.
- How to model concurrent programs and how to describe their properties.
- What kind of tools modern languages provide in order to make the concurrent programmer’s live easier.
- How to change the sequential solution of the problem into parallel one.
- What are the best practices in designing, implementing, debugging and testing concurrent programs.

Prerequisites:

- *Programming*: general knowledge of programming, the ability of writing correct sequential programs, familiarity with imperative and object oriented paradigms, basic knowledge of functional and logic programming.
- *Java language course*.
- *Operating systems*: basic concurrency-related notions: threads, processes, transactions, instruction interleaving, classical concurrency problems (reader-writer, dining philosophers, etc.), POSIX threads and inter-process communication.

Contents:

1. Basics (4h)
   a) Models of parallel computers, concurrency and parallelism, atomicity.
   b) Mutual exclusion and critical sections.
   c) Properties of parallel programs: deadlock freedom, livelocks, starvation problems, fairness, race conditions.
   d) Operating system support: native processes and threads, threads in popular virtual machines (JVM, CIL).
   e) Shared memory and distributed memory.
   f) Factors affecting performance of parallel programs: granularity, coverage, Amdahl’s law, the load balancing problem.
   g) Hardware considerations: GPU, Cell, interconnection networks.
   h) MapReduce paradigm.

2. Mathematical foundations (4h)
   a) Modal logic as a tool of expressing desired program properties (CTL, CTL*).
   b) Process calculi: CSS, CSP, Pi-calculus.

3. Shared memory (5h)
a) Locks, semaphores, mutexes, reentrant mutexes, barriers.
b) Usage of concurrent data structures: concurrent hash map, copy-on-write arrays, 
   blocking and unblocking queues.
c) Critical regions, monitors, conditional variables.
d) Thread pool support.
e) Introduction to Ada language and rendez-vous.
f) Transactional memory and its implementations.
4. Actors and message passing (5h)
   a) Blocking and unblocking messages, synchronous and asynchronous messages.
   b) Message passing in occam.
   c) Tuple spaces, Linda programming language, JavaSpaces.
   d) Actor model: Erlang.
5. Implicit parallelism (4h)
   a) Parallelism in declarative languages. Prolog AND-parallelism and OR-parallelism.
      Parallel constraint solving. Parallel Haskell.
   b) Parallelism in imperative languages (Matlab).
   c) Parallel optimisations in compilers. Dependence analysis and recognising forall 
      loops, communication optimisations, Array privatisations.
6. Design patterns for parallel programming (3h)
   a) Four common steps in creating parallel programs: Decompose, Assign, Orchestrate, 
      Map.
   b) Data, Task and Pipeline decompositions.
   c) How to transform sequential programs into parallel ones.
   d) SPMD pattern, loop parallelism, master/worker, fork/join.
   e) Recursive and geometric decompositions.
   f) Communication patterns: serial reduction, tree based reduction, recursive-doubling 
      reduction.
   g) Common errors in parallel programs.
   h) Programming client-server applications.
7. Other parallel programing languages (2h)
   a) Scala actors: combining shared memory with Erlang actors.
   b) Clojure: Concurrent programming through software transactional memory, an agent 
      system, and a dynamic var system.
   c) StreamIt: composing programs using: filter, pipeline, splitjoin, feedback loop.
   d) Go: channels and alternative channel inputs.
8. Overview of standards, tools and libraries (3h)
   a) OpenMP, OpenCL, CILK, Unified Parallel C.
   b) MapReduce implementations: Hadoop.
   c) Remote procedure call, remote method invocation, XML-RPC, MPI.
   d) Java concurrent utilities.

Labs & Exercises:

• Exercises (10h): expressing program properties in temporal logic, proving simple prop-
  rities of this logic, analyzing the relations between process calculi, analyzing the expres-
  siveness of these calculi, writing and discussing short parallel programs written in high 
  level concurrent languages.
• Computer lab (10h): writing parallel programs in Java.
• Computer lab (10h): writing parallel programs in Erlang, Go, and C/C++.
3.2 Distributed Systems

Building large-scale systems, such as advanced web applications, poses many interesting challenges, especially on the design level. The choice of architecture, algorithms and effective usage of resources are the key factors. Constructing fully-fledged systems that run on multiple clusters and scale to meet the requirements of the user base is a very complex process. Nevertheless, all such systems share many building blocks and software architects do not have to reinvent the wheel and program these concepts from scratch. For example, it makes little sense to invent own fault tolerance techniques, while there exist tried-and-trusted approaches. The presented building blocks are used in real-life web services such as Google, Facebook or Twitter.

What you will learn and gain from this course:

- Basic blocks and tools for creating large-scale distributed systems.
- How to create efficient and redundant systems capable of serving millions of users.
- How to mitigate the effects of the crash; how to minimize the downtime.
- How to effectively use resources (hardware and human).

Prerequisites:

- Operating systems: basic concepts: processes, synchronization, inter-process communication, file systems, security.
- Computer networks: OSI model, TCP/IP, client-server, peer-to-peer, DNS, HTTP.
1. Introduction (2h)
   a) Basic terms: transparency, openness, scalability, fault tolerance, adaptation, security, transactions.
   b) Types of distributed systems, design and implementation pitfalls.
2. Architectures (2h)
   a) Layered, blackboard, pipelined, client-server (stateful, stateless), model-view-controller, publisher-subscriber.
   b) Decentralized (P2P), overlay network, super-peers, interceptors.
   c) Self-managed (feedback, healing, optimizing).
3. Processes (2h)
   a) Processes and threads recap. Execution in distributed environment.
   b) Virtualization (utilization factor), hypervisor.
   c) Code migration (strong vs. weak mobility), resource binding.
4. Communication (2h)
   a) Layers and communication types recap.
   b) Remote procedure calls: marshalling, parameter passing, remote references, programming with RPCs (IDL, stubs), client-to-server binding.
   c) Message passing: queues, buffers, brokers.
   d) Streams: source, sink, jitter, QoS, synchronization.
   e) Multicasting: application level, epidemic algorithms (anti-entropy, gossiping).
5. Naming (2h)
   a) Concepts: names, identifiers, addresses, namespaces, name resolution.
   b) Flat namespace: broadcasting, forwarding pointers, home-base approaches, distributed hash tables, hierarchical location services.
   c) Structured name space: name linking, resolution, DNS, scalability issues.
   d) Attribute-based naming: directory services (LDAP), decentralized solutions.
6. Synchronization (2h)
   a) Clock synchronization: principles, physical clocks, GPS.
   b) Logical clocks: happened-before relation, Lamport’s clock, vector clock.
   c) Mutual exclusion: centralized, decentralized, distributed, token ring.
   d) Node positioning: latency, geometric overlay network.
   e) Election algorithms: Bully algorithm, Ring algorithm, election in wireless environment, superpeer election.
7. Consistency, replication (2h)
   a) Data-centric consistency models: sequential, casual consistency, grouping.
   b) Client-centric consistency models: system model, monotonic reads/writes, read-your-writes, write-follow-reads.
   c) Replica management: placement, content replication and placement, distribution.
8. Fault tolerance (2h)
   a) Basics: fault prevention, tolerance, removal, forecasting, failure models.
   b) Process resilience: flat/hierarchical groups, failure masking, failure detection.
   c) Reliable communication: reliable RPC, multicasting, scalable multicasting (feedback suppression, hierarchical solution), atomic multicast, virtual synchrony.
   d) Distributed commit: two phase commit, three phase commit.
   e) Recovery: consistent recovery state, checkpointing, message logging.
9. Security (2h)
   a) Basics: threats, mechanisms, policies, cryptography, design issues.
   b) Secure channels, secure group communication:
      - authentication: secret keys, public key, Needham-Schroeder protocol,
      - integrity & confidentiality: digital signatures, msg digests, pubkey signatures.
   c) Access control: authorization vs. authentication, access control matrix, protection domains, firewalls, secure mobile code, protecting an agent/a host.
   d) Security management: key establishment (Diffie-Hellman) and distribution (KDC, CA), secure group management, authorization management, delegation.

10. Analysis of real-world distributed systems (12h)
   b) Distributed file-systems: NFS, Plan 9, Coda, Google FS.
   c) Distributed web-based systems: Apache, Squid, Bind, SOAP, WebDAV, TLS.
   d) Distributed coordination-based systems: JMS, Jini, JavaSpaces, Google Chubby.
   e) Monitoring and logging in distributed systems: Ganglia, Google Sawzall.
   f) Distributed databases: Google BigTable, Cassandra, Mnesia.

Labs & Exercises:
   • Exercises (10h): study and comparison of solutions presented at the lecture.
   • Seminar-style exercises (10h): writing a design document for a chosen distributed system, presenting, peer reviewing, improving.
   • Computer lab (10h): implementing a prototype along the lines of the proposed design.

References:
   • *Distributed Systems Concepts and Design*, George Coulouris, Jean Dollimore, Tim Kindberg; Addison Wesley, 2005.

Similar courses:
   • CS244b Distributed Systems, Stanford University
   • 22C:166 Distributed Systems and Algorithms, University of Iowa
   • CS347: Transaction Processing and Distributed Databases, Stanford University
   • Google Code University - Distributed Systems

ACM CR CS2008 knowledge units:
   • Architecture and Organization / Distributed Architectures [elective]
   • Operating Systems / Fault Tolerance [elective]
   • Information Management / Transaction Processing [elective]
   • Information Management / Distributed Databases [elective]

3.3 Concurrent Data Structures

On this course, you learn how to design concurrent data structures in multiprocessor systems with shared memory. This is seemingly an easy task as one may take a well-known sequential
implementation of the data structure in question (e.g., a stack) and then ensure that from threads running in parallel only one can access this data structure simultaneously. But then such data structure becomes a bottleneck and may ruin all the gains from the parallel execution!

How to effectively parallelize these data structures is the central question of this course. We both show how to utilize fine-grained locking mechanisms and how to use hardware support to construct lock-free algorithms. We show how to rigorously reason about the validity and crucial properties (lack of deadlocks, etc.) of constructed solutions. At the same time, we will point out the deficiencies of standard runtime complexity analysis: knowing the details of memory hierarchy organization in multiprocessor systems is crucial in designing scalable low-contention algorithms.

What you will learn and gain from this course:

- How to really utilize the power of modern multiple processors/cores in shared memory environment.
- How to create low-contention algorithms, i.e., how to program keeping in mind that the memory is the main bottleneck.
- How to reason about the interplay between threads and avoid bugs that are difficult to detect.
- How to parallelize seemingly sequential data structures such as stacks or queues.
- Solid basis for the concurrent algorithms related research.

Prerequisites:

- *Algorithms and data structures*: analysis of algorithms’ complexity, basic data structures: trees, priority queues and hash tables.
- *Computer architectures*: familiarity with computer construction (data bus, processor, memory hierarchy), memory organization in multiprocessor systems (MESI protocol, false sharing, etc.).

Contents:

1. Basics (6h)
   a) Implementing basic blocks: critical sections, mutual exclusion algorithms, consistency of shared objects.
   b) Foundations of shared memory: register consistency, atomic registers.
   c) Consensus problem: hardware requirements beyond atomic registers. Universality of consensus, i.e., relation between consensus and wait-free versions of basic data structures and read-modify-write operations (e.g., compare-and-set).
   d) Real-world synchronization issues in modern hardware: scalable spin locks, backoff protocols.
2. Concurrent data structures (18h)
   a) Linked lists: coarse-grained vs. fine-grained locking, deadlocks, lock-free implementations, concurrent queues.
   b) Memory reclamation: ABA-type problems.
   c) Parallelizing high-contention sequential structures by elimination: concurrent stack.
   d) Parallel sorting and counting: bitonic networks, diffracting trees, sample sort.
e) Parallel hashing: lock stripping, lock-free implementation, concurrent cuckoo hashing.
f) Skip lists and concurrent priority queues.

3. Transactional memory (6h)
   a) Motivation by inadequacy of currently applied solutions for creating parallel software: issues with robustness, scalability and compositionality.
   b) API elements: transactions, tentative changes and commits, nested transactions.
   c) Hardware approaches: transactional cache coherence.
   d) Implementation of a simple software transactional memory: transactional threads, synchronization conflicts, contention managers, inconsistent states and zombies.

Labs & Exercises:

- Exercises (10h): developing algorithms, analyzing their complexity, reasoning about correctness of algorithmic solutions, such as no-deadlock property.
- Computer lab (20h): algorithms’ implementation in an imperative programming language (e.g., C++ or Java).

References:


Similar courses:

- Multicore Programming, Tel Aviv University
- CS176: Introduction to Multiprocessor Synchronization, Brown University
- 6.852J / 18.437J Distributed Algorithms, Massachusetts Institute of Technology

4 Curriculum: Additional Courses

4.1 Distributed Algorithms

This course concentrates on distributed computations where parties communicate via network by passing messages. The goal is to familiarize students with basic algorithms that can be executed on multiple processors in a distributed environment, where the central control does not exist. Fundamental problems such as leader election, consensus, mutual exclusion or data aggregation are the core of this course. Finally, this course concentrates also on message passing implementations that induce low loads on the links.

It is worth noting that the growth of the number of processors or cores in a single machine will render standard communication patterns (e.g. by a shared bus) infeasible. Thus, the methods and concepts of this course — while currently applicable mostly in networking systems — will be needed also in single devices.

What you will learn and gain from this course:

- What is the message complexity and why should you aim at minimizing it.
• A toolbox for creating efficient distributed algorithms both in synchronous and asynchronous environment.
• How to organize low-contention message passing in structured networks.
• Solid theory basis for research on distributed algorithms.

Prerequisites:
• *Algorithms and data structures:* analysis of algorithms’ complexity, analysis of randomized algorithms.

Contents:
1. Distributed algorithms in synchronous model (12h)
   a) Synchronous model: message-passing systems, executions, failures, complexity measures.
   b) Leader election in rings: upper and lower bounds.
   c) Leader election in general networks: flooding and optimizations.
   d) Distributed breadth-first search algorithms, building spanning trees, broadcasts and convergecasts.
   e) Maximal independent sets: Luby’s randomized solution.
   f) Distributed consensus: fault tolerance, stopping failures, byzantine failures, approximate agreements, impossibility results.
2. Distributed algorithms in asynchronous model (12h)
   a) Asynchronous model: properties, proof methods, complexity measures.
   b) Breadth-first search, spanning trees and convergecast.
   c) Synchronizers: local and safe.
   d) Termination detection: Dijkstra-Scholten algorithm.
   e) Logical time for asynchronous networks, clock synchronization.
3. Optimizing link loads (6h)
   a) Basic notions: packet switching, congestion, dilation.
   c) Online packet switching: random rank protocol, bounding packet queues.

Labs & Exercises:
• Exercises (15h): developing distributed algorithms, optimizing time and message complexity, proving lower bounds.
• Computer lab (15h): implementing distributed algorithms in Erlang (this is the language of choice as it allows easy implementation of message passing algorithms).

References:
• *Distributed Algorithms,* Nancy A. Lynch, Morgan Kaufmann, 1996.
• *Introduction to Reliable and Secure Distributed Programming,* Christian Cachin, Rachid Guerraoui, Luís Rodrigues; Springer, 2011.
4.2 Stream Processing

Nowadays, many modern processors, even single-core ones, are able to process specific data in parallel. This computation model, called SIMD (single instruction multiple data) allows a single processor to execute the same instruction, in parallel, on a vector of several numbers. The allowed instructions are typically numeric operations on floating point numbers. In some applications, including but not limited to computer graphics, a sensible usage of this feature may result in tenfold speed-up of the program. This data parallelism paradigm was further developed in graphics processing units (GPUs), which can be viewed as a cluster of independent cores, each capable of running instructions on vectors of data.

However, stream processing computation model puts some strict demands on the design of algorithms and their implementation. In particular, the usage of branching instructions has to be limited and hence data dependencies should be reduced whenever possible. Further, the memory utilization has to be as small as possible. (GPUs leave cache memory organization to the programmer!) This course is devoted to harnessing the power of such parallelism to utilize the processor capabilities to the full extent.

What you will learn and gain from this course:

- What kind of data parallelism is supported in modern architectures by Intel, AMD, ATI or NVidia.
- How to effectively program massively parallel GPUs.
- How to utilize data parallelism in computations.
- When to use the power of the CPU and when to shift the load to the GPU.

Prerequisites:

- *Algorithms and Data Structures*: sorting algorithms (quicksort, radix sort, sorting networks).
- *Concurrent Programming*: basics, map-reduce paradigm.
- Programming in C/C++.

Contents:

1. Basics (6h)
   a) CPU vs. GPU architecture: programming models (SSE/AVX, CUDA, OpenCL), threads, memory model.
   b) Simple examples: parallel reductions, matrix addition, matrix multiplication.
   c) Limitations of the streaming model: branching and conditional expressions.
2. Single-core stream algorithms (12h)
   a) Introduction to SSE/AVX instructions: compiler support (gcc, Intel Compiler Suite).
   b) Overview of libraries and tools: Intel Array Building Blocks, automatic vectorization.
   c) Image processing primitives: color conversions, image smoothing, interpolation algorithms, gaussian blur.
   d) Image and video compression: inverse discrete cosine transform.
   e) Codes: computing block ciphers (AES) and CRC checksums.
   f) String processing: fast XML validation.

3. Multiple-core stream algorithms (12h)
   a) GPU memory architecture: variables, memory banks, warps, half-warps, prefetching, coalescence.
   b) Physical computations: electrostatic potential map, parallel n-body simulations.
   c) Sorting on GPU: bitonic sort, radix sort, quicksort, hybrid algorithms.
   d) Map-reduce implementations: Mars framework, sample programs.
   f) Implementing Fast Fourier Transform.

Labs & Exercises:
   - Computer lab (16h): simple programs corresponding to the lecture, using SSE extensions and the CUDA framework.
   - Computer lab (4h): CUDA SDK debugger and profiler, detecting bottlenecks.
   - Computer lab (10h): final project.

References:
   - Intel SSE4 Programming Reference.
   - NVIDIA CUDA Programming Guide.

Similar courses:
   - CS193g: Programming Massively Parallel Processors with CUDA, Stanford University

ACM CR CS2008 knowledge units:
   - Architecture and Organization / Multiprocessing [core]

4.3 Architectures of Contemporary Processors

The goal of this course is to present the complex structures of modern microprocessors. The lowest level on which we can interact with the processor are assembler instruction. In old days, a single instruction would just translate into one or several processor cycles. Nowadays, however, most processors implement a so-called instruction-level parallelism, which allow to execute instructions in parallel. For efficiency of such execution, processors implement many algorithms and policies, in which instructions can be pipelined, executed in different order, or executed speculatively (e.g., the outcome of the compare-and-jump instruction, and thus the
further flow of the program instruction is guessed). Further, the memory hierarchy and the cache internals are also transparent to the programmer.

While these optimizations are always enforced, the understanding of them is crucial for constructing effective software, as in certain, rare cases they may lead to performance degradation, e.g., due to cache trashing or branch misprediction. Moreover, some out-of-order executions while safe on single processors can change the program semantics when run on multiple cores simultaneously.

What you will learn and gain from this course:

- The internals of the modern microprocessors.
- The ability to work as the designer of digital systems.
- Understanding compilers’ low-level optimizations.
- Base for the research on computer architectures.

Prerequisites:

- Computer architectures: familiarity with construction concepts of computers (data bus, throughput, RAM) and processors (instruction set, pipelining, caches, virtual memory, interrupts, stack).

Contents:

1. Processor Design (2h)
   a) Architecture, implementation, realization, analysis.
   b) Instruction Set Architecture, Dynamic/Static Interface, microarchitecture.
   c) Processor performance: CPI, IPC, performance evaluation.
   d) Instruction Level Parallelism: pipelining, superscalar execution, limits of ILP, VLIW.

2. Pipelining (2h)
   a) Fundamentals: motivations, limitations, tradeoff, examples.
   b) Prerequisites: uniform subcomputations, identical computations, independency.
   c) Design: stages (IF, ID, EX, MEM, WB), unifying instruction set, stalls, depth.
   d) Minimizing stalls: pipeline hazards, forwarding, interlocks.
   e) Interrupts: precise vs. imprecise, handling.

3. Superscalar Organization (2h)
   a) Motivation: limitations of scalar pipelines.
   b) Scalar pipeline extensions: parallel, diversified, dynamic pipelines.
   c) Overview: fetching, decoding, dispatching, execution, completion and retiring.

4. Superscalar Techniques
   a) Instruction Flow (2h)
      - Motivation: control dependencies, branch instruction.
      - Branch speculation: target, condition prediction; history-based predictors (BTB, 2-bit branch predictor), misprediction recovery.
      - Adaptive predictor: pattern history table, branch history shift register.
   b) Register Data Flow (4h)
      - Register allocation and recycling, dependency types, data flow limit.
      - Register renaming: source read, destination allocate, register update.
      - Tomasulo’s algorithm: reservation stations, common data bus, register tags.
• Dynamic execution core: reservation stations, reordering buffer, dynamic instruction scheduler.

c) Memory Data Flow (2h)
• Memory access: address generation and translation, load/store execution.
• Load/store out-of-order execution: load bypassing and forwarding.
• Multi-ported data cache, nonblocking cache, prefetching.

5. Advanced Instruction Flow Techniques (4h)
a) Static BP: forward/backward always (non-)taken, branch delay slot, branch hints, Ball/Larus heuristics, profiling.
b) Dynamic BP: Smith’s Algorithm, 2-Level BP, index-sharing BP, interference reduction, loop counting predictor.
c) Hybrid BP: tournament predictor, predictor selection, prediction fusion.
d) Other Techniques: target prediction, branch confidence prediction, trace cache.

6. Advanced Register Data Flow Techniques (2h)
a) Value locality, memoization, instruction reuse.
b) Speculative techniques: weak dependence model, value prediction, execution using predicted values.

7. Executing Multiple Threads (2h)
a) Explicit multithreading: resource sharing, fine-grained, coarse-grained, symmetric.
b) Implicitly multithreaded CPUs.

8. Caches (4h)
a) Types: transparent, software managed, scratch-pad RAM.
b) Virtual addressing and protection: virtually vs. physically tagged or indexed, ASID.
c) Heuristics for replacement and prefetching: on-line and off-line techniques.
d) Consistency and coherence: memory consistency models, MESI protocol, snoopy and directory-based protocols, scalability.
e) Interfacing processor: pipelined and multiported caches.

9. Virtual memory (2h)
a) Address space organization, ownership, identifiers.
b) Hierarchical vs. inverted page tables, modern segmentation, disjunct page tables.
c) Implementation: integrating MMU with CPU’s pipeline, handling interrupts.

10. Intel P6 microarchitecture description (2h)

Labs & Exercises:
• Exercises (20h): study of contemporary solutions, making amendments to techniques explored at lectures and conveying performance evaluation.
• Computer lab (10h): low-level (assembler & C) programming, measuring the effectiveness of different code optimization approaches.

References:
• Memory Systems: Cache, DRAM, Disk, Bruce Jacob, Spencer Ng, David Wang; Morgan Kaufmann, 2007.
4.4 Multi-core Programming Tools for C++

While C and C++ were designed for sequential applications, they are still widely used in modern world, especially where efficiency is the key factor. However, writing concurrent programs using standard threading libraries requires a lot of effort from the programmer as threads have to manually created, terminated and synchronized. Also assigning task to threads has to be optimized by hand. This course presents tools and libraries that extend C/C++ in various ways and automatize these mundane tasks. In general, the programmer divides program into tasks that can be executed in parallel: their dynamic assignment to threads as well as the creation and synchronization of the threads is performed automatically by the library algorithms. This allows writing efficient and portable code, without the hassle of low-level synchronization issues and sometimes enables almost effortless parallelization of a previously sequential code. The above material is complemented by the the usage of debugging and profiling tools for writing parallel applications.

What you will learn and gain from this course:

- How to write effective, parallel and portable programs using C/C++.
- What are the advantages of different parallel-supporting libraries.
- How to debug parallel C/C++ programs and make them more effective after profiling.
- How to adapt your parallel C/C++ programs to a particular hardware architecture.
- How to use Intel Parallel Building Blocks, OpenMP, and OpenCL

Prerequisites:

- Concurrent Programming: basic concepts (shared memory, message passing), parallel programming patterns.
- Programming in C/C++.
- Computer architectures: familiarity with memory hierarchy, caches, virtual memory, TLB, general knowledge on low-level programming.
Contents:

1. Intel Threading Building Blocks (6h)
   a) Library overview.
   b) Basic Loop parallelization: parallel_for, parallel_reduce, parallel_scan.
   c) Parallelizing complex loops: parallel_do, parallel_while.
   d) Pipelining.
   e) Concurrent containers: concurrent_queue, concurrent_vector, concurrent_hash_map.
   f) Mutual exclusion and atomics.
   g) Portable global time stamp.
   h) Scalable memory allocation.
   i) Task scheduler.

2. Intel Array Building Blocks (4h)
   a) Regular and irregular containers.
   b) Collective operators: reductions and scans.
   c) Loops and conditionals.
   d) ArBB virtual machine and execution model.
   e) Closures and capturing.
   f) Debugging and optimizing for performance.

3. Cilk, Cilk++, Intel Cilk++ (4h)
   a) Basic Parallelism in Cilk: cilk, spawn and sync.
   b) Using inlets.
   c) Cilk scheduler policy.
   d) Cilk evolution: Cilk++ and Intel Cilk++.

4. Debugging and profiling parallel programs (6h)
   a) Differences between debugging sequential and parallel programs.
   b) Visual debugging of parallel programs.
   c) Common kinds of bugs in parallel programs.
   d) Automatic race and deadlock detections: Valgrind (Helgrind), Intel Thread Checker.
   e) Profiling tools: OProfile, HPC Toolkit, PAPI, VTune.
   f) Intel Parallel Studio.

5. Open MP (6h)
   a) Threads and work sharing constructs: for, do, sections, single master.
   b) Data sharing clauses.
   c) Synchronization clauses: critical section, atomic, ordered, barrier, nowait.
   d) Scheduling clauses, scheduling loops to balance load, static and dynamic scheduling.
   e) Data copying and initialization.
   f) Performance considerations.

6. Open CL (4h)
   a) OpenCL execution model.
   b) Workgroups and memory model.
   c) Kernel programming, scalar and vector OpenCL types.
   d) Synchronization: queues and events.
   e) Relations between OpenCL and OpenGL.

Labs & Exercises:

- Computer lab (15h): simple programs in C/C++ corresponding to the lecture using Intel Parallel Building Blocks.
• Computer lab (7h): simple programs using OpenMP and OpenCL.
• Computer lab (8h): final project.

References:

• *Parallel Programming in OpenMP*, Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, Jeff McDonald, Morgan Kaufmann, 2000.

4.5 Distributed Programming in Erlang

Why should you learn yet another language? Isn’t Java or C++ sufficient for everything? Well, yes and no. Clearly, you can program a large-scale application in these languages. You can make it efficient. But at some point you want your application to run on multiple cores. Or — even better — on multiple computers. Thousands of them. And as the user base grows you want your system to scale without doing a major rewrite each time you add new processing power. In this case, Erlang is the language of choice. And yes, it is used in the industry.

What you will learn and gain from this course:

• How to implement applications that scale almost effortlessly.
• How to write fault-tolerant systems.
• How to use industrial-grade programming language with large and neat standard library.
• The synergy between functional languages and concurrency.

Prerequisites:

• *Programming*: basics of programming in functional languages.
• *Computer networks*: network protocols, basics of socket programming and cryptography.
• *Distributed Systems* (optional).

Contents:

1. Basics (5h)
   a) Introduction: sample programs, single assignment philosophy.
   b) Built-in data types, lists, tuples, strings, bitstrings, records, expressions.
   c) Clauses, pattern matching, functions, guards, recursion.
   d) List comprehensions: mapping, filtering, generating.
   e) Exceptions and errors: raising and handling.
   f) Preprocessing: macros, conditionals.
   g) Modules: creating, compiling, loading.

2. Concurrent and distributed programming basics (5h)
   a) Actor model, processes, message passing semantics.
   b) Underlying mechanisms: brief introduction to Erlang VM.
   c) Fault tolerance: process error handling, error processing strategies.
   d) Distributed systems recap: properties, architectures and topologies.
   e) Distributed Erlang: nodes, remote spawning, connections, authentication.
   f) Distributed computing: RPCs, master election, global namespace, partitioning, load balancing.


3. Standard library (3h)
   a) Support for data structures: lists, arrays, dictionaries, sets, trees, queues, graphs.
   b) Text processing: operations on strings, regexps, lexical analysis and parsing.
   c) Exchange formats: XML parser and ASN.1 encoder/decoder.
   d) Miscellaneous: file operations, math functions, time and calendar handling.

4. Best practices (2h)
   a) Testing: EUnit, specification based (QuickCheck), functional (Common Test).
   b) Logging importance, message priorities, reporting, storage, log analysis.
   c) Tracing: using dbg tracer, narrowing down corner cases (matching specs).
   d) Documenting code with EDoc.

5. OTP process templates (4h)
   a) Non-functional features: unified handling, fault-tolerance, on-flight updating, naming, separating implementation from behaviour.
   b) Behaviours: idea, common features, state, timeouts, callbacks, code swapping.
   c) Basic application building blocks: Server, Finite State Machine, Event Handler.
   d) OTP design principles: process trees, benefits, consequences.
   e) Supervisor behaviour: children specs, fault handling, restart policies.
   f) OTP process internals: creating behaviours conforming to OTP standards.

6. Socket programming (3h)
   a) Sockets: ownership, data handling, options, querying DNS.
   b) Client-server communication: UDP datagram and TCP stream sockets.
   c) Parsing data: built-in packet decoders (e.g. binary, line based, HTTP)
   d) Encryption recap: symmetric & asymmetric ciphering, certificates, signatures, fingerprints, crypto module.
   e) SSL/TLS sockets, handshake, session caching.

7. Databases (2h)
   a) Erlang Term Storage: ownership, searching, updating, deleting, matching queries.
   b) Disk ETS: comparison with ETS, accessing, traversing, syncing, conversion.
   c) Query List Comprehensions: handles, cursors, merging, sorting, counting.

8. Distributed DBMS: Mnesia (2h)
   a) Basics: features and restrictions, tables, operations, locks, transactions.
   b) Management: replication, indexes, checkpoints, backups.
   c) Scaling up: fragmentation, handling coherency issues.

9. Distributed application deployment and maintenance (2h)
   a) Applications: directory layout, resource file, basic ops: start, stop, restart.
   b) Application controller: handling dependencies, failovers, takeovers, heartbeat.
   c) Releases: configuration, bootstrapping, packaging, installing.
   d) Maintenance: upgrades and monitoring.

10. Interfacing (2h)
    a) Ports and linked-in drivers: comparison, ownership, basic ops: open, close.
    b) Erlang C interface: building, encoding and decoding terms, pattern matching, memory management, communication with Erlang nodes.

Labs & Exercises:
- Computer lab (20h): simple programs in Erlang corresponding to the lecture.
- Computer lab (10h): final project.
4.6 Seminar: Cache-Aware Algorithms and Data Structures

The computation model considered on basic courses on algorithms and data structures is the random access machine model, where accessing a single word takes constant time. While this abstraction is theoretically appealing, the effects of memory hierarchies (L1/L2/L3 caches, RAM, disks) cannot be neglected. In practice, it is not rare to see that a quadratic time algorithm outperforms a linear one just because it uses processor cache much more effectively. The more realistic, external memory model captures this effect by distinguishing two levels of memory: a fast one of limited capacity (e.g. M words in cache), and a slow large one (e.g. RAM). In this model, it is possible to transfer B contiguous words between layers at one fell swoop, and the complexity measure is the number of such transfers. Such assumption is motivated by the architectural choices (cache lines in processors and memory pages in operating systems), and hence the model was successfully used for developing better algorithms for the real world hardware.

The algorithms operating in such model, called cache-aware or external memory algorithms, are the central topic of this seminar. A special case of these algorithms are cache-oblivious algorithms that work well in the external memory model, but do not need a priori knowledge about the cache parameters (such as B and M). These algorithms are of particular interest, because they preserve locality, and thus scale better. The seminar is based on the Algorithms for Memory Hierarchies book and the current papers on cache-aware and cache-oblivious algorithms. Presented topics include:

- data structures: B-trees, hashing-based external dictionaries, I/O-efficient search routines, funnel heaps (priority queue), batched processing (buffer trees);
- basic techniques: external-memory sorting, graph blocking;
- algorithmic primitives on graphs: traversals, connected components, shortest paths using tournament trees, minimum spanning trees;
- text algorithms: giraffe trees (text dictionaries), skew algorithm for suffix arrays;
- lower bounds for various models.

Prerequisites:

- Algorithms and data structures: sorting, data structures (trees, priority queues).
- Computer architectures: familiarity with memory hierarchy, caches, virtual memory, TLB.

References:

4.7 Seminar: Algorithms for Massive Data Sets

Nowadays, we are facing the problem of processing enormous volumes of data. Objects such as raw data, scientific measurements, genome projects, network traffic data, market-basket data, etc., can easily take up dozens of gigabytes or terabytes. While the problems that we want to solve are casual ones, like computing statistics (e.g., median or variance), finding interesting patterns or answering some queries (e.g., does the data contain some type of objects?) the scale of these problems is unprecedented. Clearly, applying a quadratic-time algorithm is infeasible here, but also the performance gain for replacing an $O(n \log n)$ algorithm by an $O(n)$ one is non-negligible.

Further, additional restrictions on the computational model are sometimes in place. For example, it is common that our RAM cannot accommodate all the data at once. Worse than that, sometimes we can read the input only once (consider, e.g., a stream of packets flowing through router where we want to find the heavy hitter, i.e., the source of the denial-of-service attacks). In some cases, we cannot even afford to read the whole input, but still want to give approximate answer that is correct with high probability!

Problems in such computational models call for new algorithmic approaches. The seminar is based on current papers on these types of algorithms and devoted to presenting the basic tools and solutions in this area, including:

- basic tools: random sampling, hashing, random projections, Bloom filters;
- frequency estimation, heavy hitters, histogram maintenance of a stream, counting distinct elements, moments estimation;
- similarity search and clustering: k-means, k-center problems, hierarchical clustering, sampling based approaches, parallelization.
- geometric problems: dimensionality reductions, (approximate) nearest neighbor search, locality sensitive hashing;
- algorithms for compressed data;
- stochastic properties of common inputs, algorithmic implications of power law distributions.

Prerequisites:

- *Algorithms and data structures*: basic data structures, hashing.
- *Probability and statistics*: random variables, expected values, Chernoff-type bounds.

References: