



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

CONTENT

[1] Introduction to Multicore Systems



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Current trend
- Classification of parallel computer architectures
- Architecture examples for multicore platforms
- Parallel programming models & terms
- Theoretical models and consideration

[2] Overview of Parallel Programming Models



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Parallel vs. concurrent
- Level of parallelism
- Parallel Programming models
 - Independent of a specific language
 - Different ways of thinking parallelism

[3] Software Engineering



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Concurrency patterns
 - Patterns deal with the multi-threaded programming paradigm
- Case studies

[4] Parallel Design Patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Finding concurrency design space
- Algorithm structure design space
- Supporting structure design space
- Implementation mechanisms

[5] Parallel Programming in Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Basics
- Synchronization of threads
- Concurrent data types
- High-level thread management constructs
- Energy consumption of thread management constructs

[6] Parallel Programming in .Net



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Overview of .NET
- Parallel Programming in .NET
 - Win32 Application Programming Interface (Win32 API)
 - Common Language Runtime (CLR)
 - .NET Threading Library
 - Task Parallel Library (TPL)
 - Parallel Language Integrated Query (PLINQ)

[7] Parallel Programming in C/C++



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- OpenMP
- Threading Building Blocks (TBB)
- C++ 11 Concurrency

[8] Software Engineering: Testing and Debugging



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Why is debugging in parallel programs difficult?
- Race conditions
- Deadlocks
- Testing frameworks
- Happens-before relation, Lamport- & vector clocks
- Empirical studies on debugging of real-world parallel applications

[10] Optimization



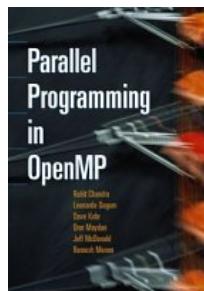
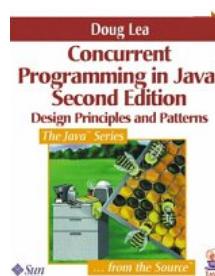
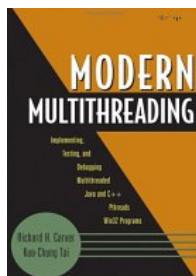
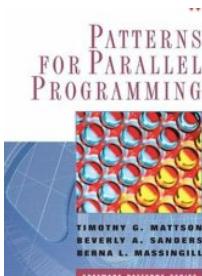
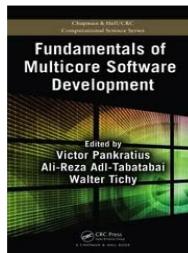
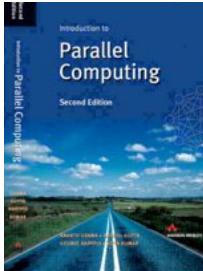
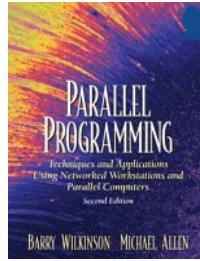
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Best practices for developing multi-threaded applications
- Compiler optimizations
- Load balancing

Literature



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Parallel Programming, Wilkinson & Allen, Prentice Hall, 2004, ISBN 978-0131405639

Introduction to Parallel Computing, Ananth Grama et al., Addison Wesley, 2003, ISBN 978-0201648652

Fundamentals of Multicore Software Development, V. Pankratius, A. Adl-Tabatabaei, W. F. Tichy , CRC Press, 2013, ISBN 978-1439812730

Patterns for Parallel Programming, Timothy G. Mattson et al., Addison-Wesley, 2004 ISBN 978-0321228116

Modern Multithreading, Richard C. Carver et al., Wiley, 2005, ISBN 978-0471725046

Concurrent Programming in Java, Doug Lea, Prentice Hall, 1999, ISBN 978-0201310092

Concurrency: State Models and Java Programs, 2nd Edition, Jeff Magee, Jeff Kramer, Wiley, 2006, ISBN: 978-0-470-09355-9

Parallel Programming in OpenMP, Rohit Chandra et al., Morgan Kaufmann, 2000 ISBN 978-1558606715

Using OpenMP, Barbara Chapman et al., The MIT Press, 2007, ISBN 978-0-262-53302-7



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

ORGANIZATION

General information



TECHNISCHE
UNIVERSITÄT
DARMSTADT

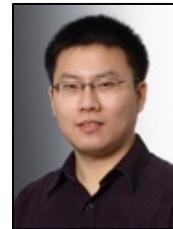
- Multicore Programming Group @ LPP
- Research objectives
 - Advanced tools & methods for multicore programming
 - Focus on General Purpose Computing (GPC)
- Multicore Programming Group



Dr. Ali
Jannesari



Rohit
Atre



Zhen
Li



Zia
Ul Huda



Mohammad
Norouzi



Arya
Mazaheri

- More Information: www.parallel.informatik.tu-darmstadt.de/multicore-group/

General information



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Location
 - CASED Building, Morneweg Str. 30, Room 3.1.05
- Lecture team
 - Dr. Ali Jannesari: jannesari@cs.tu-darmstadt.de
 - Zhen Li: li@cs.tu-darmstadt.de
 - Zia Ul-Huda: huda@cs.tu-darmstadt.de
 - Mohammad Norouzi: norouzi@cs.tu-darmstadt.de
 - Arya Mazaheri: mazaheri@cs.tu-darmstadt.de

General information



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Register for this course via tucan
 - Makes it easier for us to notify you of changes
 - Gives you access to the course material via moodle
- Schedule
 - Wednesdays, 13:30h – 15:00h (regular)
 - Last session on Feb 10
- Exercise
 - Wednesdays, 15:15h – 16:30h (biweekly)

General information



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Slides will be available in PDF in moodle
 - Only for the purpose of this class
 - Redistribution not permitted
- Requirements
 - Programming skills
 - Software engineering

Learning objectives



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Concepts and methods on software development for multicore systems
- Understanding the principles of programming on multicore systems
- Complete other lectures in the area of parallel computing

Exercises



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Exercise sheets available every tow weeks in moodle
- First exercise sheet available on Wed. Oct 21, 2015
- Exercise sheets (each person)
 - Duration: 2 weeks
- Contact
 - Zhen Li: li@cs.tu-darmstadt.de
 - Zia Ul-Huda: huda@cs.tu-darmstadt.de
 - Mohammad Norouzi: norouzi@cs.tu-darmstadt.de
 - Arya Mazaheri: mazaheri@cs.tu-darmstadt.de

Exercises solutions



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- PC-Pool Stadtmitte
 - http://www.hrz.tu-darmstadt.de/studium_lehre/pcpools/index.de.jsp
 - TU ID account for login
- TU Darmstadt SMP cluster nodes
 - How to use the cluster:
 - <http://www.hhlr.tu-darmstadt.de/hhlr/index.de.jsp>

Exam

When: Wednesday February 24, 2016 , 13:00 – 15:00

Where: S101/A1, Building Audimax

Registration via tucan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

INTRODUCTION TO MULTICORE SYSTEMS

Agenda

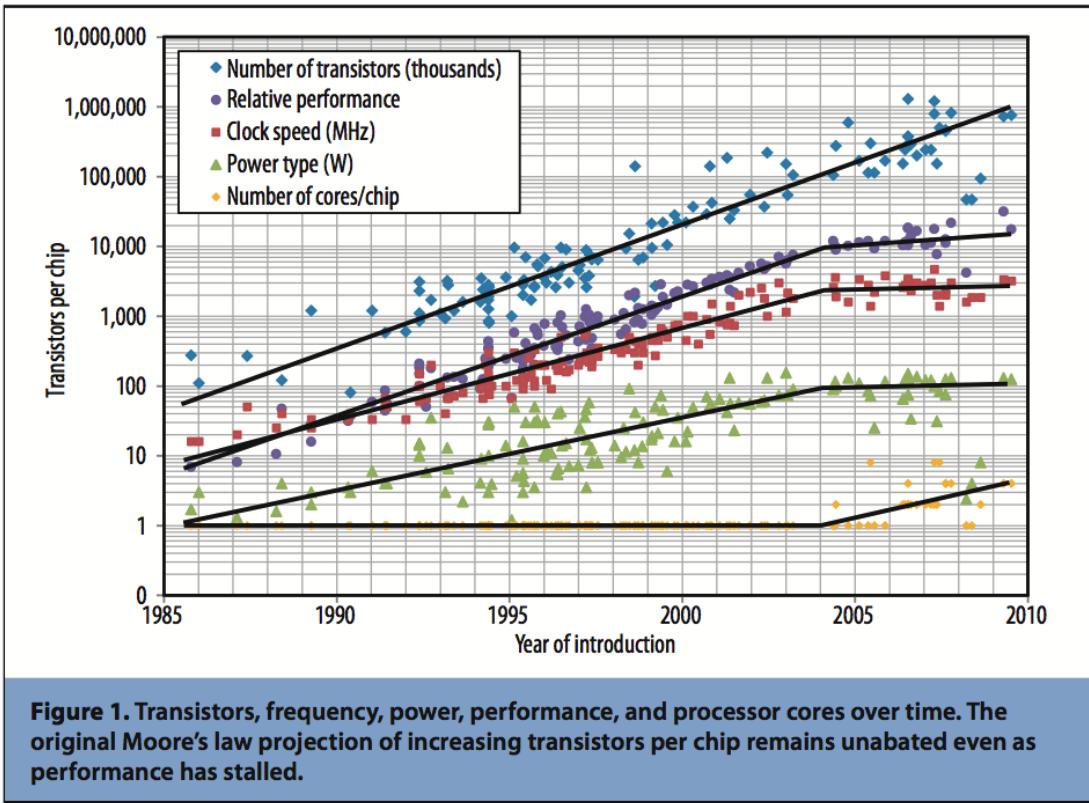


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Current Trend
- Classification of parallel computer architectures
- Architecture examples for multicore platforms
- Parallel programming models & terms
- Theoretical models and consideration

Current Trend

Transistors, Clock Rates, Energy



Source: Fuller & Millter, Computing Performance: Game Over or Next Level?, IEEE Computer 41(1), Jan. 2011

- **Power Wall:** chips get too hot!
- **Memory Wall:** not enough improvement for memory latency!
- **ILP Wall:** parallelism at instruction level is already exploited!
- **Put execution cores in one die!**

Current Trend

Emergence of Multicore



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Stagnation of single-core performance (Moore's Law)
 - Limits of improvement of single core
 - Emergence of multicore processors
- Multicore system
 - A processing system composed of two or more independent cores (or CPUs)
 - Typically integrated onto a single integrated circuit die (aka a chip multiprocessor or CMP), or
 - Integrated onto multiple dies in a single chip package
- Decreased power consumption and heat generation
- Minimized wire lengths and interconnect latencies

Current Trend Implications



TECHNISCHE
UNIVERSITÄT
DARMSTADT

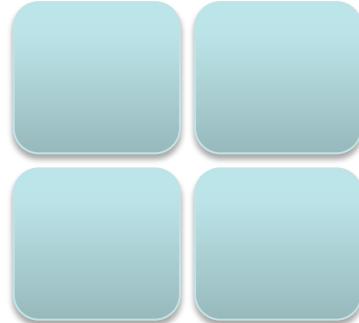
- Implicit parallelism in the processors are not enough!
 - Pipelining and prefetching
 - Dynamic scheduling
 - Very long instruction word (VLIW)
 - Take advantage of instruction level parallelism (ILP)
- Compiler optimization not enough!
- Explicit parallel programming is indispensable!
 - Particularly on higher level of abstractions

Current Trend

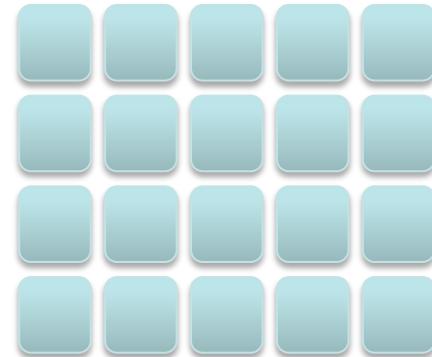
Multicore vs. Manycore



- A processing system composed of two or more independent cores integrated onto a single chip
 - Each core optimized for executing a single thread
 - Fast Serial Processing
- A manycore processor is one in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient
 - Several tens of cores (likely requires a network on chip)
 - Cores optimized for aggregate throughput
 - Deemphasizing individual performance
 - Scalable Parallel Processing (manycore assumes work load is highly parallel)



Multicore

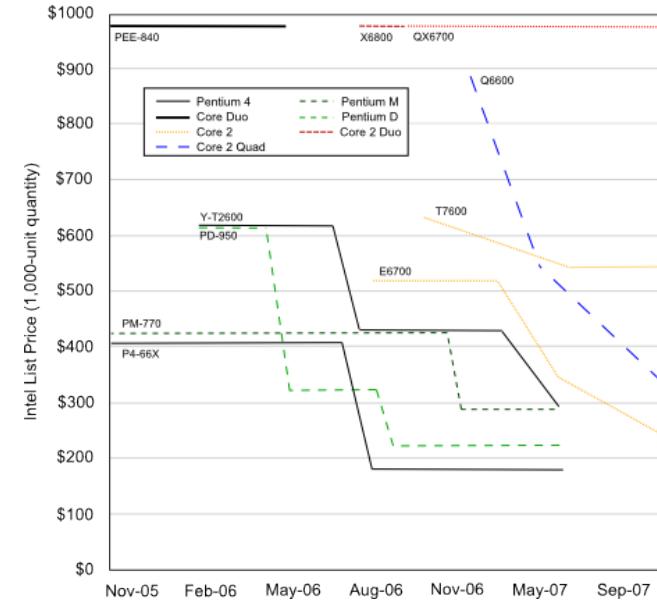
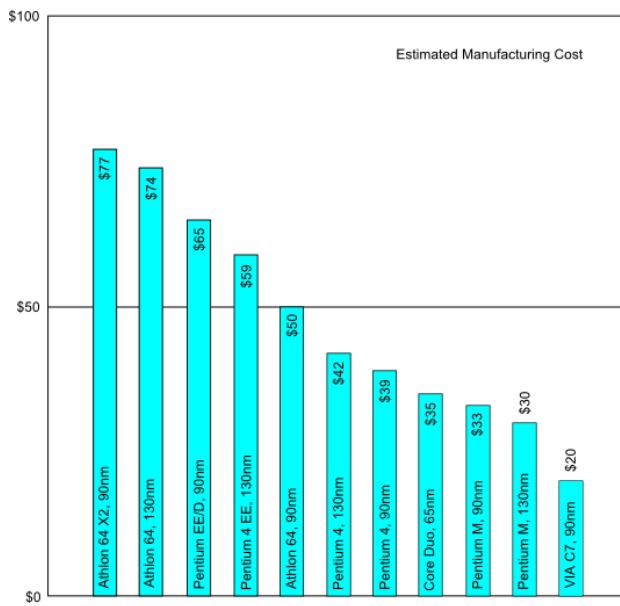


Manycore

Current Trend Price comparison



- Parallel processing is ubiquitous and available for every programmer!
- Companies produced multicore products: AMD, ARM, Broadcom, Intel, and VIA...



Source: In-Stat Microprocessor Report, Mai 2007

Classification of Parallel Computers



Control & data flow (Flynn)

- SISD: Single Instruction Stream, Single Data Stream
(e.g. serial Processor)
- SIMD: Single Instruction Stream, Multiple Data Stream
(e.g. vector computers, graphic cards)
- MISD: Multiple Instruction Stream, Single Data Stream
(e.g. systolic computer)
- MIMD: Multiple Instruction Stream, Multiple Data Stream
(e.g. multiprocessor systems, cluster, multicore
computers)

Classification of Parallel Computers



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Organization of physical memory

- Distributed memory: processes having their own Address spaces; communication by message passing (e.g. Cluster)
- Shared memory: processes can access the whole address space; communication via accesses on shared memory locations (e.g. multicore computers)
- Combinations possible (hybrid)
- Virtualization possible (logical process view of shared memory & implementation as distributed memory + additional layer)

Classification of Parallel Computers



Memory access

- Uniform Memory Access (UMA): same access times to memory for all processors
- Non-Uniform Memory Access (NUMA): Access time depends on the location
 - For example, local addresses are faster accessible
 - Cache coherent NUMA: shared caches exist; cache coherency is ensured by hardware
 - Cache Only Memory Access: for each node only exists cache memory (distributed caches form shared memory)

Symmetric Multicore Processor (SMP)

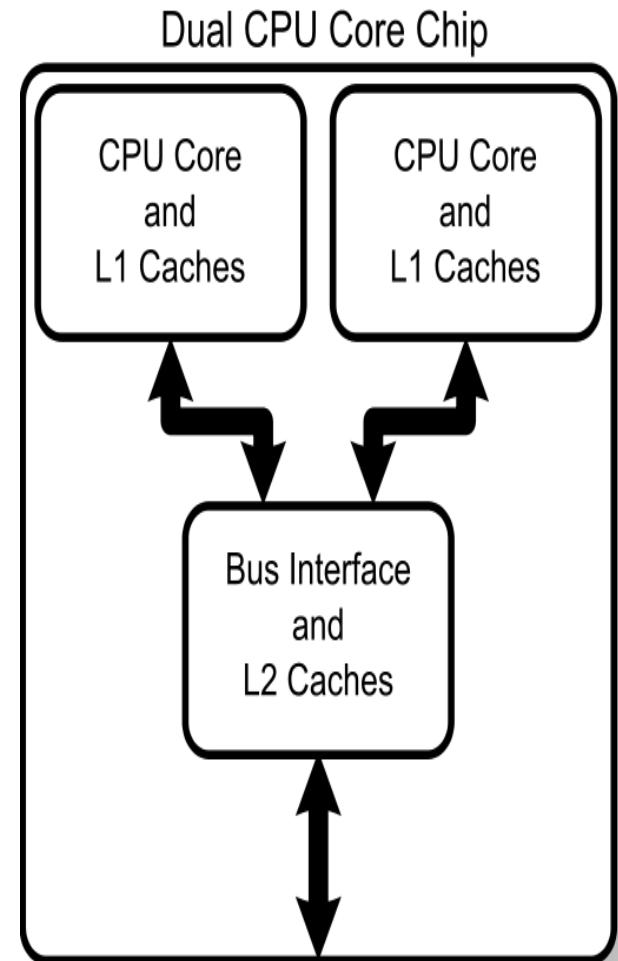


- Symmetric multi-core processor (SMP) has multiple cores on a single chip, and all of those cores are identical
 - All CPUs are equal
 - Same I/O access time
 - Same view of the overall system
- Every single core has the same architecture and the same **capabilities**
- It requires an arbitration unit to give each core a specific task
- **Multithreading** makes the best use of a multicore processors

Symmetric Multicore Processor (SMP)



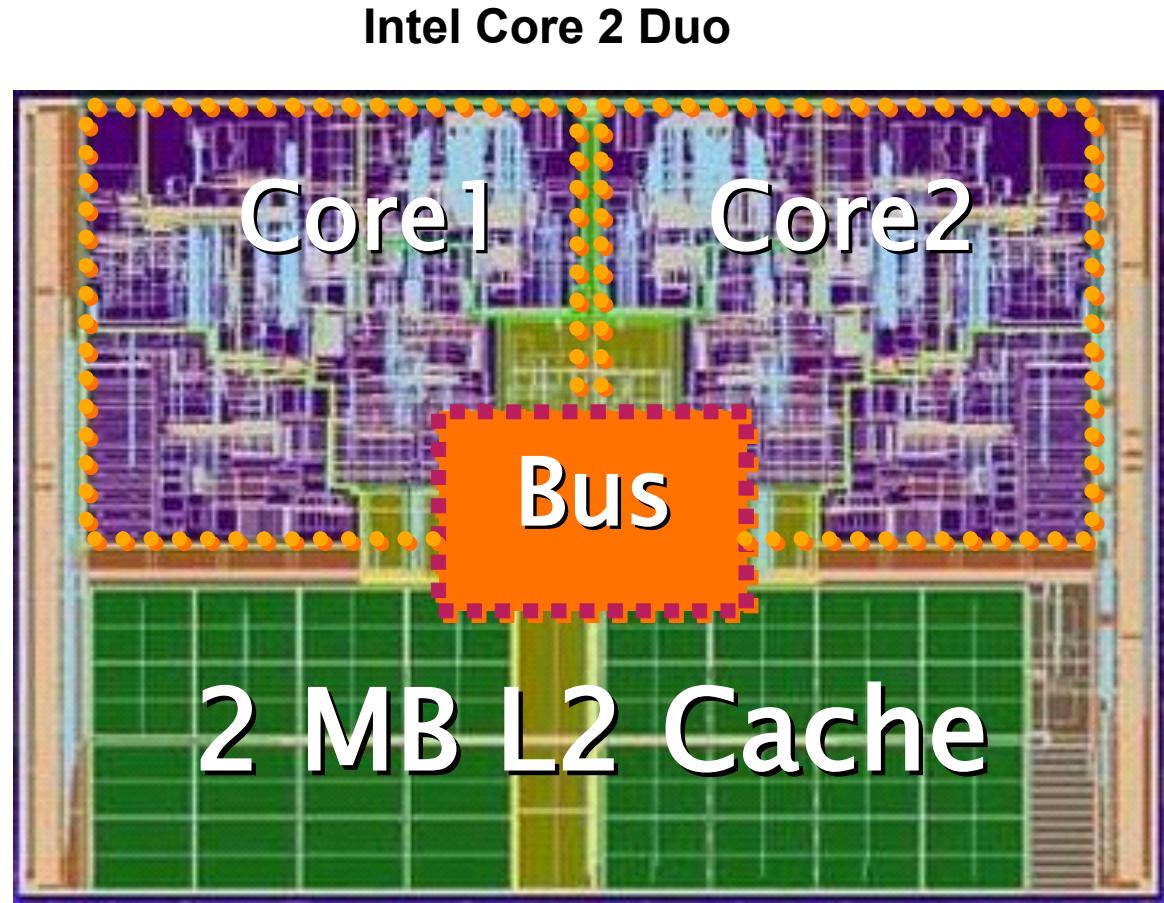
- Dual-core processor
 - Two independent microprocessors.
- The two processors are actually plugged into the **same socket**
 - Faster connection between them!
- Performance and energy
 - Ideally: a dual-core processor is nearly twice as powerful as a single core processor
 - In practice: about one-and-a-half times as powerful as a single core processor
 - A dual-core processor uses slightly less power than two coupled single-core processors



Symmetric Multicore Processor (SMP) Example



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Source: Intel

Symmetric Multicore Processor (SMP) Example



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- The Intel Core 2 can have either 2 cores on chip ("Core 2 Duo") or 4 cores on chip ("Core 2 Quad")
- Each core in the Core 2 chip is symmetrical, and can function independently of one another
- It requires a mixture of scheduling software and hardware to farm tasks out to each core

Multicore Architecture



- Implements multiprocessing in a single physical package
- Cores may be coupled together tightly or loosely
 - Cores may or may not share caches,
 - They may implement message passing or shared memory inter-core communication methods
- Common network topologies to interconnect cores:
 - Bus, ring, 2-dimentional mesh, and crossbar
- Cores may implement architectures such as:
 - Superscalar, vector processing, or multithreading

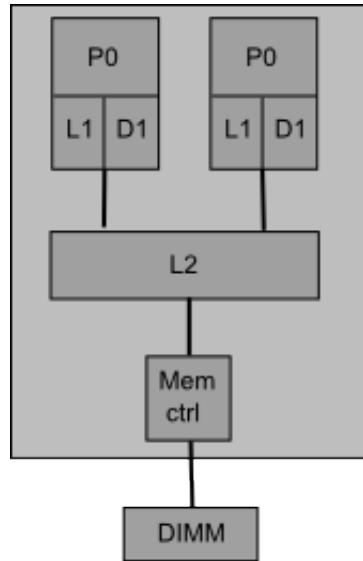
Multicore Architecture Examples (SMP)



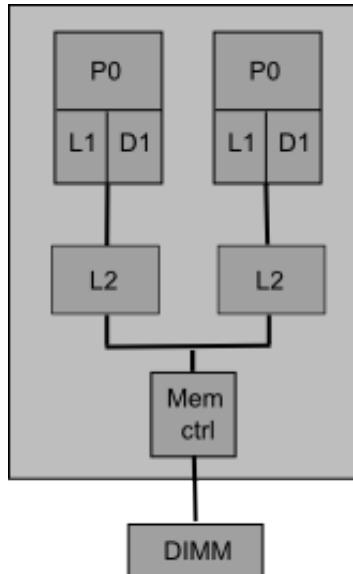
TECHNISCHE
UNIVERSITÄT
DARMSTADT

Dual Core Processors

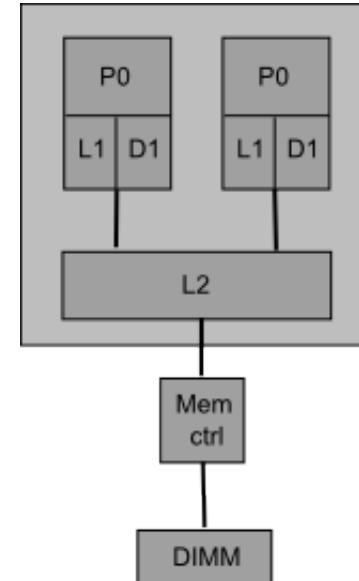
IBM Power5



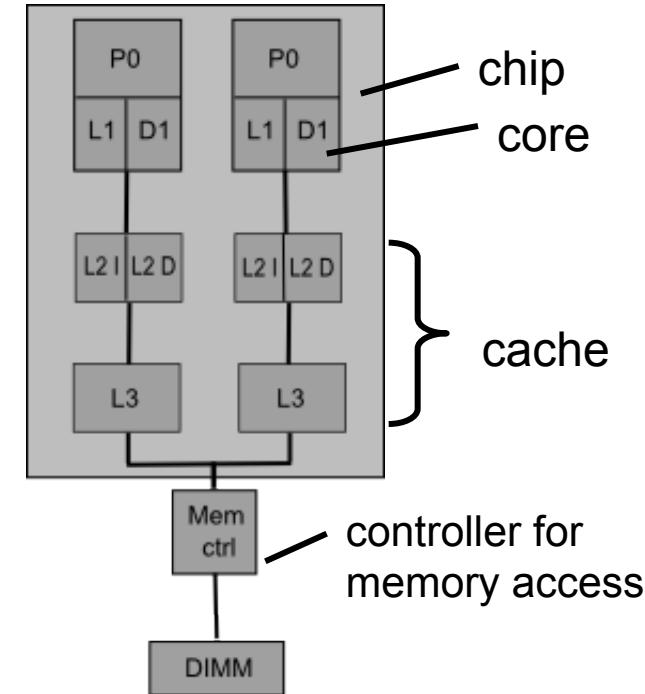
AMD Opteron



Intel Xeon



Intel Montecito



Source: N. Aggarwal et al.,
Isolation in Commodity Multicore Processors
IEEE Computer

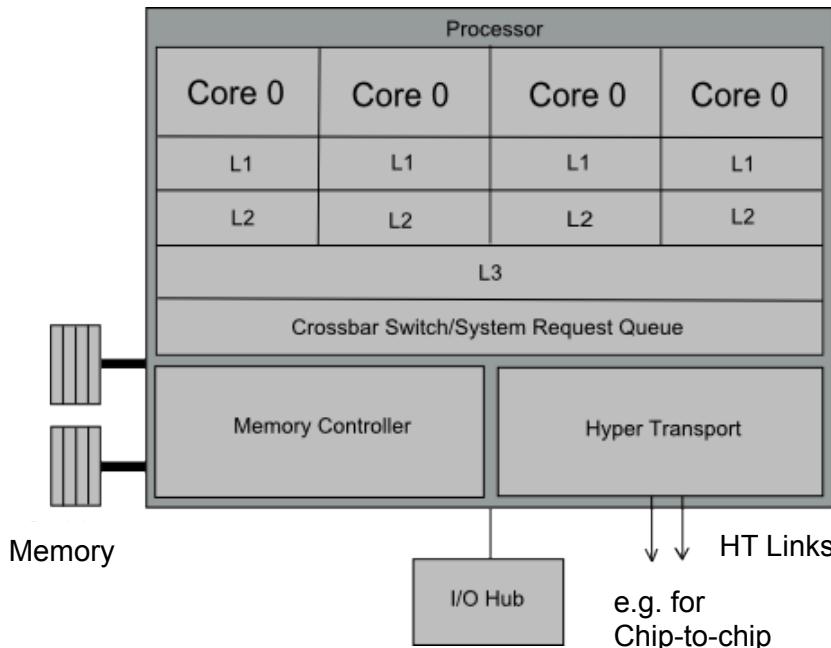
Multicore Architecture Examples (SMP)



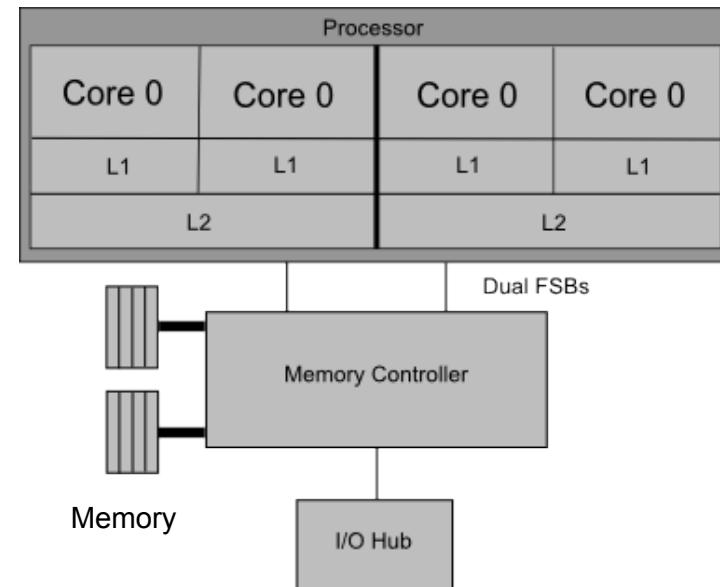
TECHNISCHE
UNIVERSITÄT
DARMSTADT

Quad Core Processors

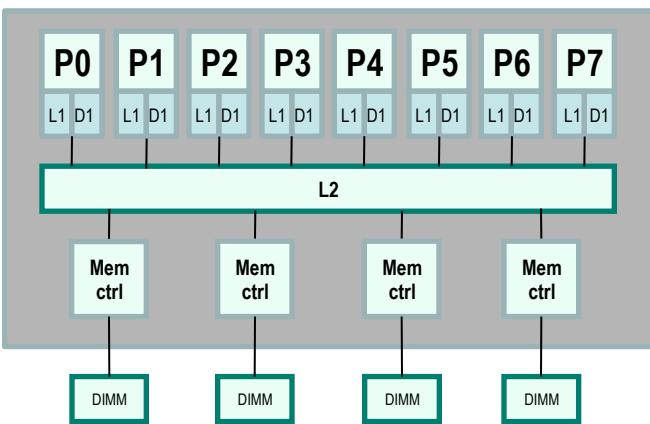
AMD Barcelona Quad Core



Intel Quad Core

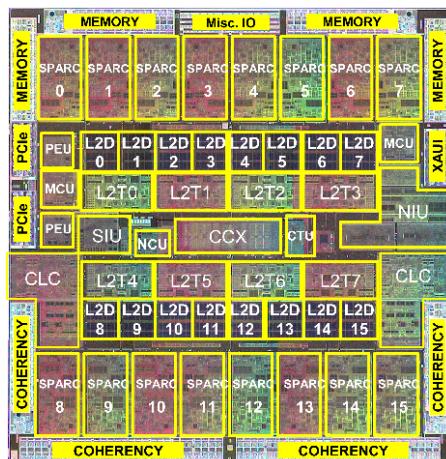


Source: J. McGregor. The New X86 Landscape Microprocessor.,



Sun Niagara (8 cores)

Sun (Oracle) Niagara (16 cores)



source: <http://arstechnica.com>

...details:

	Niagara 3	Niagara 2	Niagara 1
Launch date	2010	3. Q 2007	4. Q 2005
Clock frequency	1.6 GHz	1.4 GHz	1.2 GHz
Cores	16	8	8
Threads/core	16	8	4
Integer units	32	16	8
Floating point units	16	8	1
L2 Cache	6 MB	4 MB	3 MB

Multicore Architecture Examples (SMP)



Sun Niagara I (UltraSPARC T1)

- CC-UMA
- Symmetric Multiprocessor (SMP)



Processor	Ultra SPARC T1
Architecture	SPARC V9
Cores	8 cores running 4 threads each
Pipelines	8 integer units with 6 stages, 4 threads running on a single core share one pipeline
Clock speed	1.0 GHz
L1 Cache (per Core)	16 KB instruction cache, 8 KB data cache (4-way set-associative)
L2 Cache	3 MB on chip, 12-way associative, 4 banks
Memory Controller	four DDR2-533 SDRAM interfaces
Power Consumption	72 Watt

Asymmetric Multicore Processor (ASMP)

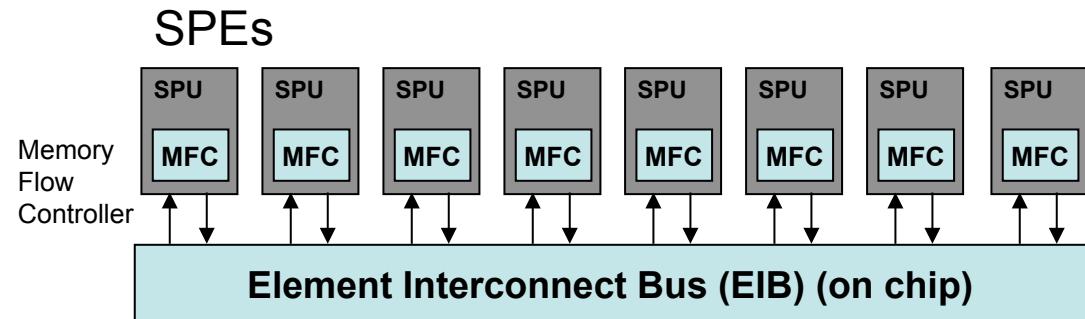


- Multiple cores on a single chip, but different designs
 - e.g. 2 general purpose cores and 2 vector cores on a single chip
- Each core will have different capabilities
- Example: IBM Cell Processor
 - 1 Power Processor Element (PPE) that controls the chip
 - 8 data-processing cores called Synergistic Processor Elements (SPEs)
 - Designed for high mathematical (also SIMD or vector operations)
 - Initiating memory transfer by themselves
 - SPEs only connect to the PPE, and not to each other
 - PPE core is much larger than the individual SPE cores

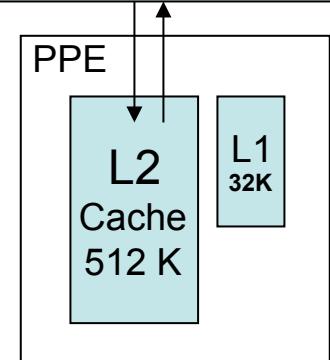
Asymmetric Multi-core Processor (ASMP) Example



- IBM Cell Broadband Engine (Cell Processor 9 cores)
 - 1 Power Processor Element (PPE)
 - 8 Synergistic Processor Elements (SPEs)



- Bridged the gap between conventional desktop processors (such as Core 2 families) and more specialized high performance processors (such as Graphics processors - GPUs)
 - Used in HD displays, recording equipment and entertainment systems



SMP vs. ASMP

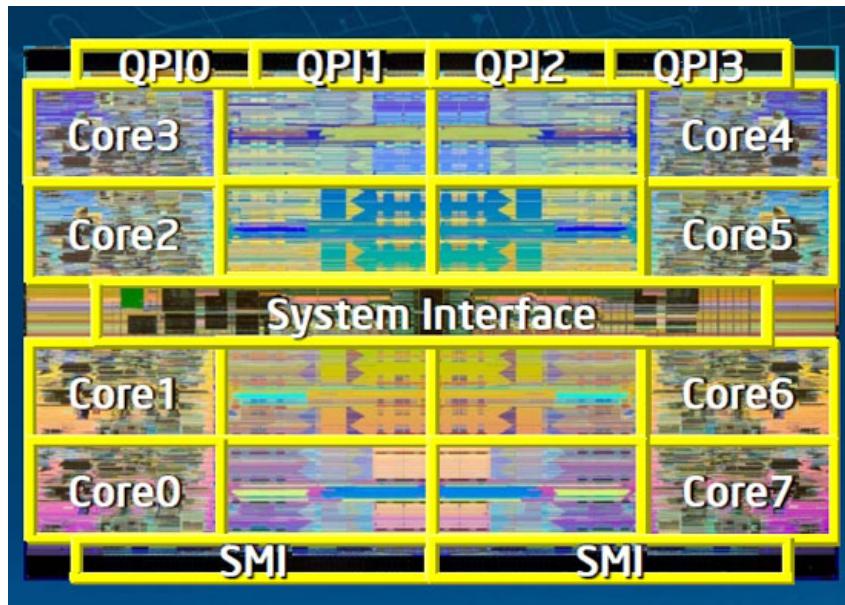


- SMP
 - Easy to implement (lots of cores put in one integrated circuit)
 - Easier programming than ASMP (all the cores are identical)
 - Easy to keep the development speed
 - Apply to any type of system (general usage)
- ASMP
 - Suitable and efficient for **certain specific** system (reason why ASMP has emerged!)
 - Audio/video processing, data compression and etc.
 - Does not waste the silicon and power, since is not made for the general purpose

Multicore Architecture Examples



Intel Nehalem EX 8 Core



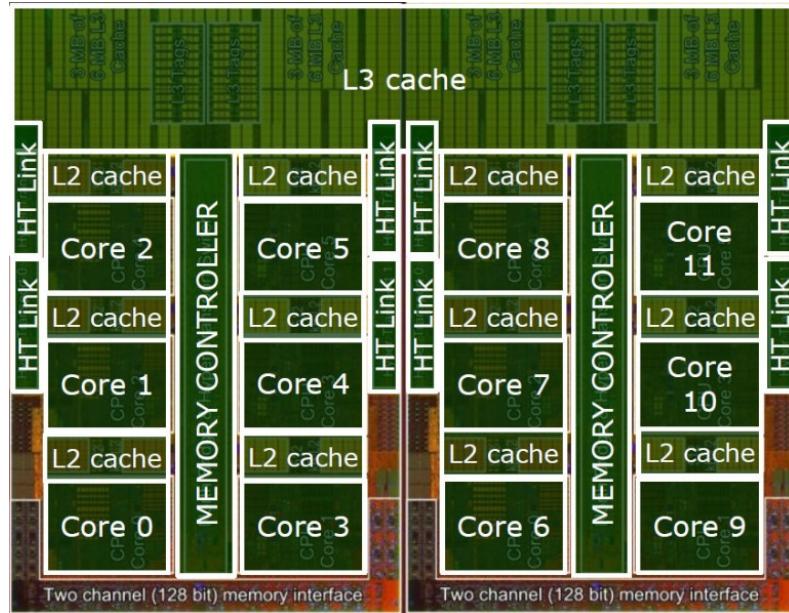
(Source: www.intel.com, April 2010)

- 8 cores (16 threads)
- 24 MB shared L3 cache
- Integrated memory controller
- 4 Quick Path Interconnect (QPI) links
 - Bidirectional, 20 Bit Bus for the communication between CPU und Chipset
 - Replacing front side bus (Bottleneck!)
- 2,3 billion transistors

Multicore Architecture Examples



AMD Opteron 12 Core



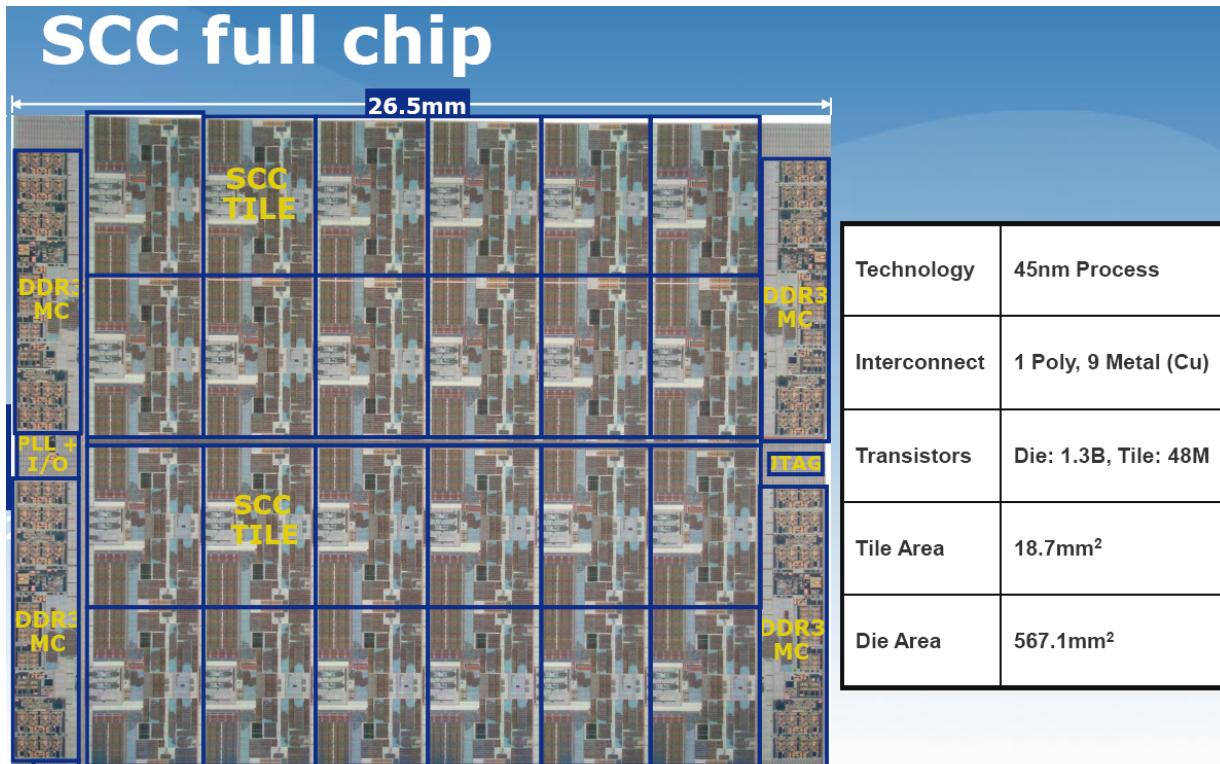
- 12 cores
- 12 MB shared L3 cache
- Memory controller
- Hyper Transport 3 (HT3)
 - competitive Technology to Intel's QPI

(Source: www.amd.com, April 2010)

Multicore Architecture Examples



Intel „Single Chip Cloud Computer“ (48 cores)



Source:
[http://techresearch.intel.com/
UserFiles/en-us/File/terascale/
SCC_Symposium_Feb212010_
FINAL-A.pdf](http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC_Symposium_Feb212010_FINAL-A.pdf)

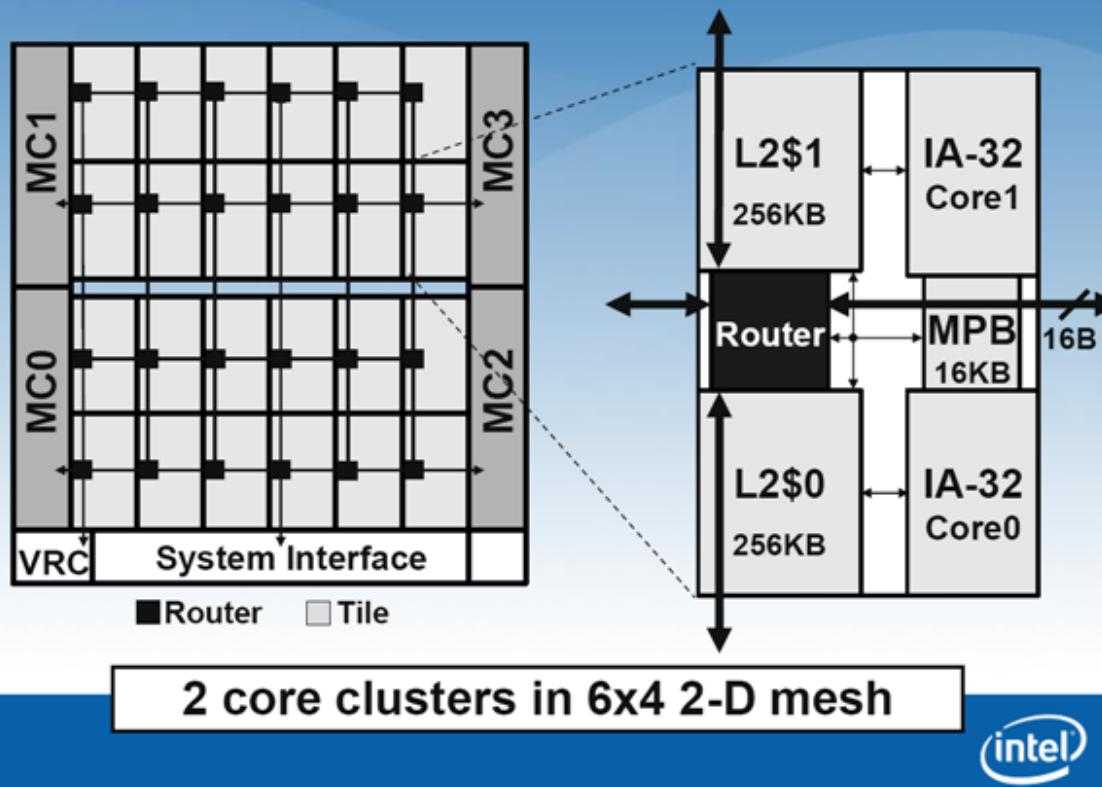


Multicore Architecture Examples



Intel „Single Chip Cloud Computer“ (48 cores)

Architecture



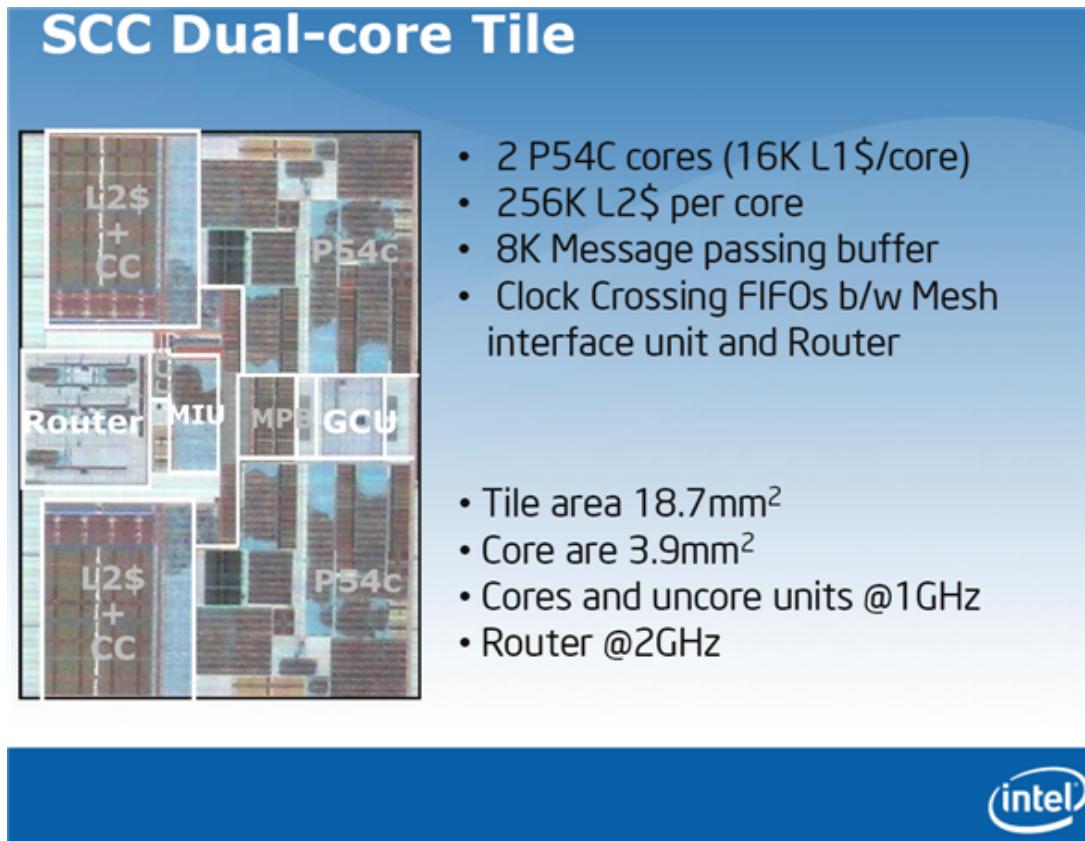
Source:
[http://techresearch.intel.com/
UserFiles/en-us/File/terascale/
SCC_Symposium_Feb212010_
FINAL-A.pdf](http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC_Symposium_Feb212010_FINAL-A.pdf)

Multicore Architecture Examples



TECHNISCHE
UNIVERSITÄT
DARMSTADT

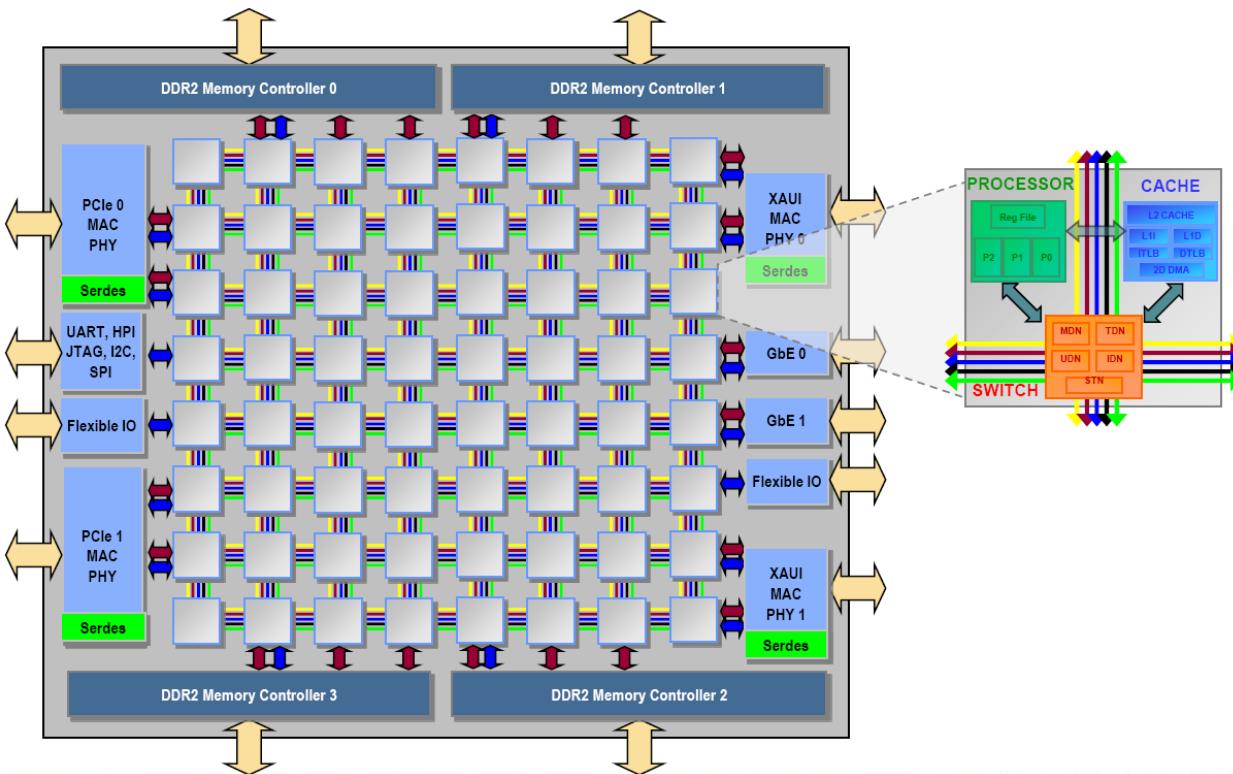
Intel „Single Chip Cloud Computer“ (48 cores)



source:
[http://techresearch.intel.com/
UserFiles/en-us/File/terascale/
SCC_Symposium_Feb212010_
FINAL-A.pdf](http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC_Symposium_Feb212010_FINAL-A.pdf)

Multicore Architecture Examples

Tilera Tile64 (64 cores)



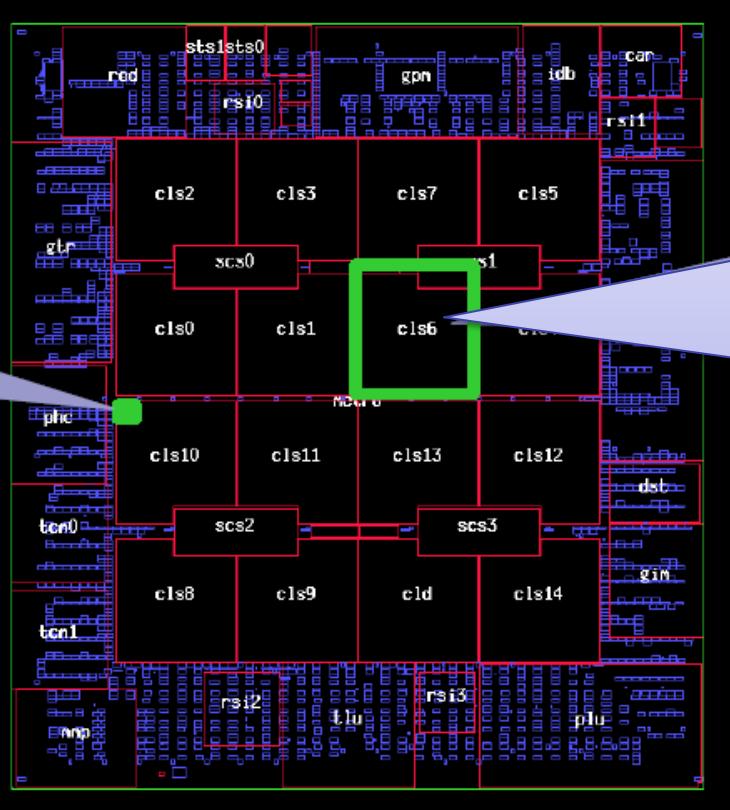
- 64 homogenous cores
- 750 MHz
- Cache: totally 5MB (distributed on chip)
 - Each tile has its own L1+L2 Cache
- 5 independent Networks:
 - System + I/O
 - Cache misses, DMA
 - Tile-to-Tile memory access
 - User-level

Source: Tilera

Multicore Architecture Examples



Cisco Metro (192 cores)

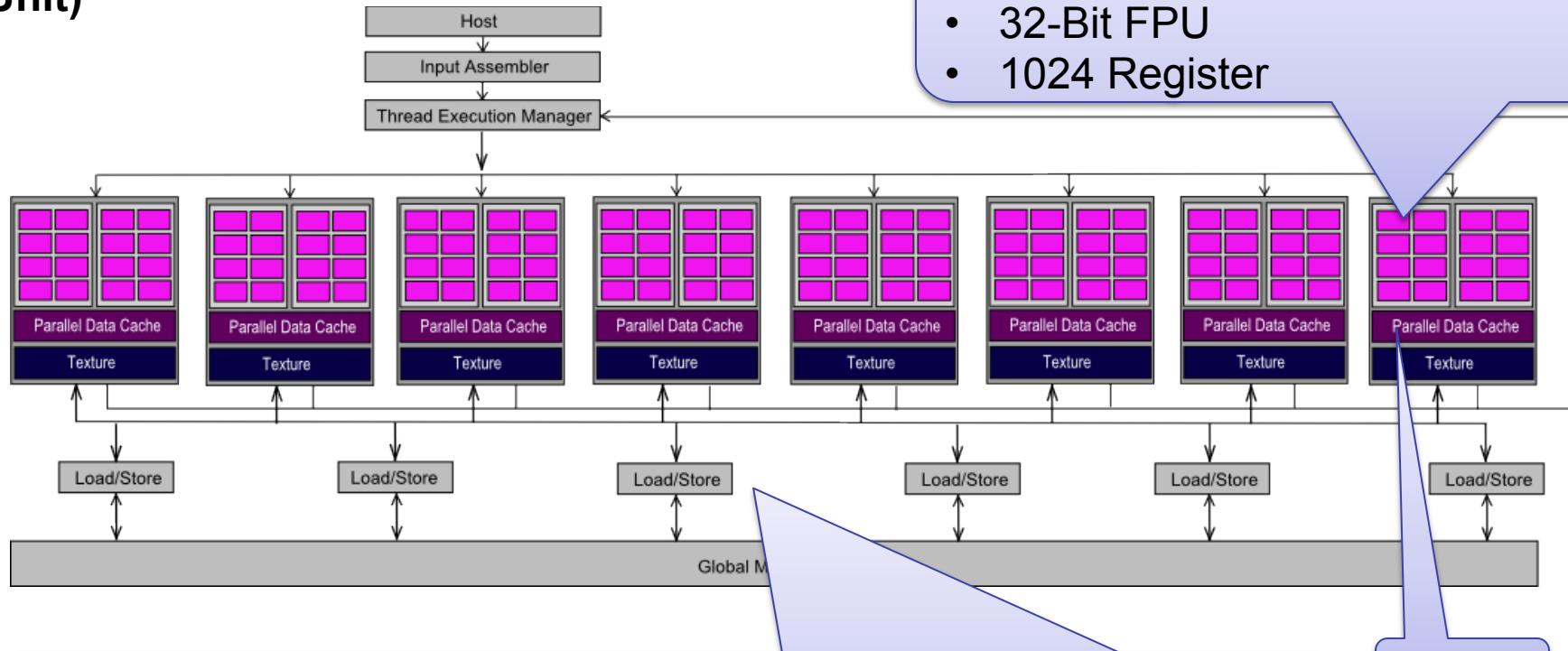


- 16 Groups
- Each 12 Processors
- Overall: 192 Tensilica Processors @ 250 MHz
- (2005)

Multicore Architecture Examples (Manycore / Accelerators / co-processors)



Nvidia GeForce (8 Graphics Processing Unit)



- 128 Processors (16 Processors each unit), each processor has 96 hardware threads, overall 12288 HW threads!
- Thread management widely automatic

Multicore Architecture Examples (Manycore / Accelerators / co-processors)

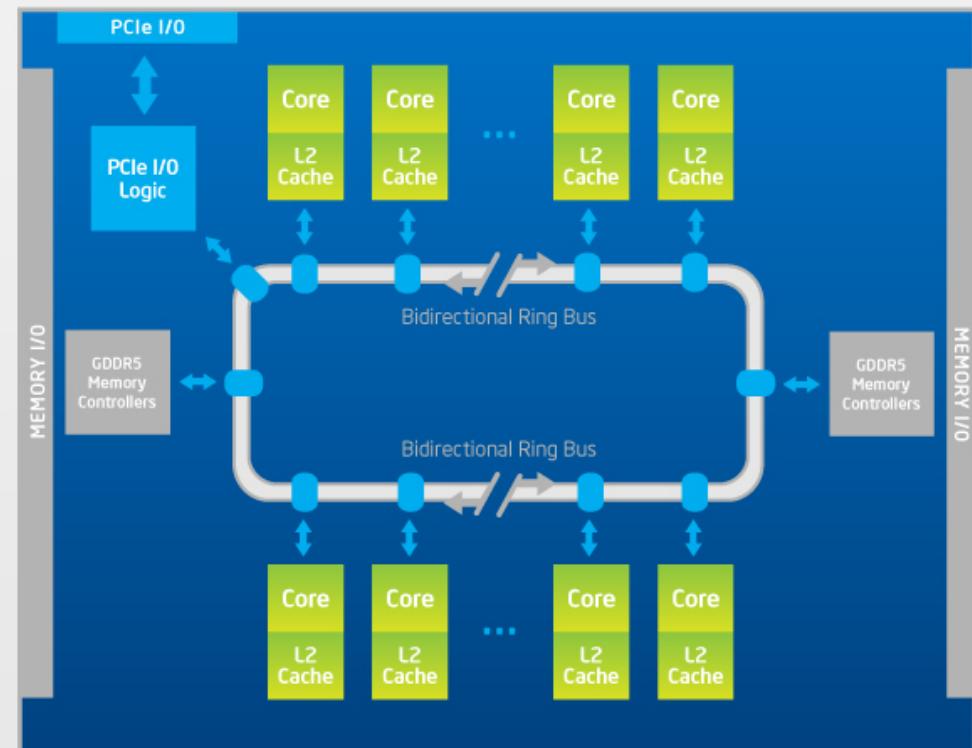


TECHNISCHE
UNIVERSITÄT
DARMSTADT

Xeon Phi family of processors

- Knights Ferry (May 2010)
 - 32 cores
 - up to 750 GFLOPS
- Knights Corner (Nov. 2011)
 - 60 cores
 - up to 1.2 TFLOPS
- Knights Landing (June 2013)
 - 72 cores
 - up to 3 TFLOPS!!!

Intel® Xeon Phi™ Coprocessor Block Diagram



Source: Intel

Challenges for Multicore Software



- Program relies on effective exploitation of multi-threaded parallelism
 - Need for parallel computing model and parallel programming model
 - Some video games will run faster on a 3 GHz single-core than on a 2GHz SMP dual-core (**not efficient parallelism!**)
 - Memory accesses (memory bandwidth and memory latency)
 - Synchronization and communication
 - Cache effects (fragments in cache)

The **difficult problem** is not building multicore hardware, but **programming!**

Parallel Programming Models & Terms

Two important parallel programming models:

- **Parallel computers with shared memory**
 - Process and threads
 - Applicable for **multicore computers**
- Parallel computers with distributed memory (out of the scope of this lecture!)
 - Message passing
 - Applicable for cluster computers



Process

- Created by OS
- Contains information about program resources and execution state
 - Process ID
 - Code segment (program instruction)
 - Data segment (for global variables)
 - Contains at least 1 thread
 - ...
- A fair amount of overhead for CPU context switching between processes!



Thread

- Independent set of instructions that can be scheduled to execute
- Exists within a process
- Has its own
 - Program counter, stack pointer, copy of registers and scheduling properties
- Shares with peer threads
 - Address space, code/data section and other resources (e.g. open files, locks, etc.)
 - CPU context switching between threads **less expensive**



Basic procedure in shared memory

- Threads contain parallel executing tasks (instructions)
- Exchange of data (communication btw. threads) via shared variables in memory
- Synchronization constructs coordinate execution in the case of data or control dependencies
- Threads are basically created by OS and distributed to processors or cores
 - Provided by some interfaces in programming languages or libraries (e.g. Pthreads, OpenMP)

Theoretical Models and Considerations



TECHNISCHE
UNIVERSITÄT
DARMSTADT

When does parallelization pay off?

- Consider a program with:
 - A sequential fraction of code, which can not be parallelized, and
 - A parallelizable remaining that can be divided on multiple homogeneous processors (cores)
 - σ: Execution time of the sequential part
 - π: Execution time of the parallel part
- The **overall execution time** $T(p)$ for p processors (or cores):

$$T(p) = \sigma + \frac{\pi}{p}$$



When does parallelization pay off?

- The **Speedup** $S(p)$ indicates how much faster is the program with p processors compared to the best sequential execution (1 processor):

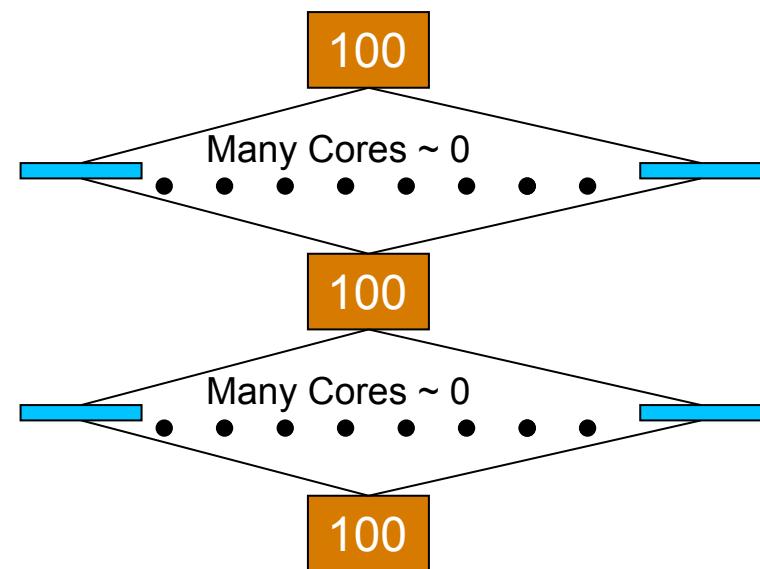
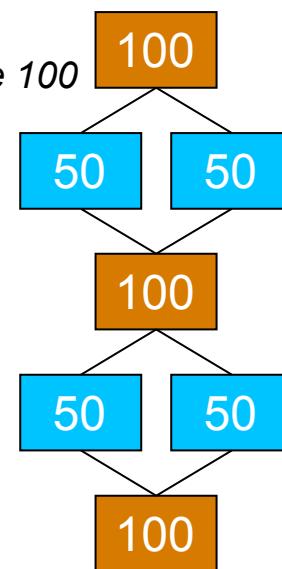
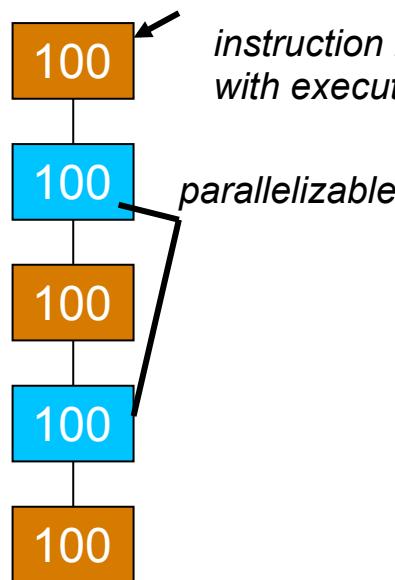
$$S(p) = \frac{T(1)}{T(p)}$$

- Ideal scenario: $S(p) = p$

Theoretical Models and Considerations



Speedup – Example



Theoretical Models and Considerations



- Consider relative **proportions**
 - Let f be the proportion of the **sequential fraction of code** i.e. not parallelizable part: $f = \frac{\sigma}{\sigma+\pi}$, $0 \leq f \leq 1$

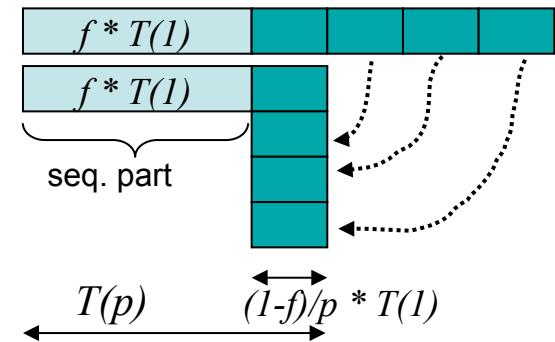
$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{\underbrace{f*T(1)+\frac{1-f}{p}*T(1)}_{\text{min. exe. time seq. part}} + \underbrace{\frac{1-f}{p}*T(1)}_{\text{min. exe. time par. part}}} = \frac{1}{f+\frac{1-f}{p}} \leq \frac{1}{f} \quad (p \rightarrow \infty)$$

min. exe. time
seq. part



1 processor:
many processors:

seq. part



Theoretical Models and Considerations



Amdahl's Law:

$$S(p) \leq \frac{1}{f}$$

- States that the potential program speedup is defined by the fraction of code (f) that cannot be parallelized:
 - If none of the code can be parallelized, $f = 1$ and the speedup = 1 (no speedup)
 - If all of the code is parallelized, $f = 0$ and the speedup is infinite (in theory)
 - i.e., maximum speedup is limited by the sequential part!



Communication Time (T_c)

- Additional time T_c for communication between Processors
 - σ : execution time for the sequential part
 - π : execution time for the parallel part
 - $T_c(p)$: time due to communication overhead (assumption of linear increase with the number of processors p)
- The **overall execution time** $T(p)$ for p processors (or cores):

$$T(p) = \sigma + \frac{\pi}{p} + T_C(p)$$



Communication Time (T_c)

- Relation (r) between minimum communication time and sequential computation time
 - How much time overhead through communication?

$$r = \frac{T_C(2)}{T(1)}$$

- Assumption: $T_C(2)$ is the minimum communication time

Theoretical Models and Considerations

Speedup considering communication:

$$S(p) = \frac{1}{f + \frac{1-f}{p} + r*(p-1)}$$

No self-communication



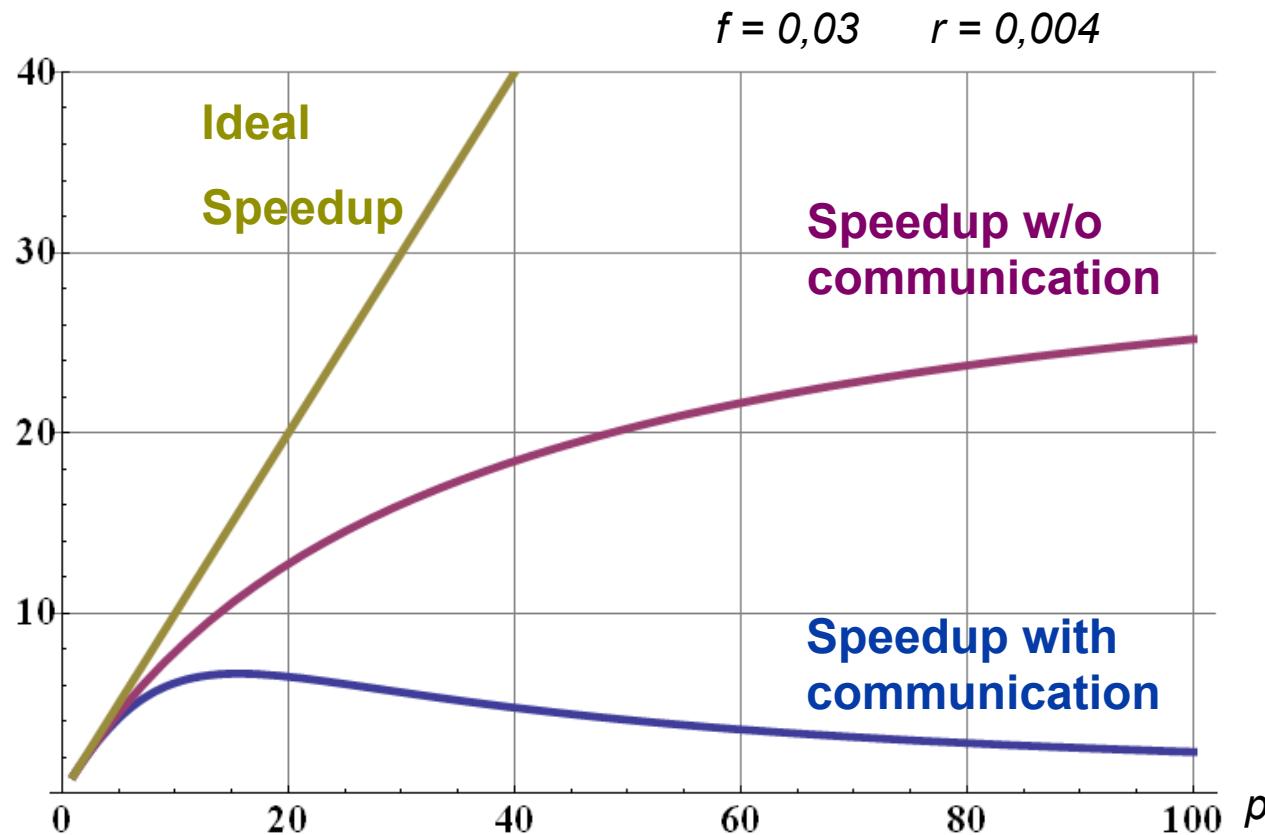
- Observations:
 - For large p , $S(p)$ decreases!
 - Too much parallelism can be counterproductive!
 - Maximum parallelism by applying theory of curves (curve tracing/sketching):

$$p^* = \sqrt{\frac{1-f}{r}}$$

Theoretical Models and Considerations



Speedup considering communication – Example



Theoretical Models and Considerations



- Empirically specify the sequential code fraction f of a parallel program
 - Using the speedup equation (without communication)
 - f should be constant as expected $S(p) = \frac{1}{f + \frac{1-f}{p}}$ $\rightarrow f = \frac{1/S(p)-1/p}{1-1/p}$
- Empirical experiments:
 - Calculate $S(p) = T(1) / T(p)$ for several p values and then compute f
 - f should be independent of p , if communication or load balancing (workload distribution) is neglected
 - Dependence of f on p indicates e.g. overhead of parallelism (communication) or unbalanced load distribution

Theoretical Models and Considerations



- Amdahl's approach consist in solving a **fixed-size** problem (a fixed data set size) faster through parallelization!
 - Amdahl's Law seems to be against massively parallel processing
 - Assumes the overall workload of a program does not change with respect to machine size
- Gustafson's argument
 - With powerful computers **bigger** problems are also processed
 - Large data sets can be efficiently parallelized, An example:
 - More powerful graphics cards increases the pixel resolution or complexity of objects in computer games
 - Although there exist some limits, but also there are additional potential parallelisms

Theoretical Models and Considerations



- Gustafson's law

$$S(P) = P - \alpha.(P - 1)$$

- P is the number of processors
- S is the speedup,
- α is the largest non-parallelizable fraction of any parallel process
- Exploiting the computing power that becomes available as the number of machines increases
- Scaled speedup (scales with P)
- More on discussion Amdahl / Gustafson, see :

http://spartan.cis.temple.edu/shi/public_html/docs/amdahl/amdahl.html



Super Linear Speedup

- In theory impossible, because
 - In principle, each parallel algorithm that solves a problem in time $T(p)$ with p processors, can be simulated by a sequential algorithm in time $T(1) = p * T(p)$
 - $T(1)$ is the execution time for the best sequential algorithm
 - Speedup = $T(1) / T(p) = p * T(p) / T(p) = p$
- However in practice, due to various effects it can be seen!
 - How can it happen? randomly?



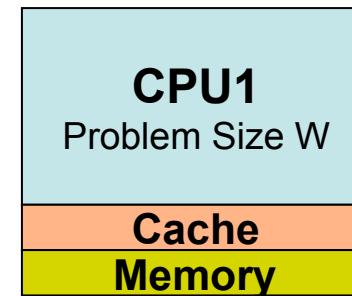
Super Linear Speedup – Example: Cache Effects

- Cache effects could be the reason for a super linear speedup
 - If the amount of data per processor or core becomes smaller through parallelization, more data blocks fit in each cache line
 - The cache hit rate is increased compared to the single processor!
 - As a result, the execution time on the individual processors decreases
 - Frequent memory accesses can be saved (assuming no loss of communication, etc.)

Theoretical Models and Considerations

Super Linear Speedup – Example: Cache Effects

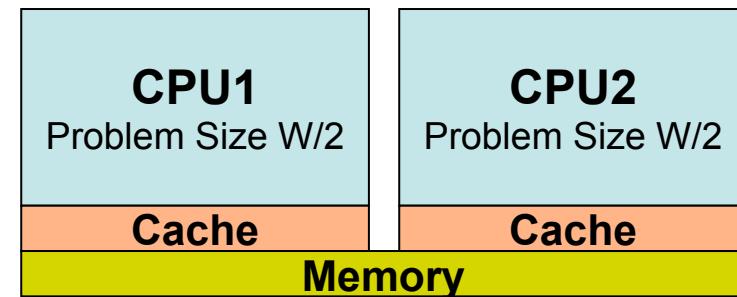
- Assumptions (single-core):
 - Cache hit rate: 80%
 - Cache access time: 2ns
 - Memory access time: 50ns
 - Expected value access time: $0.8 * 0.2\text{ns} + (1 - 0.8) * 50\text{ns} = 11.6\text{ns}$
- Program executes 1 billion memory accesses (and nothing else):
 - Total execution time **T (1): 11,6s**





Super Linear Speedup – Example: Cache Effects (cont.)

- Assumptions (dual-core):
 - Cache hit rate: **90%**
 - Cache access time: 2ns
 - Memory access time: 50ns
 - Expected value access time: $0.9 * 0.2\text{ns} + (1 - 0.9) * 50\text{ns} = 6.8\text{ns}$
- Program executes now 500 million memory accesses per CPU (assuming divisible):
 - Total execution time **T (2)**: **3,4s**
 - **Speedup T(1)/T(2) = 11,6s/3,4s = 3,4**

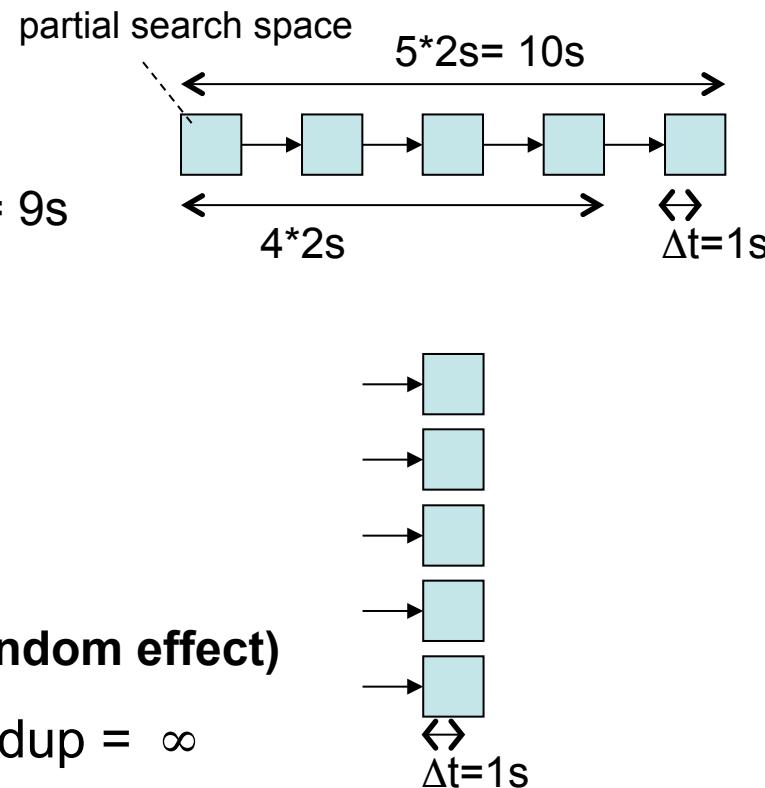


Assumption: no communication overhead and the results can be calculated separately.

Super Linear Speedup – Example: Random Effects

(Search Algorithms)

- Sequential search
 - Value found after: $8s + 1s = 9s$
 - Close to worst case
- Parallel search
 - 5 cores (CPUs)
 - Value found after: 1s
 - Speedup $T(1)/T(5) = 9$ (**random effect**)
- Special case 1: $\lim_{\Delta t \rightarrow 0}$ Speedup = ∞





Super Linear Speedup – Example: Random Effects (Search Algorithms)

- Special case 2: The sequential search finds the value in the first partial search space after 1s (best case)
 - Speedup $T(1)/T(5) = 1$
- Average case: both parallel and sequential algorithms find the value in the middle
 - Seq. time = 5s, par. time= 1s
 - Speedup $T(1)/T(5) = 5$
- Conclusion: a parallel program is not necessarily buggy by having a super linear speedup!

Efficiency

$$E(p) = \frac{T(1)}{p*T(p)} = \frac{S(p)}{p}$$

- Indicates proportion of time spent by each processor with useful work (how well processors are utilized)
- Ideal case: $S(p) = p$, i.e. $E = 1$
- In practice: $E < 1$
 - Reasons could be that each processor waits for others or sends data to others (communication/synchronization overhead)



Software Engineering for Multicore Systems

Dr. Ali Jannesari

OVERVIEW OF PARALLEL PROGRAMMING MODELS

Outline



- Parallel vs. Concurrent
- Level of Parallelism
- Parallel Programming Models
 - Threads and Locks
 - Functional Programming
 - Actors
 - Communicating Sequential Processes
 - Data Parallelism
 - Others



Parallel vs. Concurrent

- Often used interchangeably
- Refer to related but different things

A *concurrent* program has multiple logical *threads of control*. These threads may or may not run in parallel.

A *parallel* program potentially runs more quickly than a sequential program by executing different parts of the computation simultaneously. It may or may not have more than one logical thread of control.

Parallel vs. Concurrent (2)

- Concurrency is an aspect of the problem domain – your program needs to handle multiple simultaneous (or near-simultaneous) events.

*“Concurrency is about **dealing** with lots of things at once.” [1]*

- Parallelism is an aspect of the solution domain – you want to make your program faster by processing different portions of the problem in parallel.

*“Parallelism is about **doing** lots of things at once.” [1]*

[1] Rob Pike, “Concurrency is not Parallelism”. <http://concur.rspace.googlecode.com/hg/talk/concur.html>

Parallel vs. Concurrent (3)

- Concurrency and parallelism are often confused because traditional threads and locks do not provide any direct support for parallelism.
- Solution: Create a concurrent program and run it on parallel hardware.
- Concurrent programs are often nondeterministic. Parallelism, by contrast, does not necessarily imply nondeterminism.

Level of Parallelism



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Bit-level parallelism

A 32-bit computer is faster than an 8-bit one.

- Instruction-level parallelism

- Pipelining

- Out-of-order execution

- Speculative execution

- Data parallelism

- SIMD, GPU, etc.

- Task-level parallelism

- Multiprocessor.

Classification of Parallel Programming Models



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- By process interaction:
 - Shared memory
 - Message passing
 - Implicit
- By problem decomposition:
 - Data parallelism
 - Task parallelism
 - Idealized parallel systems

Threads and Locks



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Like a Ford Model T
 - It will get you from point A to point B, but it is primitive, difficult to drive, and both unreliable and dangerous to newer technology.
- The base of other models
- Little more than a formalization of what the hardware does
- Supported by almost all languages
- Still the default choice in some occasions

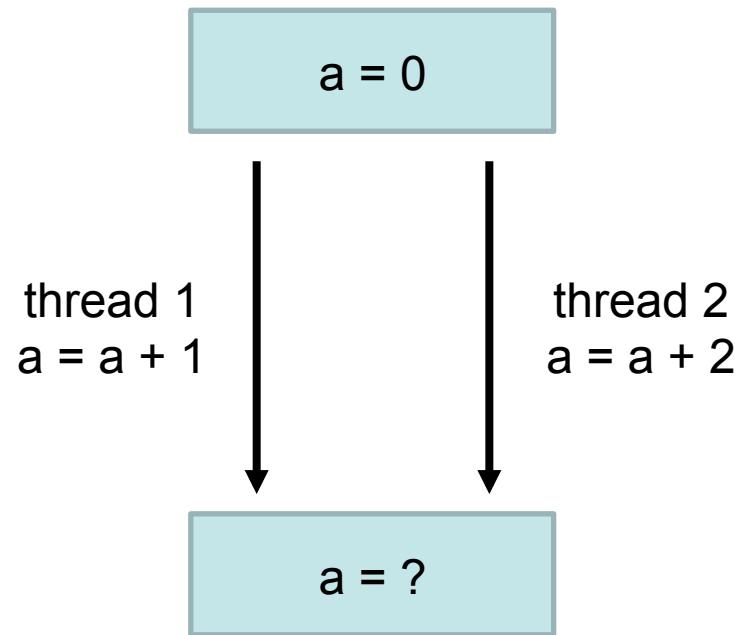
Examples shown in “Threads and Locks” are in C++.



- A thread object acts as a handle to an underlying software thread (OS thread), which is further mapped to a hardware thread.
- Software threads: threads managed by OS across all processes and scheduled for execution on hardware threads. Typically, the number of software threads is bigger than the number of hardware threads.
- Hardware threads: the actual threads performing computation. Contemporary architectures offer one or more hardware threads per CPU core.

A Basic Problem

- Threads communicate with each other via shared-memory
- What if two threads write to the same shared variable at the same time?



Locks



```
#include <iostream>          // std::cout
#include <thread>            // std::thread
#include <mutex>              // std::mutex

std::mutex mtx;             // mutex for critical section

void print_thread_id (int id) {
    mtx.lock();              // exclusive access to std::cout by locking mtx
    std::cout << "thread #" << id << '\n';
    mtx.unlock();
}

int main () {
    std::thread threads[10];
    for (int i=0; i<10; ++i)
        threads[i] = std::thread(print_thread_id,i+1);
    for (auto& th : threads) th.join();
    return 0;
}
```

Mutual Exclusion



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Critical section

A critical section is a block of code that can be executed by only one thread at a time.

- Mutual exclusion

Mutual exclusion is a property of a program stating that critical sections of different threads do not overlap.

Starvation, Deadlock, and Livelock



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Starvation**

A thread is in starvation if it attempted to acquire a lock but never succeeds.

- **Deadlock**

A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.

- **Livelock**

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

Condition Variables



A condition variable is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until:

- a notification is received from another thread*, or
- a timeout expires

* can be a spurious wakeup

Condition Variables (2)



```
#include <mutex>           // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id (int id) {
    std::unique_lock<std::mutex> lck(mtx);
    while (!ready) cv.wait(lck);
    // ...
    std::cout << "thread " << id << '\n';
}

void go() {
    std::unique_lock<std::mutex> lck(mtx);
    ready = true;
    cv.notify_all();
}
```

Atomic Variables



- An operation acting on shared memory is **atomic** if it completes in a single step relative to other threads.

```
#include <atomic>

struct AtomicCounter {
    std::atomic<int> value;

    void increment(){ ++value; }

    void decrement(){ --value; }

    int get(){
        return value.load();
    }
};
```

Memory Order



It is an illusion that sequential programs run sequentially because:

- Compiler is allowed to statically reorder instructions
- If the program relies on a runtime framework, the runtime is allowed to dynamically reorder instructions
- The hardware is allowed to reorder instructions
- Effects may not become visible to other threads immediately

All these optimizations are necessary to boost performance of sequential programs. For parallel programming, however, we need to deal with the fact that instructions are not handled sequentially.

Memory Order (2)



```
// x and y are initially zero

// Thread 1:
r1 = y.load(memory_order_relaxed);      // A
x.store(r1, memory_order_relaxed);      // B

// Thread 2:
r2 = x.load(memory_order_relaxed);      // C
y.store(42, memory_order_relaxed);      // D
```

With no memory ordering (relaxed in the example), it is allowed to produce
 $r1 == r2 == 42$:

- C and D are allowed to be reordered in thread 2's point of view
- There is no guarantee that C and D are executed atomically
- A and B cannot be reordered, but can be executed at any time when thread 2 executes C and D (no global order)

→ A possible modification order: D -> A -> B -> C

Memory Order (3)

- Synchronizing write operations is not enough. Read operations on the same variable need to be synchronized as well in order to see the changes.
- Memory orders:
 - Relaxed ordering (no order but atomicity)
 - Release-Acquire ordering
 - Release-Consume ordering
 - Sequentially-consistent ordering

Summary of Threads and Locks



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- The base of other parallel programming models
- Close to hardware
- Capable to solve almost all parallel and concurrent problems
- The only way to achieve extreme performance
- Easy to use, hard to use correctly

Functional Programming



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Like a car powered by hydrogen fuel cells

Advanced, futuristic, not yet widely used, but it is what we will rely on in 20 years.
- Functional programs model computations as the evaluation of expressions
- Functional programs have no *mutable* state

No need to worry about concurrent accesses, no mutual exclusion, no locks...
- A fundamentally different way of programming

Examples shown in “Functional Programming” are in Clojure.

A Whirlwind Tour of Clojure



- Expressions: polish notation

```
(+ 1 (* 2 3))
```

- Defining constants: **def**

```
(def meaning-of-life 42) ;; constant
(def droids [ "Huey" "Dewey" "Louie" ]) ;; vector
(def admin { :name "Paul" :age 45 }) ;; map
```

- Defining functions: **defn**

```
(defn sum [numbers] (reduce + numbers))
```

- Anonymous function: **fn/#()**

```
(fn [x] (f (g x)))
#(f (g %))
```

A Whirlwind Tour of Clojure (2)



- **get**: looks up a key in a map and returns either its value or a default

```
(def counts {"apple" 2 "orange" 1})  
  
(get counts "apple" 0)      ;; returns 2  
  
(get counts "banana" 0)     ;; returns 0 (specified default)
```

- **assoc**: takes a map, a key, and a value and returns a new map with the key mapped to the value

```
(assoc counts "banana" 1)    ;; returns {"banana" 1 "orange" 1 "apple" 2}  
  
(assoc counts "apple" 3)      ;; returns {"orange" 1 "apple" 3}
```

Effortless Parallelism



```
; Compute the sum of a sequence of numbers
; in Clojure

(defn sum [numbers]
  (reduce + numbers))

; Parallel version

(ns sum.core
  (:require [clojure.core.reducers :as r]))
(defn parallel-sum [numbers])
  (r/fold + numbers))
```

```
// Compute the sum of a sequence of numbers
// in C++

int sum(const std::vector<int> &numbers) {
    int result = 0;    // mutable state
    for(auto n : numbers)
        result += n;  // requires mutex
    return result;
}
```

This example can be easily parallelized using advanced parallel programming models for C++ but not easily with threads and locks.

Counting Words Functionally



Given a sequence of words, returns a map in which each word is associated with the number of times it appears.

```
(defn word-frequencies [words]
  (reduce
    (fn [counts word] (
      (assoc
        counts
        word
        (inc (get counts word 0)))) ; get map key default)
      {} ; empty map as identity
    words))

;; equal to the following standard library function
(frequencies [words])
;; for multiple pages that contain sequences of words
(frequencies (mapcat get-words pages))
```

Counting Words Functionally (2)



```
; ; parallel word count
(defn parallel-word-count [pages]
  (reduce
    ; ; partial returns a function with part of the arguments filled
    ; ; similar to std::bind in C++
    (partial merge-with +)
    {}
    (pmap #(frequencies (get-words %)) pages) ; ; parallel map
  ))

; ; merge-with (standard library function)
user=> (merge-with + {:x 1 :y 2} {:y 1 :z 1})
user=> {:z 1, :y 3, :x 1}
```

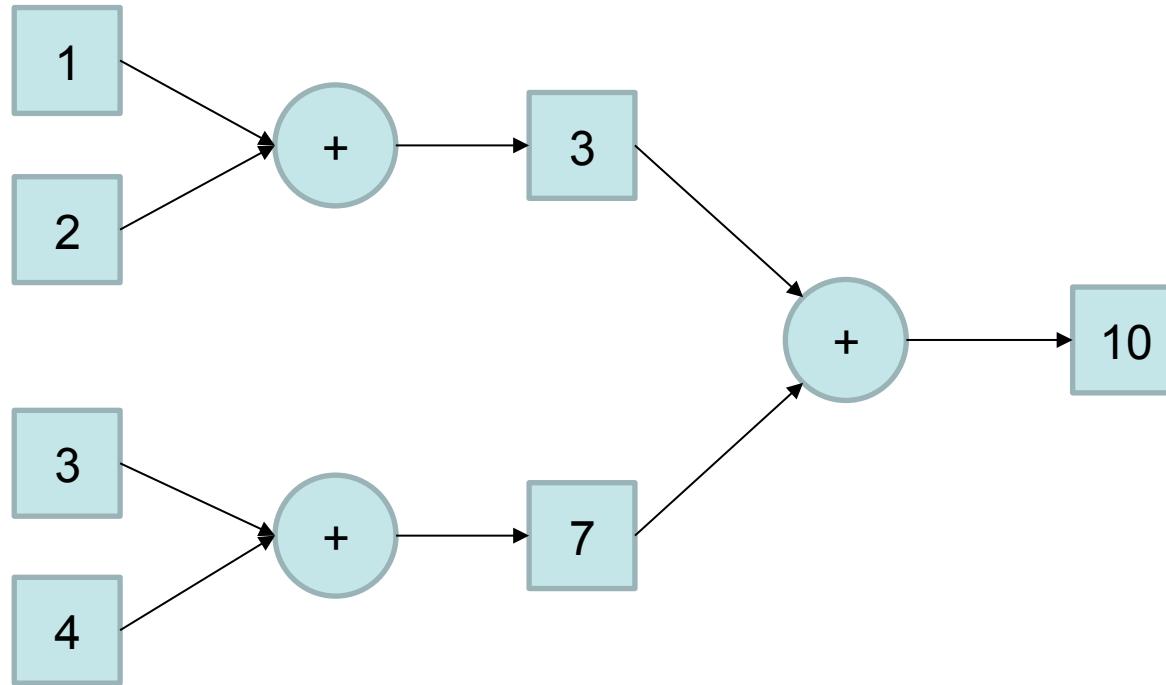
This parallel version does not give satisfying speedup. Why?

Search “clojure.core.reducers” and “fold” for detailed information about reducing.

Dataflow



(+ (+ 1 2) (+ 3 4))



Futures



A *future* takes a body of code and executes it in another thread asynchronously. Its return value is a future object.

```
(def sum (future (+ 1 2 3)))    ;; sum is the future object
(deref sum)          ;; get the result from sum (equal to @sum)
```

For previous example:

```
(let [a (future (+ 1 2))           ;; will be done in thread 1
      b (future (+ 3 4))]         ;; will be done in thread 2
  (+ @a @b))                   ;; blocks until thread 1 and 2 complete
```

Promises



A *promise* is an object that can store a value to be retrieved by a future object (possibly in another thread), offering a synchronization point.

```
; ; define a promise
(def meaning-of-life (promise))

; ; set up a future that blocks on the promise
(future (println "The meaning of life is: " @meaning-of-life))

; ; deliver a value through the promise to unblock the future
(deliver meaning-of-life 42)
```

So, Why Functional Programming?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- No mutable state

No need of mutual exclusion. The major source of bugs is eliminated!
- Functional programming is “declarative”

Evaluation order can be, and is very likely to be, different with the order of expressions in source code.
- Pure functional code is *referentially transparent**

Changing evaluation order does not change the final result.

* An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program.

And, Why Not Functional Programming?



- Generally less efficient than its imperative equivalent

Bottom line: There are problems which are $O(n)$ in the impure system which are $\Omega(n \log n)$ in the pure system. [1]
- No equivalent to some of the imperative algorithms and data structures in terms of performance
 - Union-find
 - Hash tables
 - Some graph algorithms
- Hard to analyze performance
- High learning cost

[1] Nicholas Pippenger. *Pure Versus Impure LISP*. In proceedings of POPL '96.

Actors



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Like a rental car

Quick and easy to get, and you don't bother fixing it when it breaks down.
- Has mutable state but avoids sharing it
- Targets both shared- and distributed-memory
- Strong support for fault tolerance and resilience

Examples shown in “Actors” are in Elixir.

Actors in Elixir/Erlang



- In Elixir and Erlang, an actor is called a *process*.
 - Very lightweight, even lighter than most system's threads
 - Low resource consumption
 - Low startup cost
- Elixir programs usually create thousands of processes without problems.
- An actor is like an object in an object-oriented program. Actors communicate by sending messages to each other.*

* “Sending messages” literally means sending messages. It is not just calling functions.

Actors in Action!



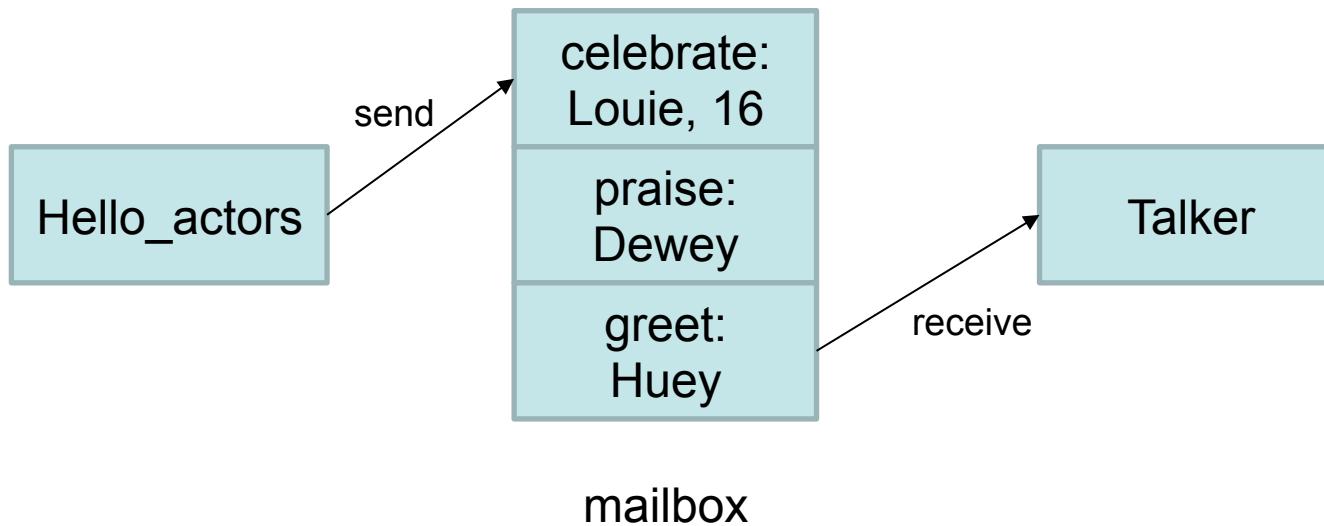
```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:shutdown}        -> exit(:normal)
    end
    loop
  end
end

# The following code is in a different file named "hello_actors.exs"
Process.flag(:trap_exit, true)
pid = spawn_link(&Talker.loop/0)
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:shutdown})
receive do {:EXIT, ^pid, reason} -> IO.puts("(#{reason})") end
```

Mailbox



In actors model, messages are sent asynchronously via a *mailbox*. Normally each actor has its own mailbox, but it is also possible that multiple actors share a mailbox.



Guarantees on Message Delivery



TECHNISCHE
UNIVERSITÄT
DARMSTADT

There are two basic guarantees about message delivery in Elixir:

- Message delivery is guaranteed if nothing breaks;
- If something does break, you'll know about it.

How can I know that something breaks?

Linking Processes



- A link between two processes can be established at any time using **Process.link**
- Links are bidirectional
 - Link process A to process B also links B to A.
- Links propagates abnormal termination, but not normal termination
 - If A terminates abnormally, B also terminates, and vice versa.
 - If A terminates normally, B continues to run, and vice versa.
- A process that can trap another's exit is called a system process
 - If A is a system process, A will be notified when B terminates either normally or abnormally. In either case, A continues to run.

Supervising a Process



A supervisor is a system process that monitors one or more processes and takes appropriate action if they fail.

```
defmodule Supervisor do
  def start do
    spawn(__MODULE__, :loop_system, [])
  end

  def loop_system do
    Process.flag(:trap_exit, true)
    loop
  end

  def loop do
    pid = spawn_link(...)
    receive do
      {:EXIT, ^pid, reason} -> loop    # restart worker thread if it fails
    end
  end
end
```

The Error-Kernel Pattern



- A software's *error kernel* is the part that must be correct if the system is to function correctly

The error kernel should be as small and as simple as possible so that there are obviously no deficiencies.
- An actor program's error kernel is its top-level supervisors

It is easy to make the error kernel of an actor program small and simple.
- Each module of a actor program has its own error kernel in turn

Leads to a hierarchy of error kernels

Fault Tolerance? Let It Crash!



- Traditional approach to achieve fault tolerance is to anticipate possible bugs, so called *defensive programming*
- Actor programs subscribe to the “let it crash” philosophy: Actors fail, and supervisors come to rescue
 - Code is simpler and easier to understand because “happy path” is clearly separated with fault-tolerance code
 - Actors are separate from one another and do not share state, so there is little danger that a failure in one actor affect another

Actors in Distributed Systems



- Whenever we create an instance of the Erlang VM, we create a node

```
user> iex -sname node1@10.99.1.1 --cookie abc
user> Node.connect(:"node2@10.99.1.2")
```

- A node can execute code on another node

```
iex(node1@10.99.1.1)1> whoami = fn() -> IO.puts(Node.self) end
iex(node1@10.99.1.1)2> Node.spawn(:"node2@10.99.1.2", whoami)
```

- An actor running on one node can send/recv messages to an actor running on another

```
# On node 2
iex(node2@10.99.1.2)1> pid = spawn(Counter, :loop, [42])
iex(node2@10.99.1.2)2> :global.register_name(:counter, pid)
# On node 1
iex(node1@10.99.1.1)3> pid = :global.whereis_name(:counter)
iex(node1@10.99.1.1)4> send(pid, {:next})
```

Summary of Actors



- Object-oriented programming to the concurrent world

Actors as objects. Similar organization of programs.
- Focus on communication among actors

Actors can have, but do not share mutable state. Within a single actor everything is sequential.
- Excellent fault tolerance
- Easy to scale well
- Susceptible to deadlock
- Unique failures (overflowing mailbox)
- Not a good choice for fine-grained parallelism

Communicating Sequential Processes (CSP)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- In real world, where you can go and how fast you can get there is primarily defined by the road network, not the car
- Similar to actors, but focus even more on communication
Channels instead of mailboxes.
- Old concept but only got popular recently because of `go`

Examples shown in “Communicating Sequential Processes” are in Clojure with `core.async` from `go`.



A channel is a thread-safe queue:

- Any task with a reference to a channel can add/remove messages to/from it
- Sender do not have to know about receivers, and vice versa

```
(def c (chan))  # create a channel named c
(thread (println "Read: " (<! ! c) "from c"))  # read from c
(>! ! c "Hello thread")                      # write to c
```

go Blocks



Like a process in Elixir, a `go` block is also very lightweight:

- Asynchronously executes the body
- Code within a go block is transformed into a state machine
 - Instead of blocking when it reads from or writes to a channel, the state machine *parks*, giving up the control of the thread it's executing on.
 - When it's next able to run, it performs a state transition and continues execution, potentially on another thread.
- Creating and executing 100,000 go blocks (performing `inc`) cost about 0.73 second (on JVM!!)

Example: Get Primes



```
(defn factor? [x y]
  (zero? (mod y x)))

(defn get-primes [limit]
  (let [primes (chan)
        ; numbers is a channel containing integer from 2 to limit
        numbers (to-chan (range 2 limit))]
    (go-loop [ch numbers]           ; initially bind ch to numbers
            (when-let [prime (<! ch)] ; reads from ch and bind to prime
              (>! primes prime)      ; write prime to output channel
              ; remove every number that can be divided by prime from ch
              (recur (remove< (partial factor? prime) ch)))
              (close! primes))
            primes))

;; >!! and <!! are the blocking versions, while >! and <! are the parking
;; versions.
```

Differences Between Actors and CSP

- In actors, mailboxes are tightly coupled with actors. In CSP, channels are first class. Channels can be independently created, written to, read from, and passed between tasks.
- Go blocks park instead of block, providing even better efficiency than actors.
- CSP lacks focus and support for distributed system and fault tolerance, although there is nothing whatsoever to stop it from supporting both.

Data Parallelism



- Data parallelism is like an eight-lane highway

Each vehicle may travel at a relatively modest speed, but the number of cars that pass a particular point is huge.
- Relevant only to a range of problems

Actually, most of the problems cannot be solved using data parallelism.
- Not a concurrency technique

Examples shown in “Data Parallelism” are in C with OpenCL.

General-Purpose Computing on GPU (GPGPU)



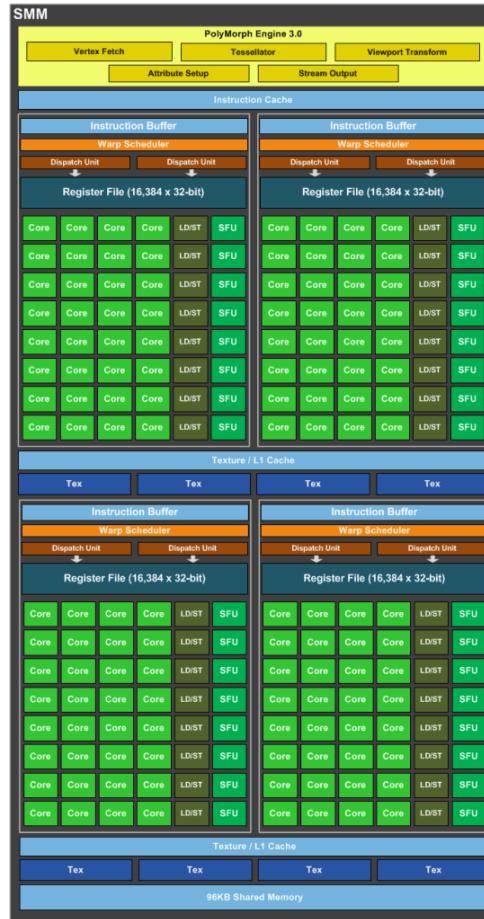
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Computer graphics is all about manipulating data -- huge amounts of data, and doing it quickly
- Although the amount of data is huge, the operations on data are relatively simple vector or matrix operations
- GPUs contain a big number of processors, rendering billions of triangles a second in parallel
- Recently, GPUs have evolved to the point that they are useful for a much wider range of application, known as general-purpose computing

Example: Maxwell (GM204)



GTX 980	
CUDA cores	2048
Base clock	1126 MHz
# SMs	16
Memory	4096 MB
Memory Interface Width	256 bit
Memory Bandwidth	224 GB/s



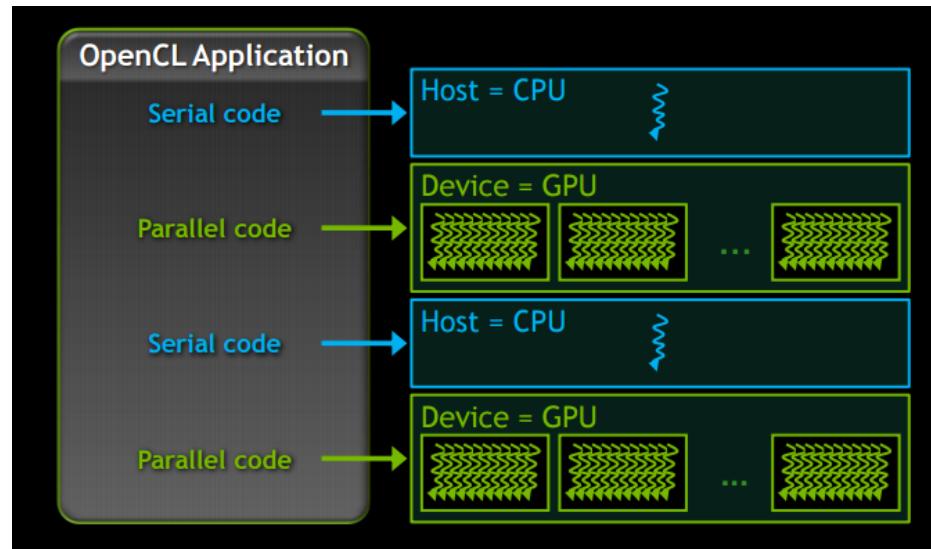
A Maxwell streaming multiprocessor. (source: nVidia)



4-way SLI. (source: Hareware.info)

OpenCL Platform Model

- Serial host code executes on CPU
- Parallel device code executes on GPU(s)
- Host code send commands to the devices
- Host code manages data movement



Anatomy of an OpenCL application. (source: nVidia)

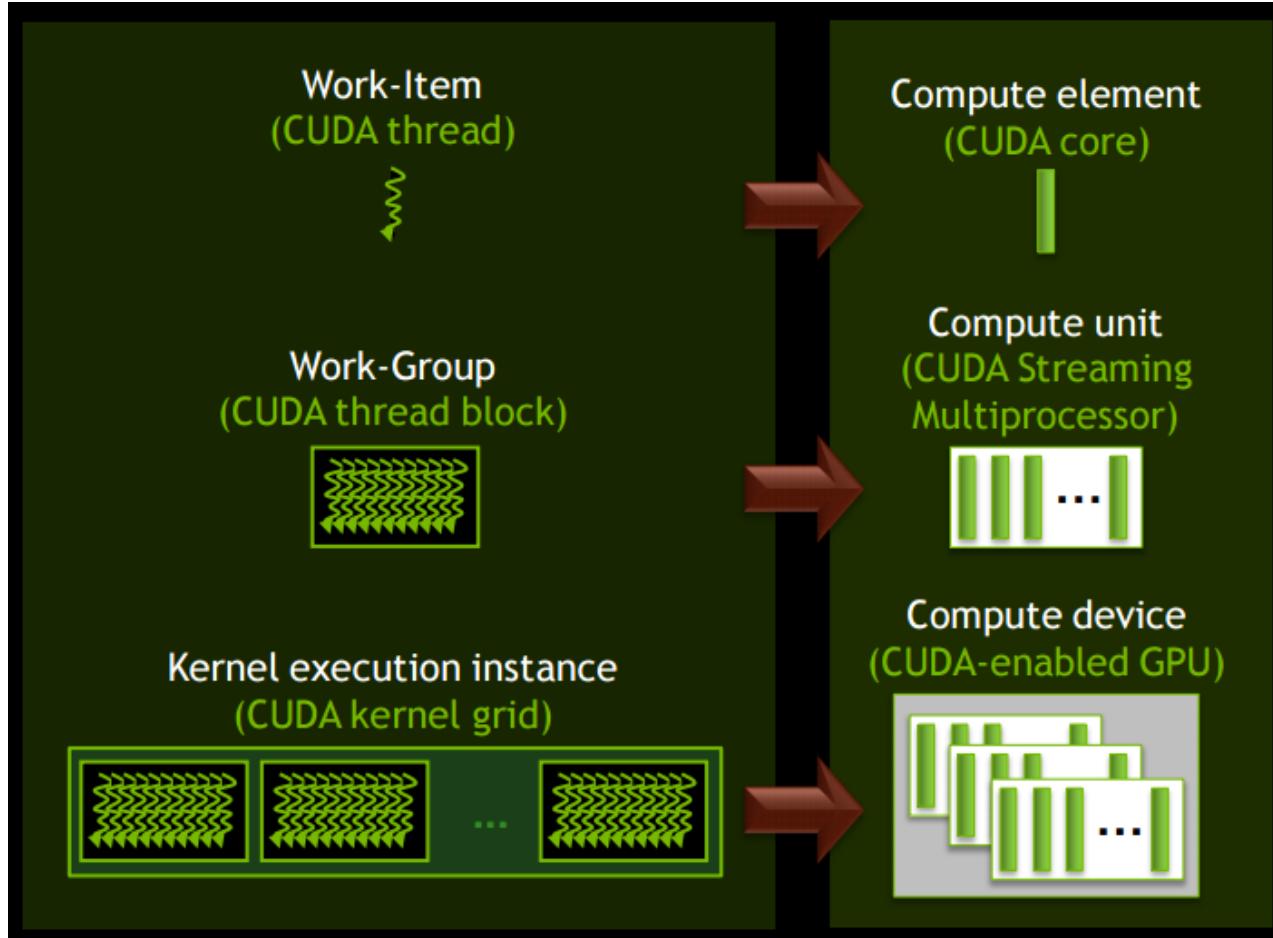
OpenCL Execution Model



- Host code submits work to the devices
- Work-item: the basic unit of work on an OpenCL device
- Work-group: a local group of work-items that executes together on the same streaming multiprocessor
- Kernel: the code for a work-item
- Program: a collection of kernels and other functions
- Context: the environment within which work-items execute
- Command queue*: a queue used by host code to submit works to a device

* Command queue is a part of the context.

OpenCL Execution Model (2)

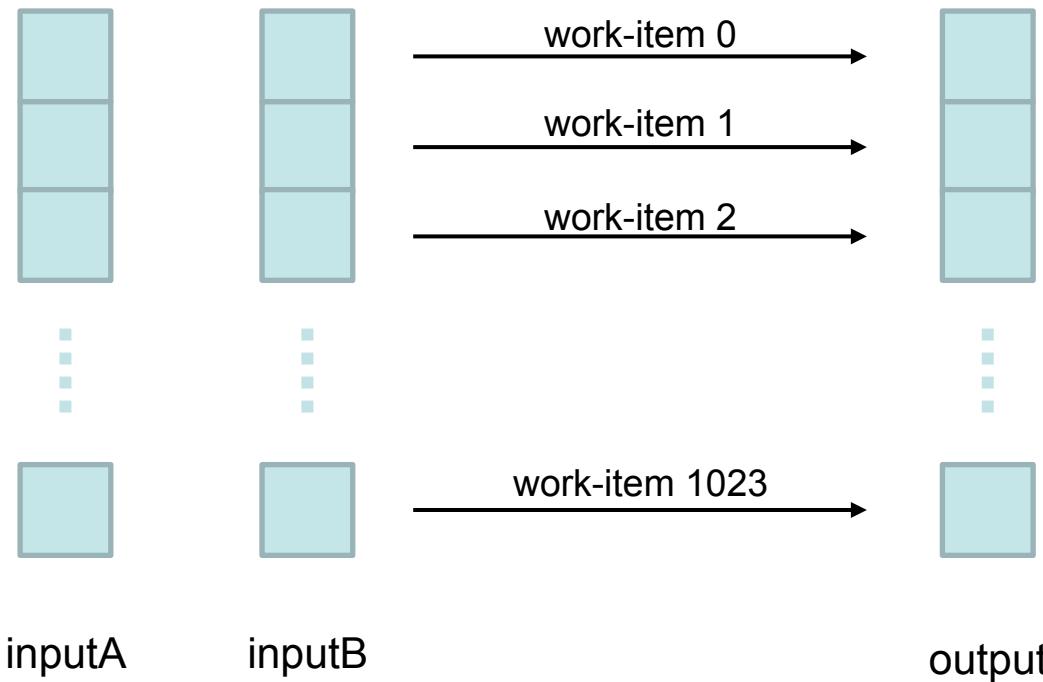


OpenCL execution model. (source: nVidia)

Work-Items



In OpenCL, work-items are very small amount of work that will be executed in parallel



Kernels



We specify how each work-item should be processed by writing a *kernel*:

```
__kernel void multiply_arrays(__global const float* inputA,
                            __global const float* inputB,
                            __global float* output) {
    int i = get_global_id(0);
    output[i] = inputA[i] * inputB[i];
}
```

Host Program



A kernel must be embedded in a *host program* that performs the following things:

- Create a context for the kernel together with a command queue
- Compile the kernel
- Create buffers for input and output data
- Enqueue a command that executes the kernel once for each work-item
- Retrieve the results
- Clean up

Example: Fragments of A Host Program



```
/* create context */
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);

/* create command queue */
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

/* compile kernel */
char* source = read_source("vector_multiply.cl");
cl_program program = clCreateProgramWithSource(context, 1, (const char**)&source,
NULL, NULL);
free(source);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "vector_multiply", NULL);
```

Example: Fragments of A Host Program (2)



```
/* create buffers (for inputA only)*/
#define NUM_ELEMENTS 1024
cl_float a[NUM_ELEMENTS];
random_fill(a, NUM_ELEMENTS);
cl_mem inputA = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float) * NUM_ELEMENTS, a, NULL);

/* execute the kernel */
clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA);
...
size_t work_units = NUM_ELEMENTS;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &work_units, NULL, 0, NULL, NULL);

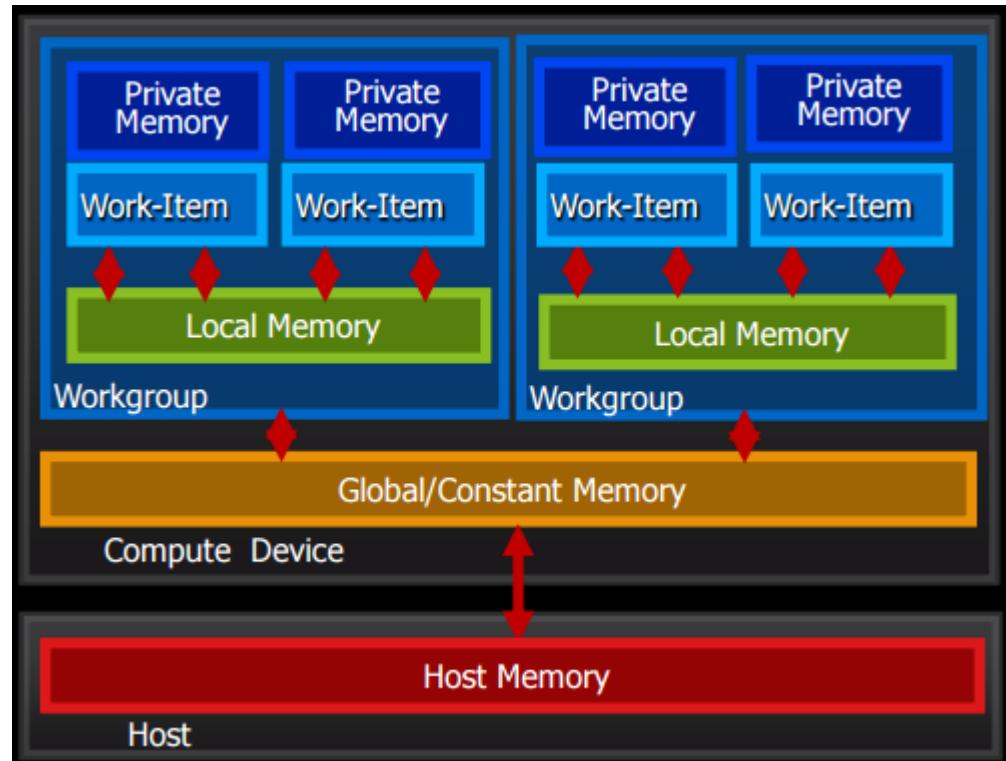
/* retrieve results */
cl_float results[NUM_ELEMENTS];
clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float) * NUM_ELEMENTS,
    results, 0, NULL, NULL);

/* clean up */
clReleaseMemObject(inputA); ...
```

OpenCL Memory Model



- Private memory for work-items
- Local memory shared within a work-group
- Global memory shared by all work-groups
- Host memory



OpenCL memory model. (source: nVidia)

Summary of Data Parallelism



- Ideal for a problem where large amounts of numerical data needs to be processes

Extremely useful in scientific computing and simulation, like fluid dynamics, finite element analysis, n-body simulation, neural networks, ...
- Utilizing GPUs yields better efficiency in terms of GFLOPS/watt

That is why modern supercomputers are all equipped with GPUs.
- Applying data parallelism is pretty much about number-crunching

It is not common in general-purpose applications.
- Hard to write high-performance cross-platform code since optimization often depends on underlying GPU architectures, which are significantly different from one to another

Other Parallel Programming Models



- Task parallelism

Usually classified as SPMD. A general solution to nearly all kinds of problems.

- Map-Reduce

Deals with big data problems where solutions can usually be divided into a pure data parallelism phase and a reduce phase.

- The Lambda Architecture

Evolves from Map-Reduce, provides better performance, robustness, and scalability.

Wrap-Up



- Threads and locks model serves as the foundation of other parallel programming models
- Mutable state is evil, and immutable is the future
- Message passing scales well, but has its unique failures
- Data parallelism is common in scientific computing, and the potential of GPUs is being realized
- Parallel models still evolve, and managing the essentials helps you learn new models fast



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

PARALLEL DESIGN PATTERNS

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

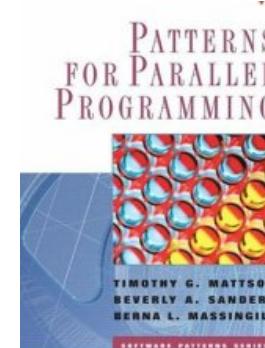
- Software Design Patterns
- A pattern language for parallel programming
- Finding parallelism
- Algorithm structure patterns
- Supporting structures patterns
- Implementation mechanism

Chapter based on the book

Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill:

Patterns for Parallel Programming

Addison-Wesley, 2005



Software Design Patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Many software from different fields have similar problems when designing them
- Software engineers came up with Software Design Patterns:
 - They provide solutions to recurring design problems
 - They provide template or description to solve the problem
 - They enhance software's:
 - quality
 - re-usability
 - maintainability

Software Design Patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Format of pattern description
 - Pattern name
 - Description of the context
 - Forces (goals and constraints)
 - Solution
- Brief history
 - First proposed by [Alexander et al., 1977] for city planning
 - Introduced to software engineering by [Beck and Cunningham, 1987]
 - Became prominent in the area of OO-programming through [Gamma et al., 1995]

Pattern language



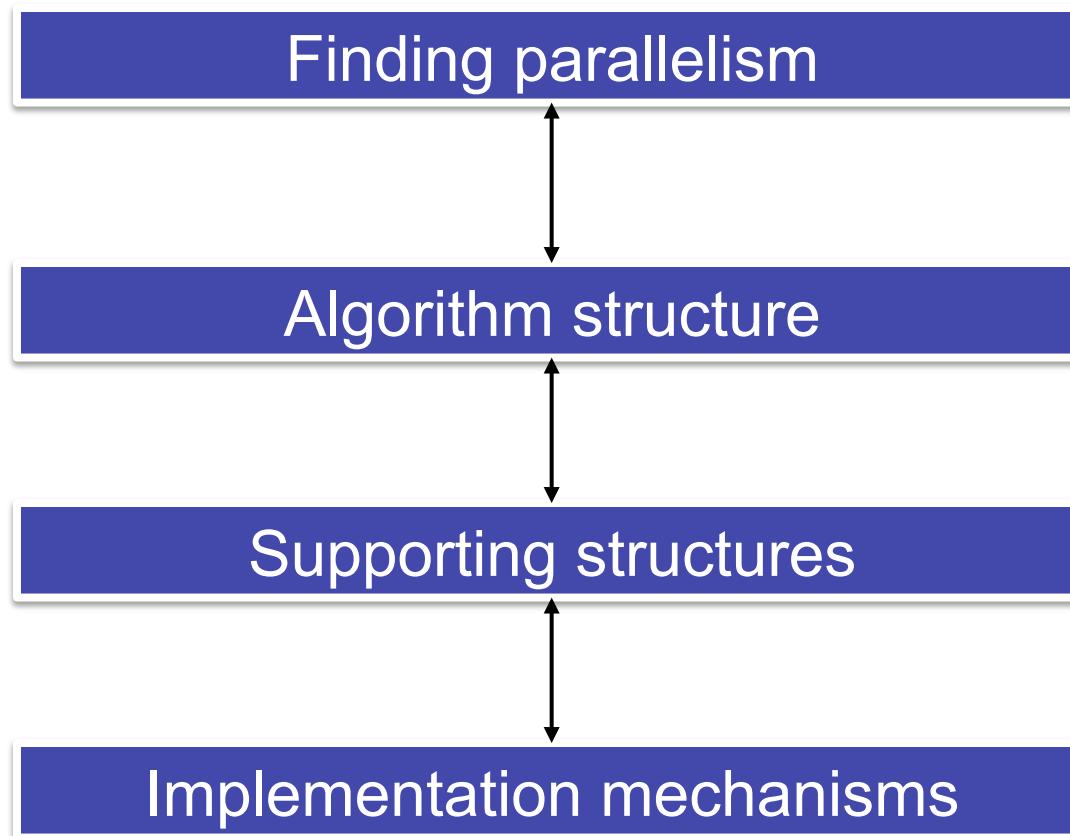
- Alexander also introduced pattern language
- Organizes patterns into a structure
 - Guides the user through the collection of patterns such that complex systems can be designed using the patterns
 - At each decision point, the designer selects an appropriate pattern
 - Each pattern leads to another pattern, resulting in a final design in terms of a web of patterns
- Embodies design methodology
- Provides domain-specific advice to the designer

Pattern language for parallel programming



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Four design spaces



Design spaces



TECHNISCHE
UNIVERSITÄT
DARMSTADT

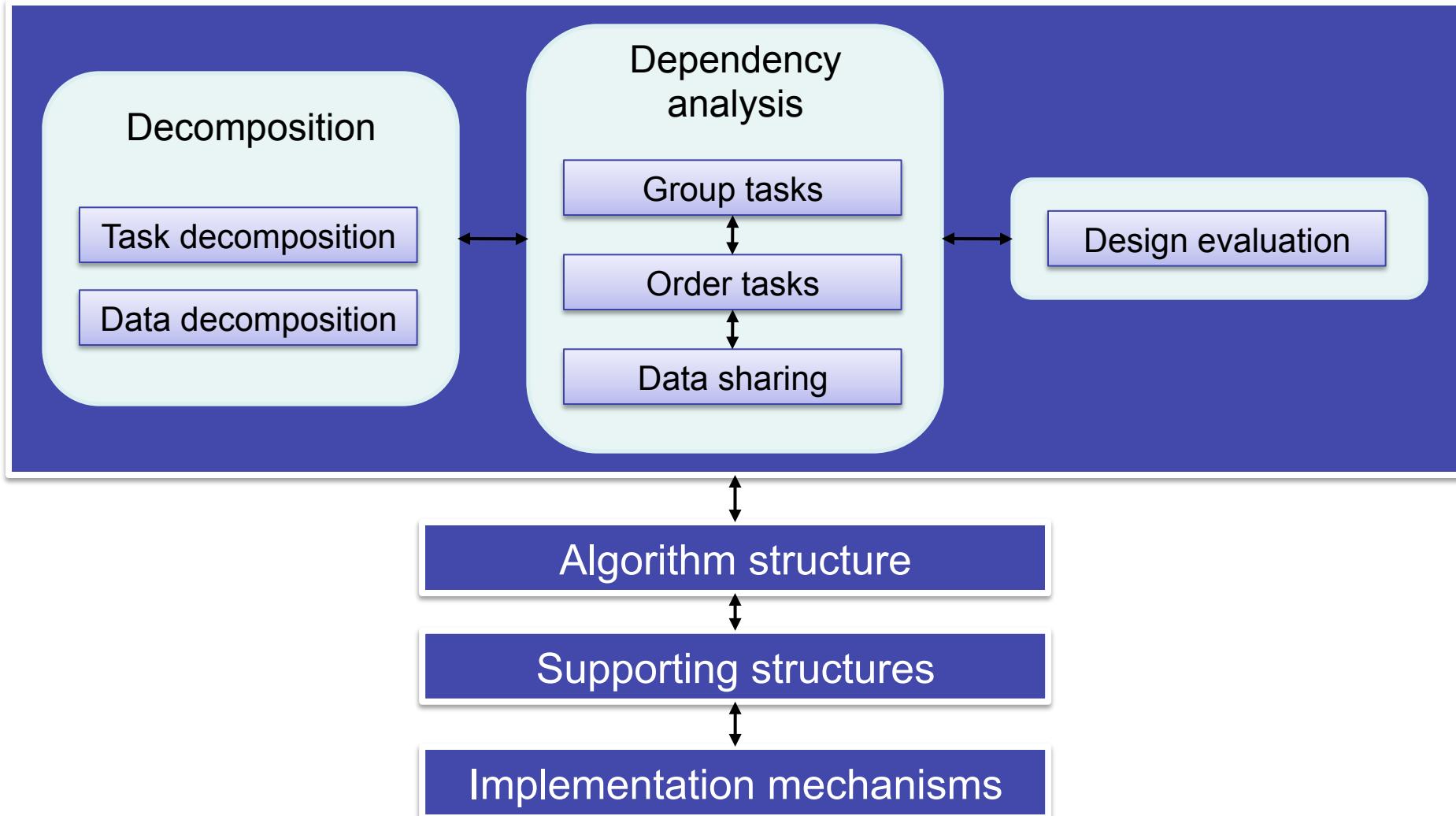
- Finding parallelism
 - High-level algorithmic issues
 - Parallelism exposed by the problem
- Algorithm structure
 - Structuring the algorithm to take advantage of parallelism
 - Overall strategies for exploiting parallelism

Design spaces



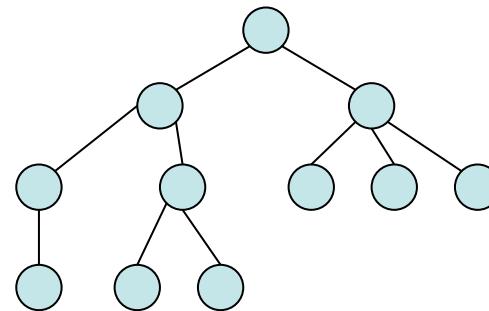
- Supporting structures
 - Program structuring approaches
 - Commonly used shared data structures
- Implementation mechanisms
 - Mapping of patterns into particular programming environment
 - E.g., common mechanism for process/thread management and interaction
 - Items in this design space not represented as patterns because they map directly onto programming constructs

Finding parallelism



Finding parallelism

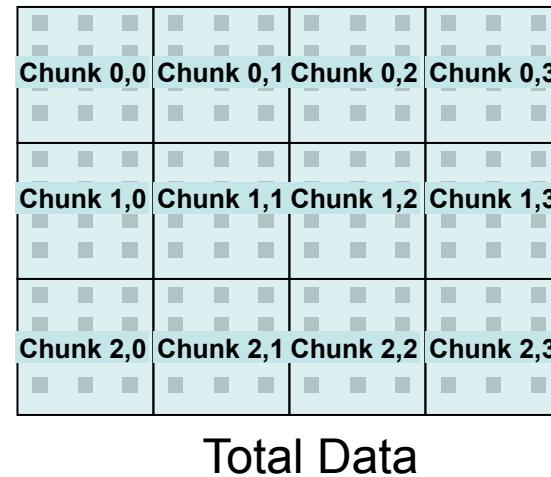
- Decomposition patterns are used to decompose the problem into pieces that can execute concurrently
 - **Task decomposition** dimension views problem as a stream of instructions that can be broken down into tasks to be executed simultaneously



Finding parallelism



- Data decomposition dimension views problem as an amount of data that can be broken down into chunks to be independently operated upon



Examples: Medical imaging



- Positron Emission Tomography (PET)
 - Images formed from distribution of emitted radiation are of low resolution – in part due to scattering of radiation
 - Different pathways through the body attenuate radiation differently
- Solution
 - Simulate trajectories of gamma rays through the body using Monte Carlo approach



Examples: Linear algebra

- Matrix transformation / multiplication

$$C = T * A \quad C_{i,j} = \sum_{k=0}^{N-1} T_{i,k} * A_{k,j}$$

- Overall complexity

$$O(n^3)$$



Task decomposition pattern

- Problem
 - How can a problem be decomposed into tasks that can be executed concurrently?
- Context
 - Good starting point if problem can naturally divided into tasks
- Forces
 - Flexibility
 - Efficiency
 - Simplicity

Task decomposition pattern



- Solution
 - Look at the problem as collection of distinct tasks (i.e., actions that are carried out to solve the problem)
 - Are there enough to keep all PEs busy?
 - Are they distinct and relatively independent?
 - Tasks can be found in many places
 - Functional decomposition
 - Loop iterations
 - Updates on different data chunks (→ data decomposition)

Task decomposition pattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Task granularity
 - Should be flexible to be able to exploit hardware parallelism
 - Avoid overhead of task creation and dependency management
 - Keep code inside tasks maintainable (e.g., reuse seq. code)

Task decomposition - examples



- Medical imaging
 - Associate task with each trajectory
 - Natural because trajectories are completely independent
 - Requires access to full body model
 - Easy with shared memory
 - Difficult with distributed memory - requires replication (communication and space)
- Matrix multiplication
 - Each element of the product matrix is a task
 - Inefficient because of low memory performance
 - Group tasks to blocks (equivalent to data decomposition)

Data decomposition pattern



- Problem
 - How can a problem's data be decomposed into units that can be operated on relatively independently?
- Context
 - Data decomposition good starting point
 - If computationally intensive part of the problem is organized around the manipulation of large data structure
 - If similar operations are can be applied independently to different parts of the data structure

Data decomposition pattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Forces
 - Flexibility
 - Efficiency
 - Simplicity
- Solution
 - Array-based computations: update different segments of an array in parallel
 - Recursive data structures: e.g., concurrent update of sub-trees

Data decomposition pattern



- Granularity of data chunks
 - Configurable for flexibility
 - Not too small to reduce overhead of managing dependencies
 - Not too large to exploit available parallelism
 - Not too irregular to achieve good load balance

Data decomposition - examples

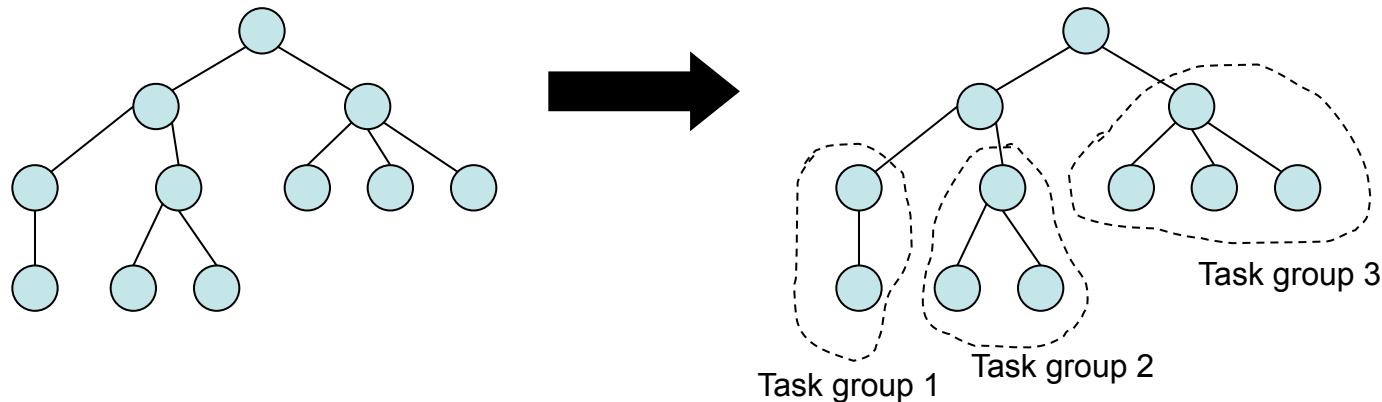


- Medical imaging
 - Body model is large central data structure
 - Different segments are mapped onto different PEs
 - Ray trajectories define tasks
 - May need to be passed between segments
- Matrix multiplication
 - Decomposition into rows
 - Decomposition into blocks (cache friendly)
 - Ratio of FP operations $O(N^3)$ to memory references $O(N^2)$

Dependency analysis pattern

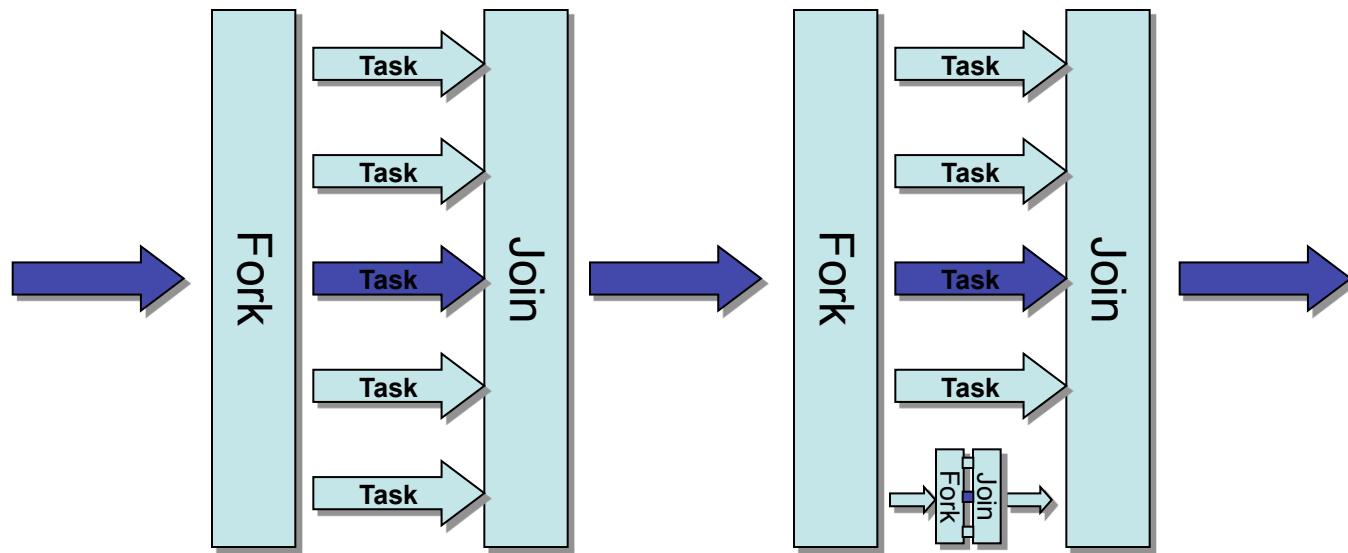


- Group task pattern
 - Group tasks to simplify the management of dependencies
 - Tasks form a dependence graph
 - Identify groups that share the same dependency constraints



Order task pattern

- Order tasks to satisfy constraints among tasks
 - Temporal dependencies
 - Availability of shared data



Data sharing pattern



- How is data shared among tasks given data and task decomposition
 - Correct management of access to shared data like:
 - Global variables
 - Access to a task's local data
 - Solution
 - Identify data shared between tasks
 - **Read-only:** only read but not written
 - **Effectively-local:** data is mainly local to task and maybe recombined in the end
 - **Read-write:** data is read and written by multiple threads
 - **Accumulate:** data is used to accumulate a result
 - **Multiple-read/single-write:** data is read by multiple tasks but written by only one

Design evaluation pattern



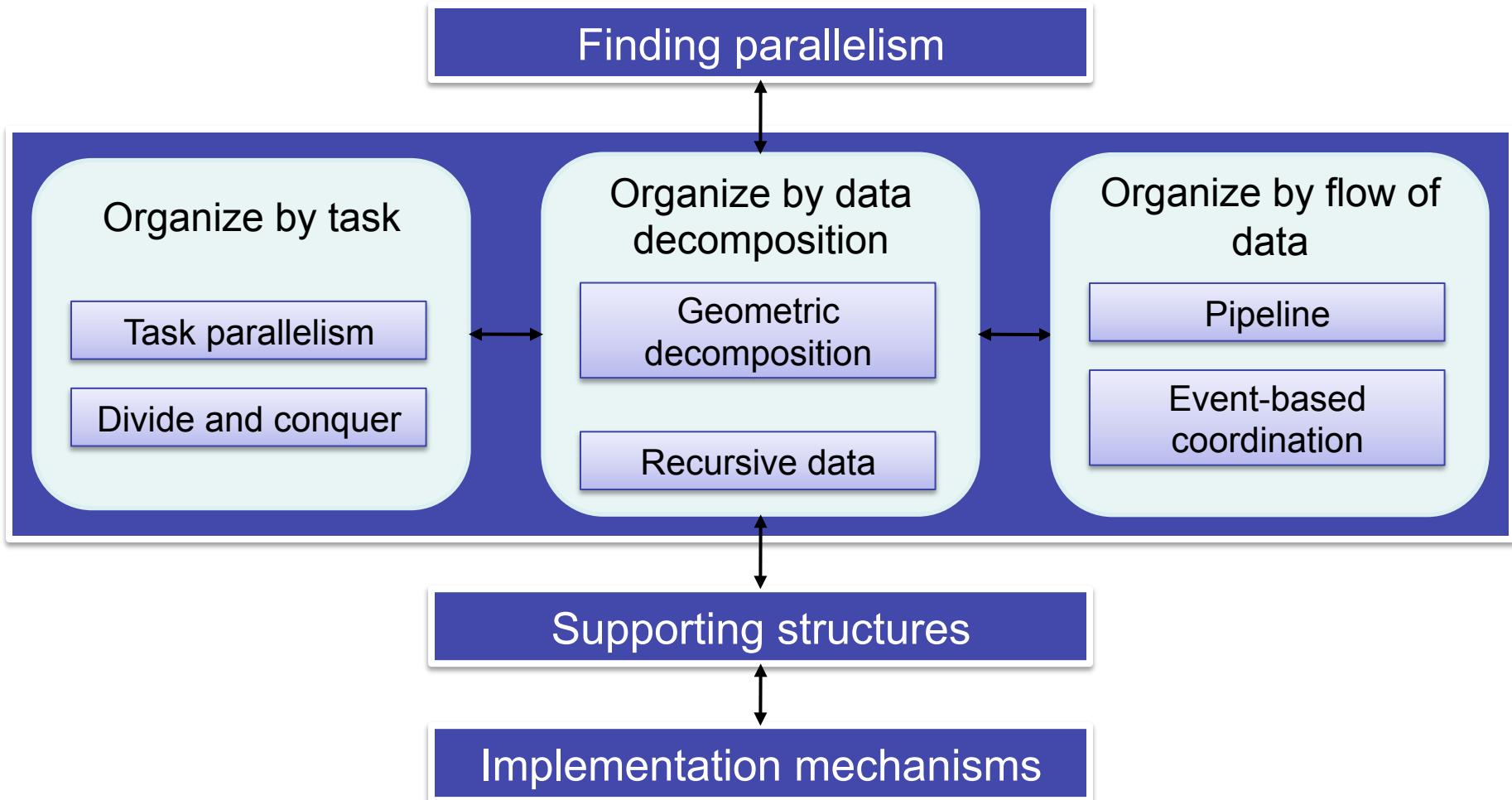
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Is decomposition and dependency analysis good enough
- Design suitable for target platform?
- #PEs, memory organization
- How about flexibility, efficiency, simplicity?

Output from finding parallelism

- A task decomposition that identifies tasks that can be executed concurrently
- A data decomposition that identifies data local to each task
- A way of grouping tasks and ordering the groups to satisfy temporal constraints
- An analysis of dependencies among tasks

Algorithm structure design space



Forces



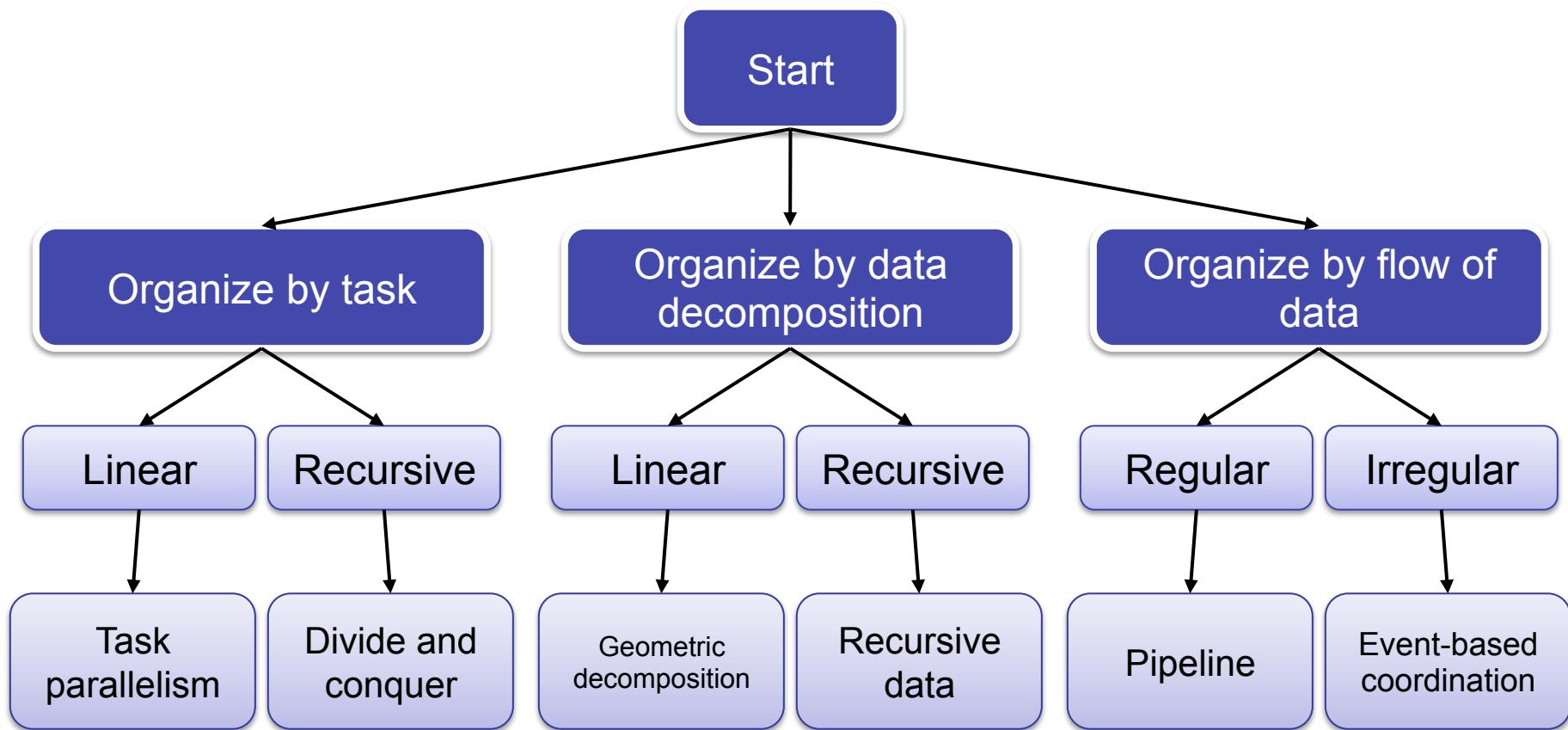
- **Efficiency** – algorithm should utilize computer resources
- **Simplicity** – algorithm should be easy to develop, debug, verify, and modify
- **Portability** – protect investment in view of evolving platforms
- **Scalability** – algorithm should run on wide range of processor configurations

Choosing an algorithm structure pattern



- Target platform
 - How many UEs can be supported (magnitude)?
 - How expensive is it to share information?
 - Shared memory and memory bandwidth
 - Distributed memory and network
 - Which programming environment?
- Major organizing principle
 - Task decomposition
 - Data decomposition
 - Flow of data

Decision tree





Task parallelism pattern

- Categorize dependencies
- Schedule
- Program structure
- Common idioms

Categorize dependencies



- Removable dependencies

```
int ii = 0, jj = 0;

for (int i = 0; i < N; i++) {
    ii = ii + 1;
    d[ii] = big_time-consuming_work(ii);
    jj = jj + i;
    a[jj] = other_big_calc(jj);
}
```

```
for (int i = 0; i < N; i++) {
    d[i] = big_time-consuming_work(i);
    a[(i*i+i)/2] = other_big_calc((i*i+i)/2);
}
```

Categorize dependencies (2)

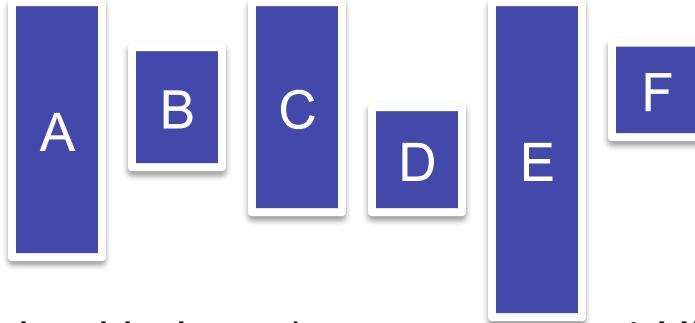


- Separable dependencies
 - Accumulation into shared data structure can be separated from task
 - Often accumulation is reduction operation
 - Create copy of data structure on each UE
 - Each copy initialized to identity element of binary operation
 - Each task accumulates into local data structure
 - When all tasks are complete, local data structures are combined

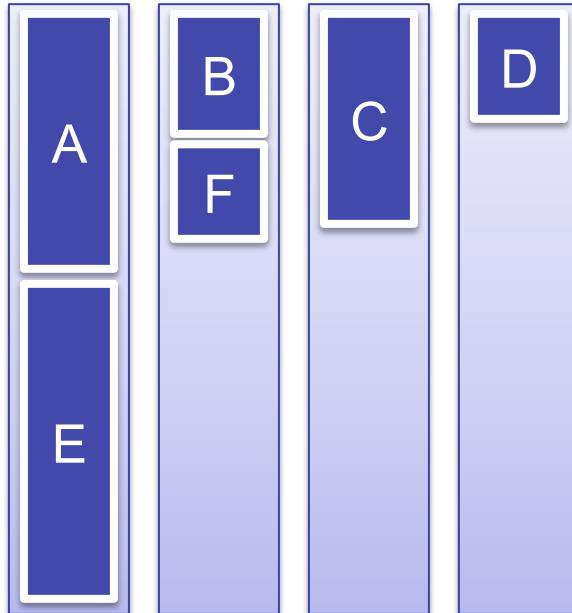
```
for (int i = 0; i < N; i++) {  
    sum = sum + f(i)  
}
```

- Other dependencies

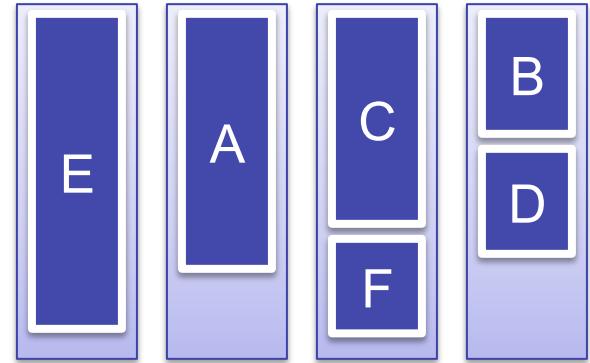
Schedule



4 UEs (poor load balance)



4 UEs (good load balance)



Schedule (2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Static schedule
 - Tasks divided into blocks
 - Blocks assigned to UEs
 - Assumption: effort for each block about the same

Schedule (2)



- Dynamic schedule
 - Used when
 - Effort associated with each task is not known and varies widely
 - Capability of each UE varies widely
 - Often implemented as **task queue** or **task pool**
 - Sometimes via **work stealing**
 - Each UE maintains its own queue
 - If queue empty, UE “steals” work from other UE
 - Often too complex

Program structure



- Loop-based
 - Parallelize the iterations of a loop
- Task queue
 - SPMD = Single Program Multiple Data
 - Master / worker
- Termination condition
 - Examples
 - All tasks finished
 - Acceptable solution has been found by one task
 - Make sure that condition is met and that program ends if condition is met

Common idioms



- Embarrassingly parallel
 - E.g., rendering frames, statistical sampling
 - No dependencies to manage
 - Focus on efficient scheduling
- Replicated data or reduction problem
 - Dependencies can be separated from tasks
 - Overall solution consists of three phases
 - Replicate data into local variables
 - Solve now-independent tasks
 - Recombine local results into global result

Mandelbrot set

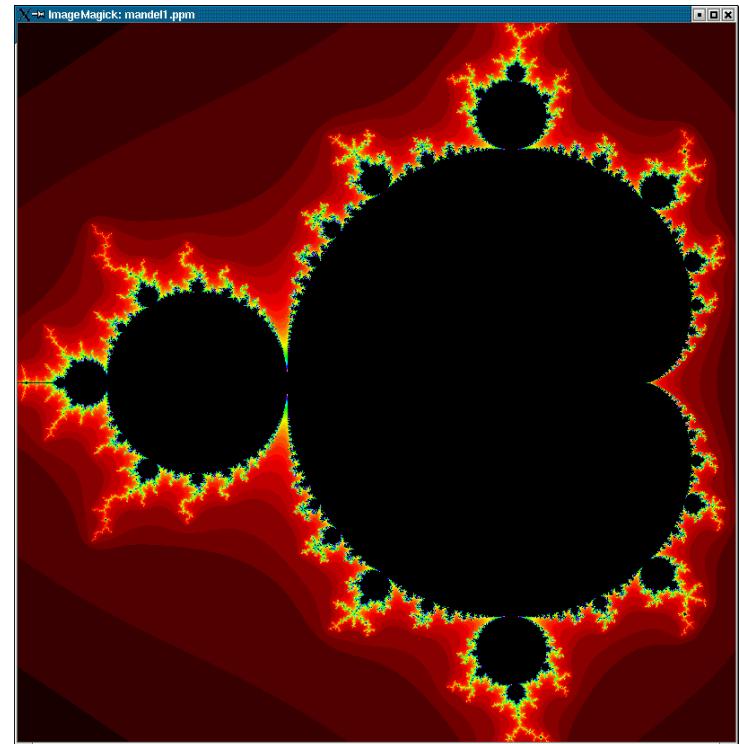


- Each pixel is colored according to the behavior of the quadratic recurrence relation

$$Z_{n+1} = Z_n^2 + C$$

$$Z_0 = C$$

- Pixel is black if recurrence relation converges to a stable value or is colored depending on how rapidly the relation diverges

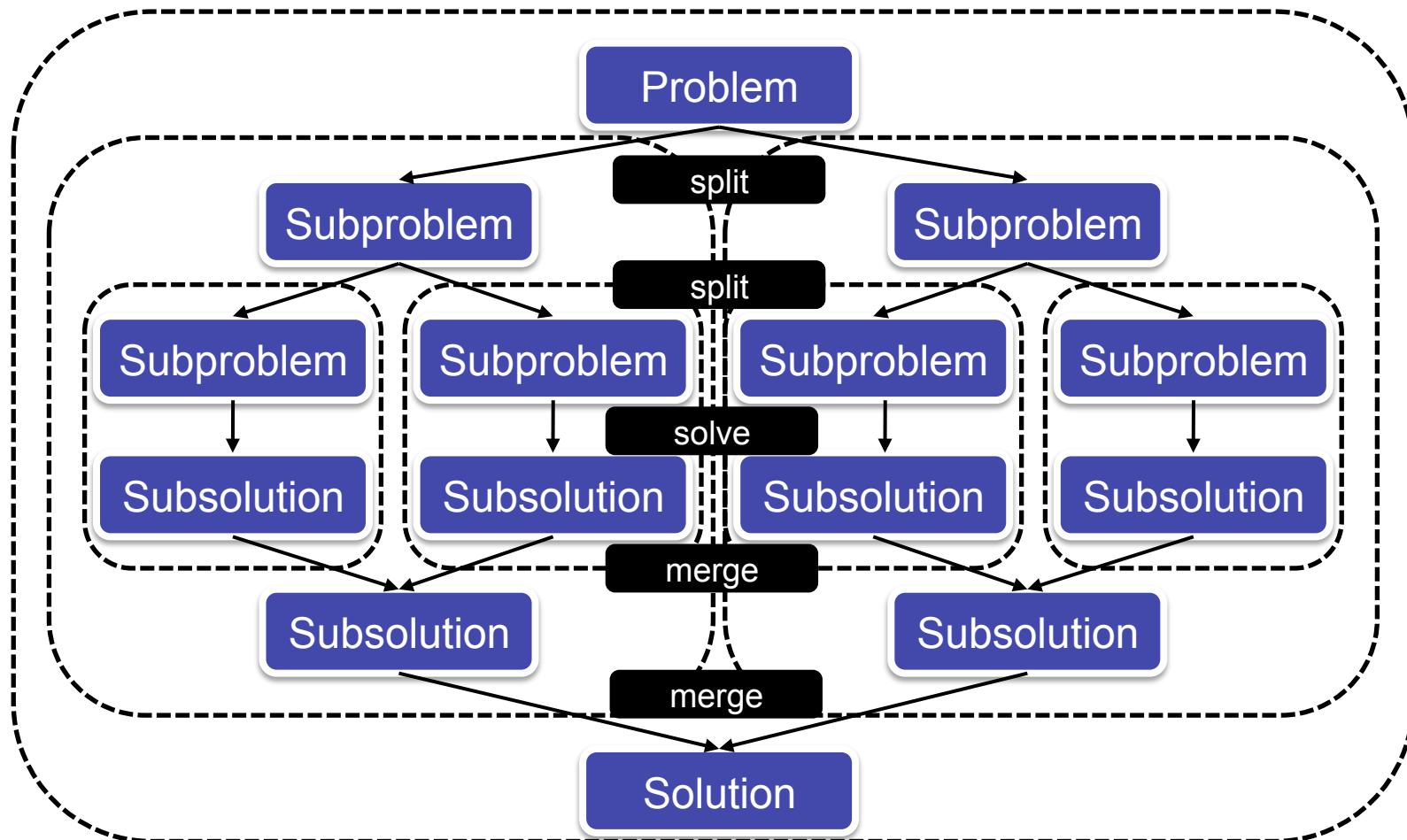


Mandelbrot set



- Task granularity
 - Single pixel – effort depends on divergence behavior
 - Row / block – per-block effort can still be comparable
- Schedule
 - Static schedule with many more tasks than UEs can still have good load balance
- Program structure
 - Loop parallelism on shared-memory machine
 - SPMD with distributed memory machine
- Heterogeneous cluster might indicate dynamic schedule

Divide and conquer pattern



Divide and conquer pattern

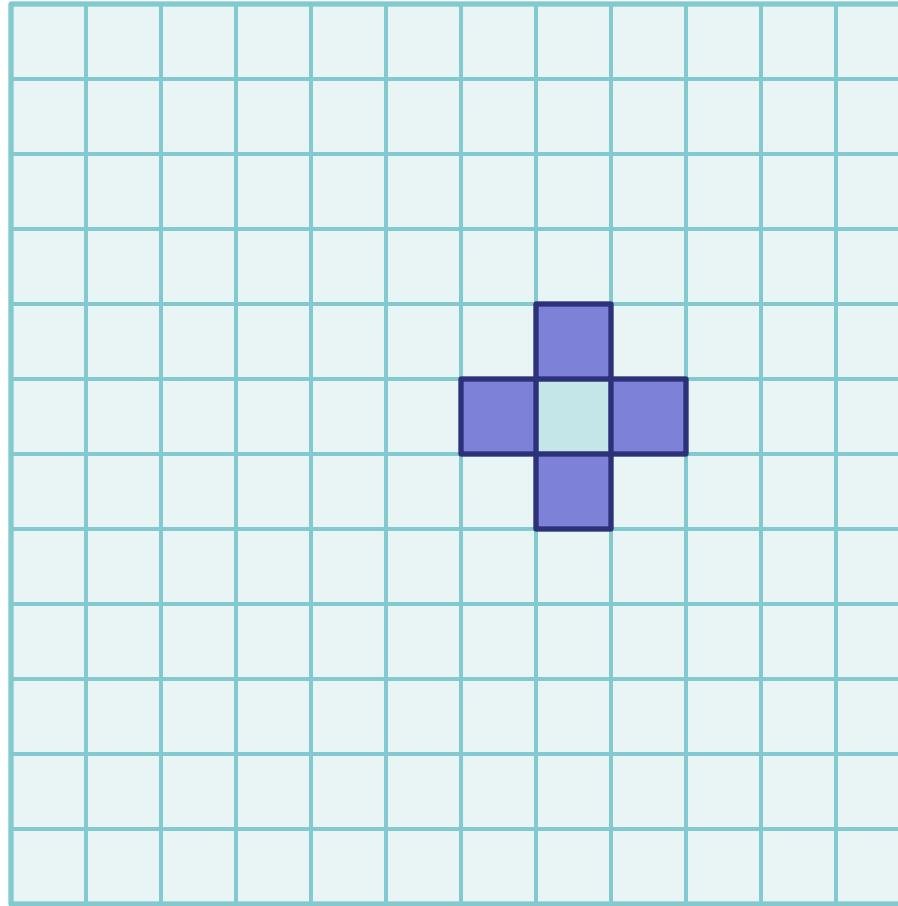


- Commonly used in recursive programs
 - Merge sort, Fibonacci number etc.
- **Mapping tasks to UEs and PEs** is simple and straight forward
- **Communication costs:** data must be moved to PE to solve the subproblem
- **Dealing with Dependencies:** commonly subproblems should be independently solvable. Access to common data can be handled using data pattern
- **Other Optimizations:** parallelization of split and merge processes

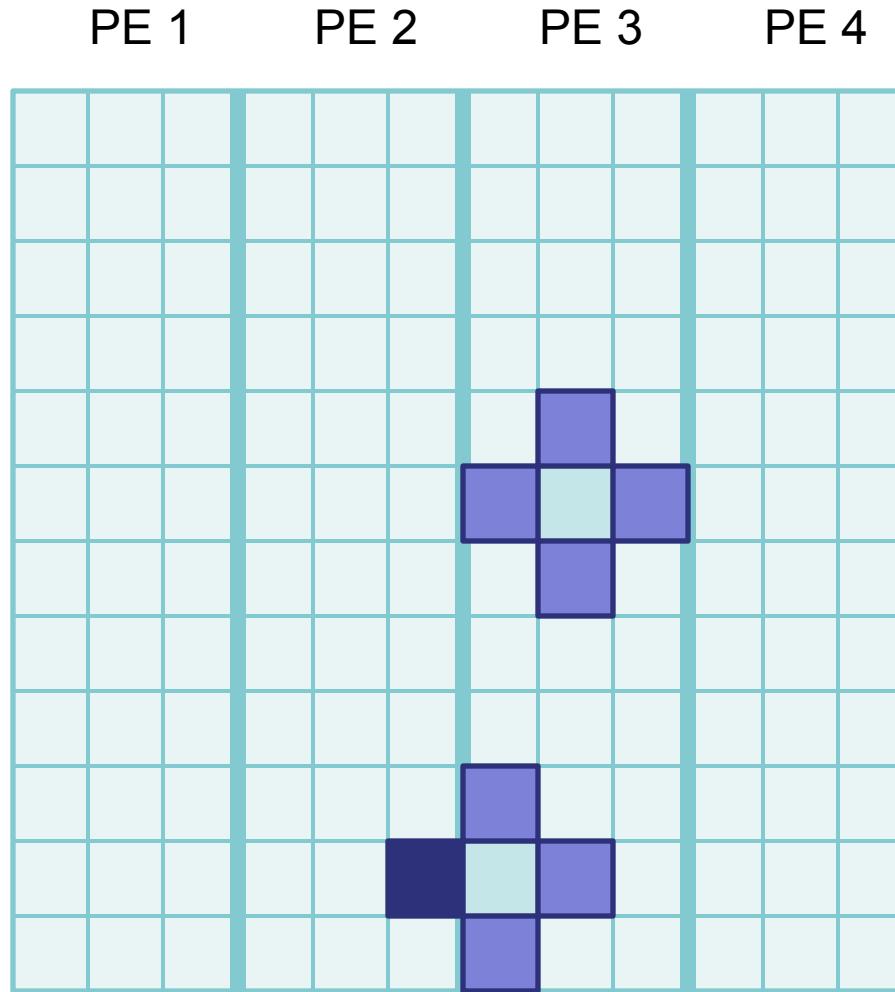
Geometric decomposition pattern



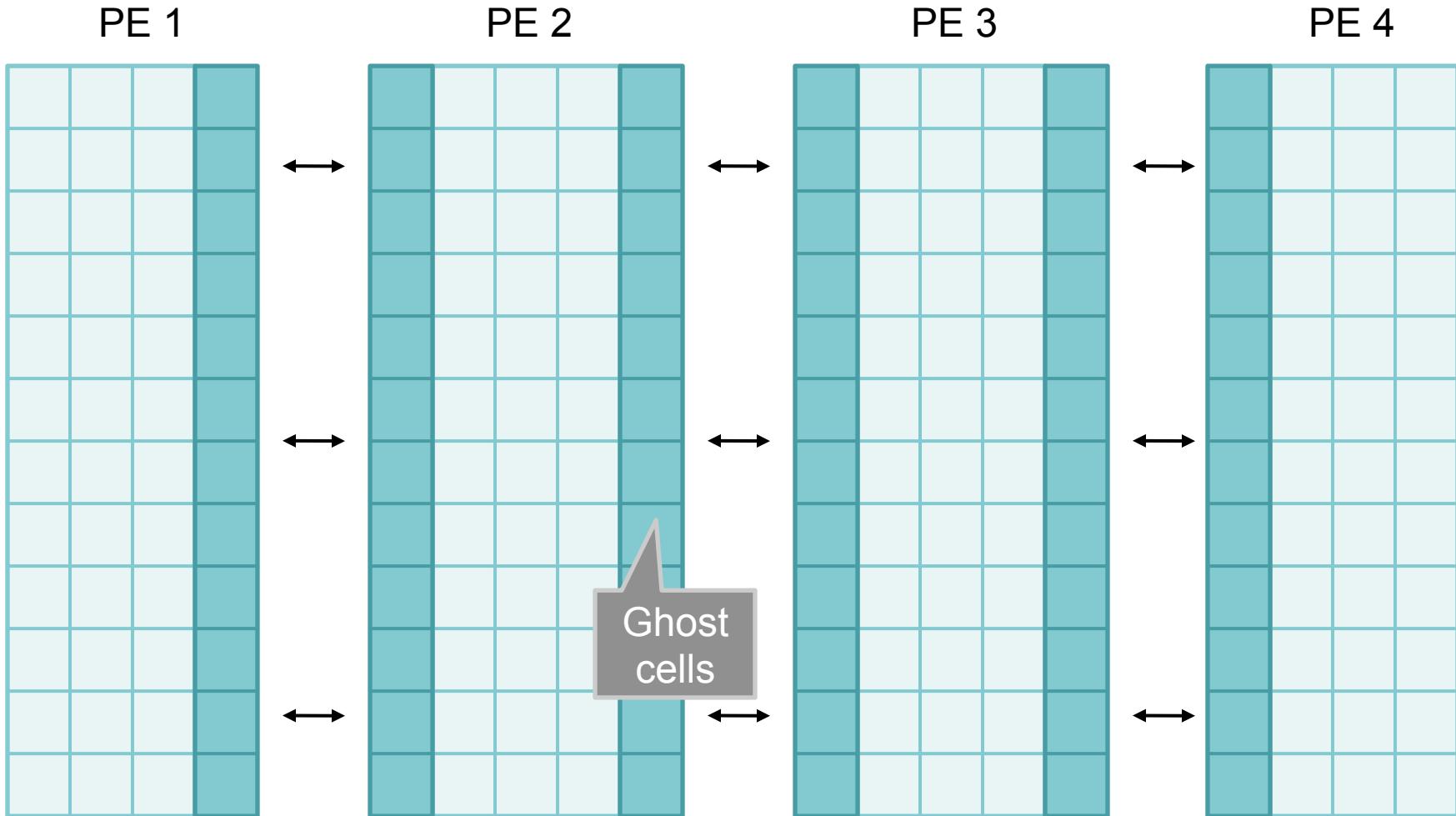
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Geometric decomposition pattern



Geometric decomposition pattern



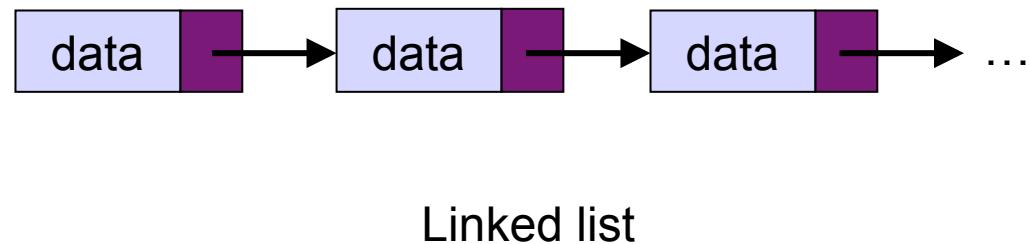
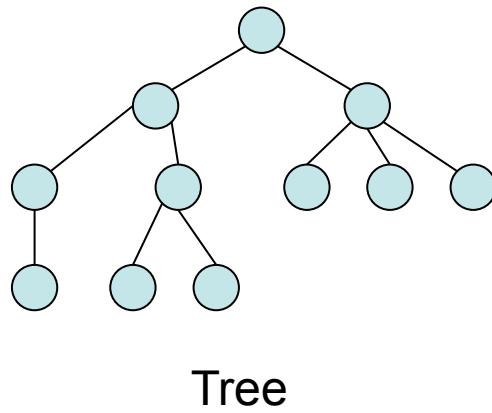
Geometric decomposition pattern



- Commonly used in:
 - solution of differential equations
 - computational linear algebra
- The pattern reduces to **embarrassingly parallel** if update to chunk is not dependent on other chunks
 - This is **task parallelism!**
- If data structure is recursive in nature, then use **divide and conquer** or **recursive data** patterns

Recursive data pattern

- Applicable when the data structure is recursive in nature
 - For example



Recursive data pattern

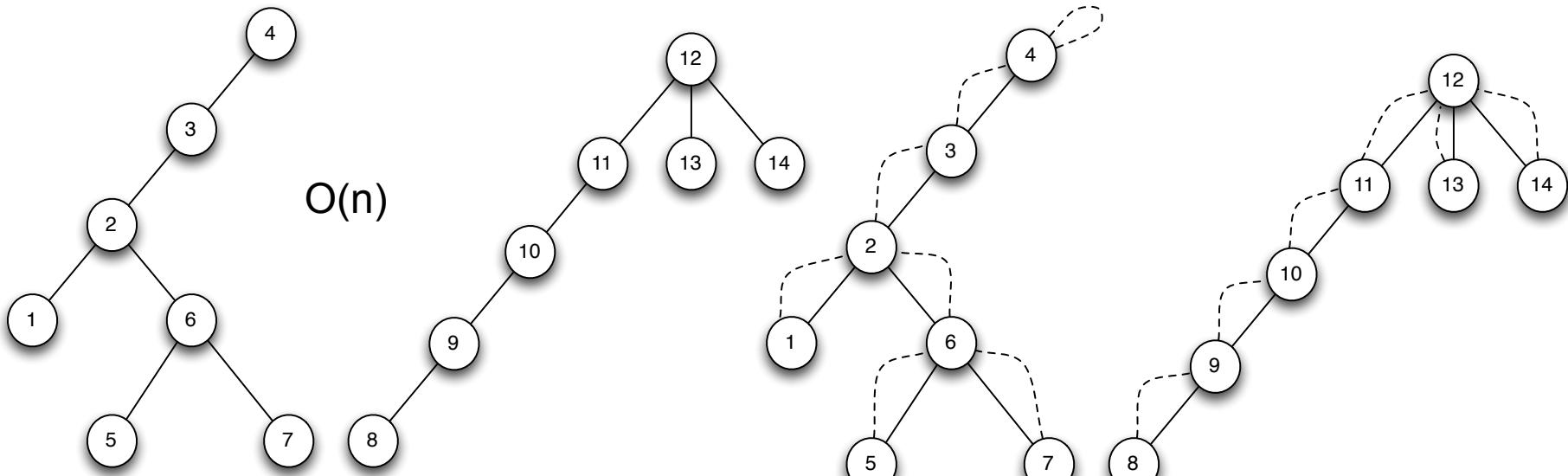


- Some recursive data structures are naturally compatible to use with divide & conquer pattern
- In some structures, apparently the only way is to sequentially go through all elements and compute results
 - Sometimes it is possible to reshape the operations to achieve concurrency

Recursive data pattern



- Example: Finding roots in a forest



Forest of trees

Solid lines represent original
parent-child relationships

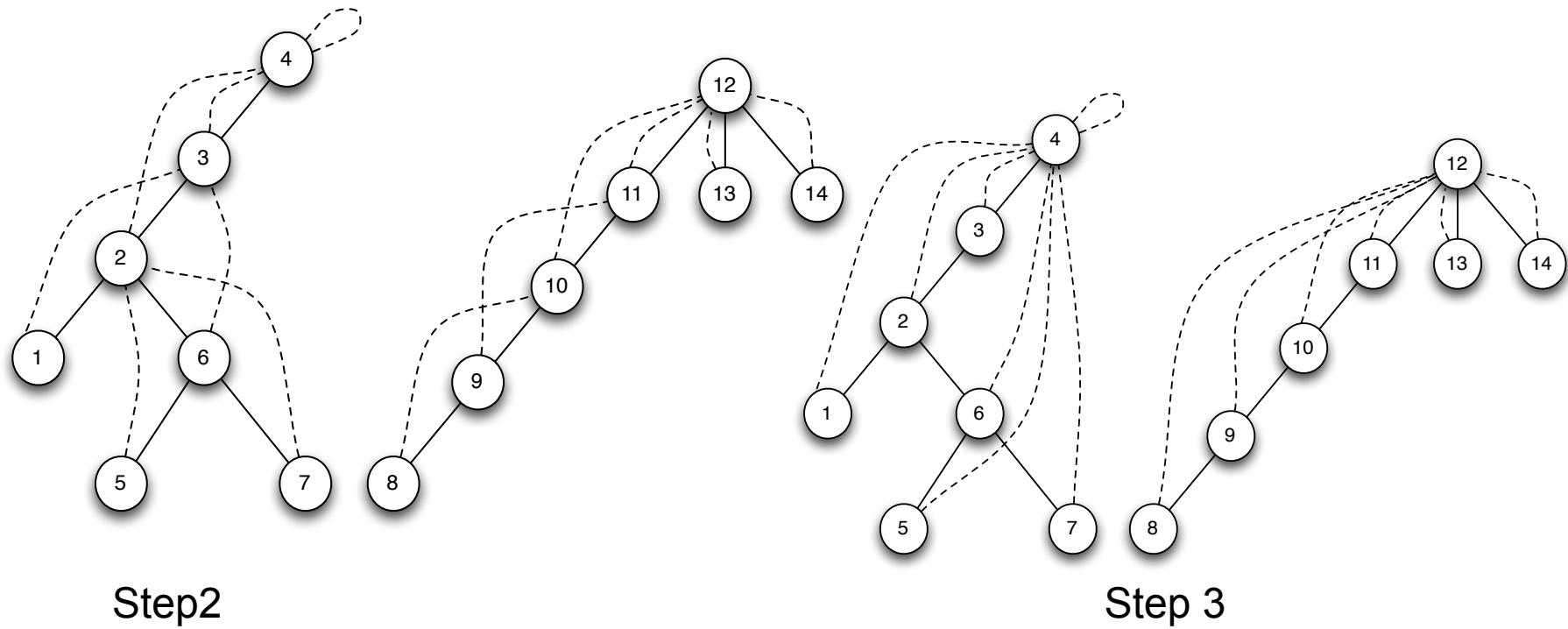
Step 1

dashed lines point from
nodes to their successors

Recursive data pattern



- Example: Finding roots in a forest



- All nodes processed concurrently
- Complexity $O(\log(n))$

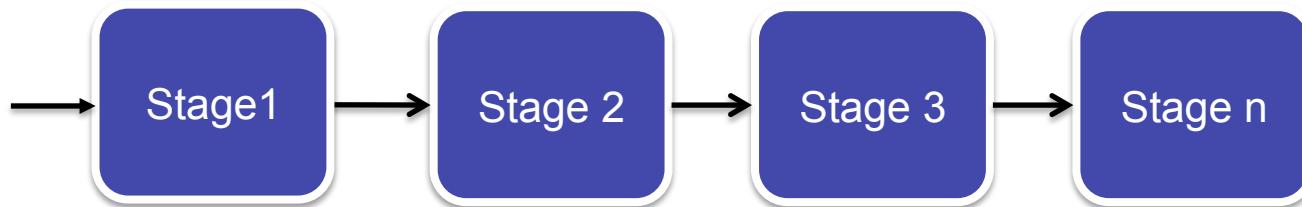
Recursive data pattern



- Restructuring of operations is the most difficult task
- Recursive data structure is decomposed into individual elements
 - Each assigned to separate UE
- Typical solution is a loop with its each iteration performing an operation on the element of structure in parallel
- Synchronization maybe needed in some cases

Pipeline pattern

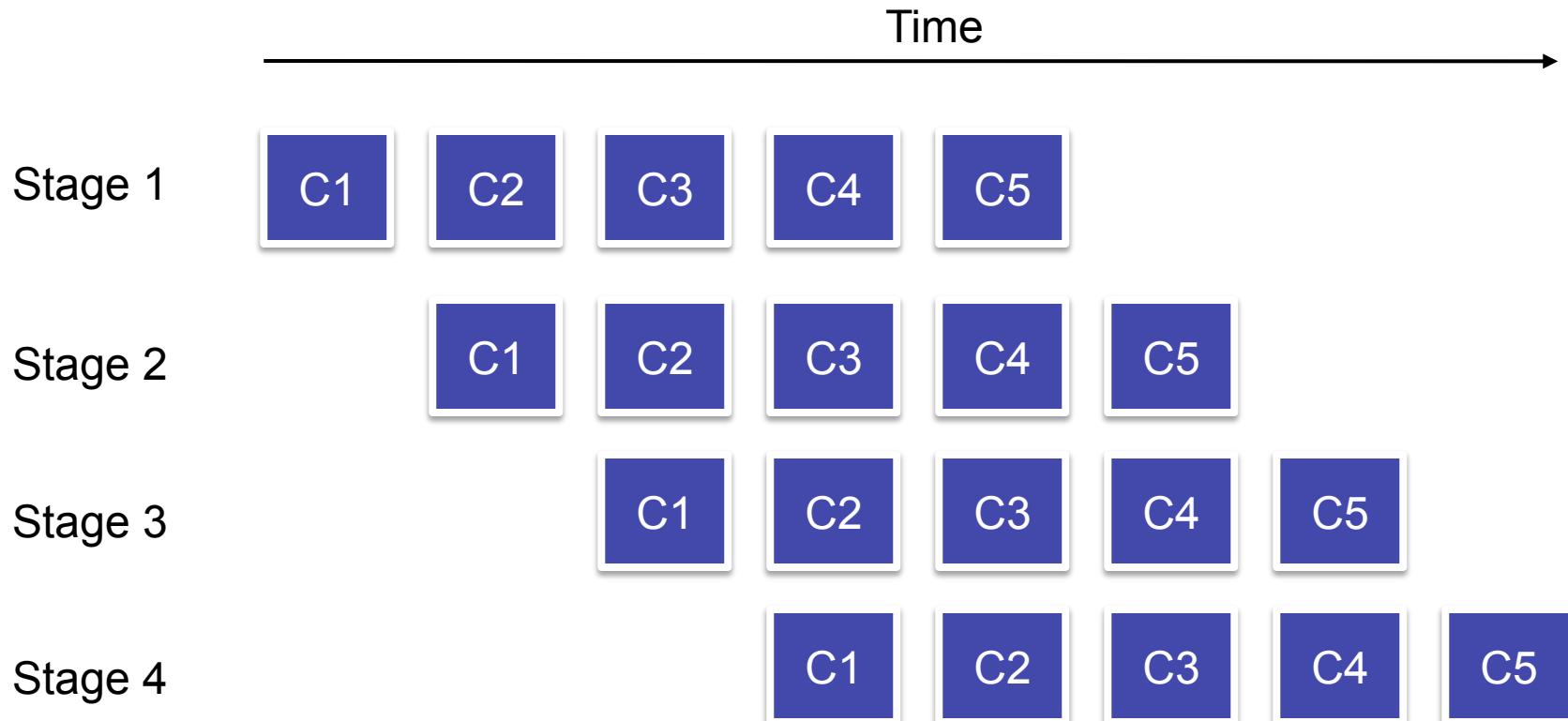
- Computation on different data sets
- Data can be viewed as flowing through a sequence of stages
- How to exploit potential parallelism



Pipeline pattern



- Assign each stage to a different worker



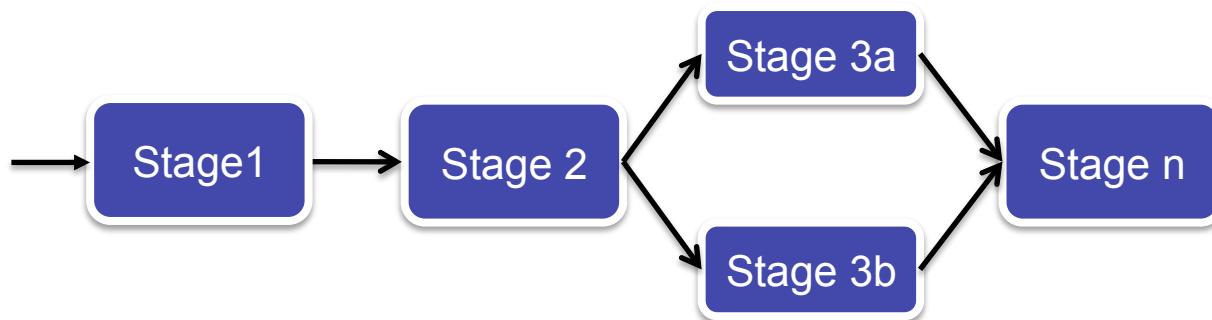
Pipeline pattern



- Pipeline can be of complex structure



Linear pipeline



Nonlinear pipeline



Pipeline pattern

- Example
 - Fourier-transform computations
 - Stage 1: Perform DFT on a set of data
 - Stage 2: Manipulate the result of the transform element-wise
 - Stage 3: Perform inverse DFT on the result of the manipulation
- Related pattern
 - Applications with multiple stages and without temporal dependencies between the inputs should be parallelized using task parallelism pattern

Event-based coordination pattern



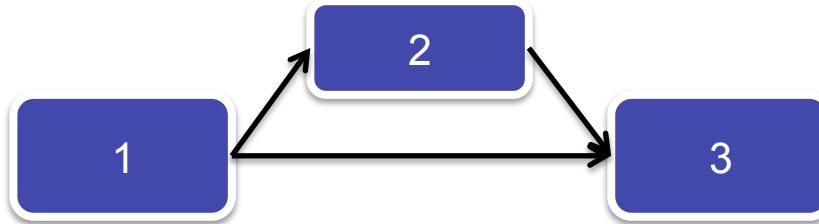
- Problem decomposed into a group of semi-independent tasks interacting irregularly
 - Interaction based on the flow of data
- Solution:
 - Express data flow using abstraction called *event*
 - Each event have a task generating it and a task processing it
 - Events define ordering between the tasks

```
Initialize
while (not done) {
    receive event
    process event
    send event
}
```

Event-based coordination pattern



- Asynchronous communication between the tasks
 - Message-passing environment
 - Shared queue in shared memory environment
- Event ordering:
 - Tasks may process events in order different than they were generated



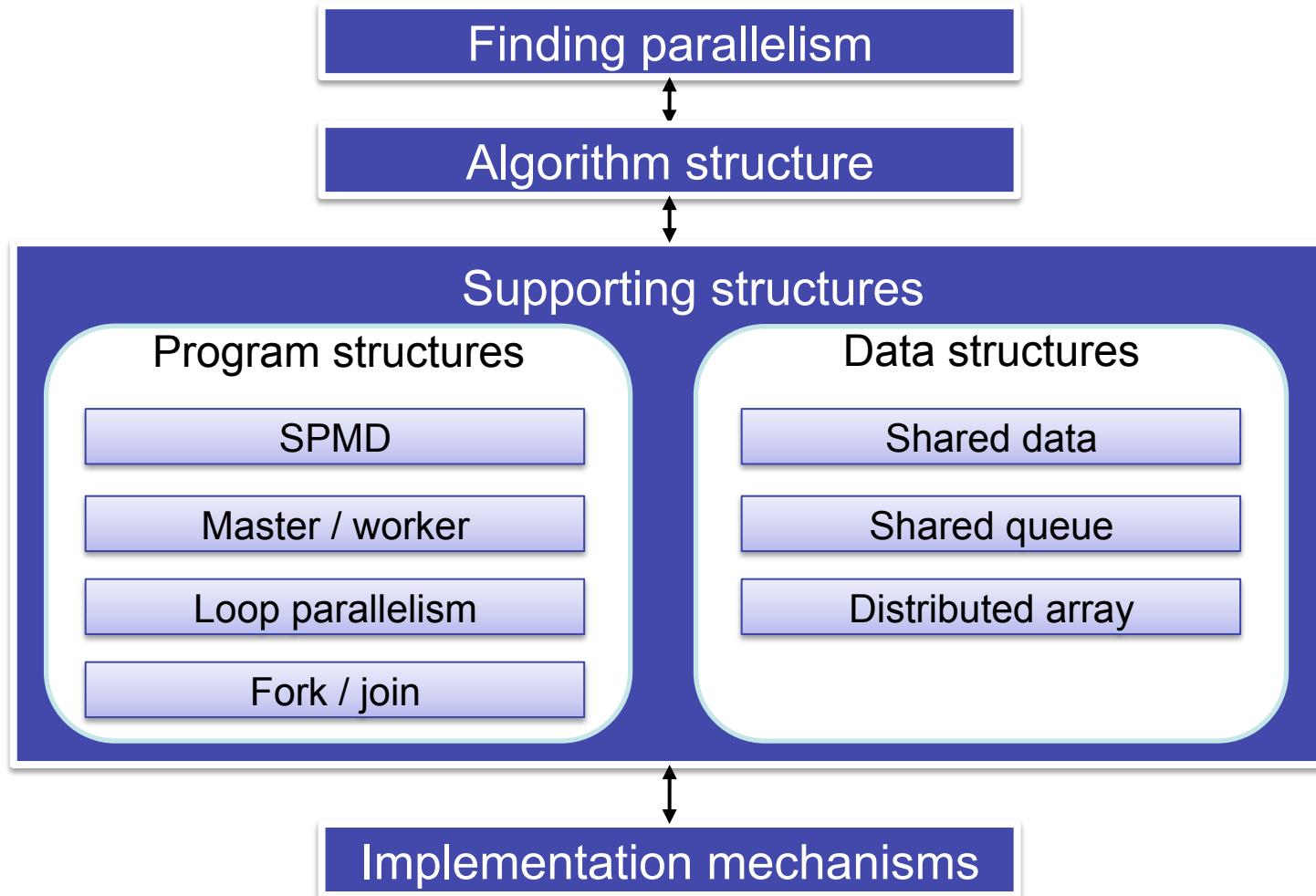
- Avoid deadlocks
 - Each task requiring an event from other before processing

Event-based coordination pattern



- Example
 - DPAT simulation
 - Simulation to analyze air traffic control systems
 - CSWEB
 - Simulation of voltage output of combinatorial digital circuits
- Related pattern
 - Similar to pipeline
 - Communication in pipeline pattern is loosely synchronous, while communication in event-based coordination is asynchronous
 - Pipeline pattern has a fixed structure (number of stages and interaction), but in event-based coordination, the structure can be more dynamic

Supporting structures design space



Forces



- **Clarity of abstraction** – is the parallel algorithm clearly apparent?
 - Good for writing correct code
- **Scalability** – how many PEs can the parallel program efficiently utilize?
- **Efficiency** - how close does the program come to fully utilize the resources of the parallel computer?
- **Maintainability** – is the program easy to debug, verify, and modify?
- **Environmental affinity** – is the program well aligned with the programming environment and hardware of choice?
- **Sequential equivalence** – does the parallel version produce the same results as the sequential version? If not, is the relationship between them understood?

Algorithm structure vs. program structures



	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
SPMD	****	***	****	**	***	**
Loop parallelism	****	**	***			
Master / worker	****	**	*	*	*	*
Fork / join	**	****	**		****	****

Supporting structures vs. Programming environment



	openMP	Java
SPMD	***	**
Loop parallelism	****	***
Master / worker	**	***
Fork / join	***	****

SPMD pattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Problem
 - How can programmers structure their programs to make the interactions between different UEs more manageable
- Context
 - Most commonly used pattern for structuring parallel programs
 - Particularly relevant for
 - Task parallelism pattern
 - Geometric decomposition pattern
 - Divide and conquer pattern
 - Recursive data pattern

SPMD pattern



- Solution
 - Initialize
 - Obtain a unique identifier
 - Run the same program on each UE
 - Use the unique identifier to differentiate behavior on different Uses
 - Distribute data
 - Finalize

SPMD pattern



- Discussion
 - Clarity of abstraction sometimes challenging
 - Overheads associated with start-up and termination are segregated at the beginning and at the end of the program
 - SPMD pattern does not assume anything concerning hardware
 - Suitable for any MIMD computers
 - A big strength of SPMD pattern



Numerical integration

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

- Estimate value of π using the integral (Serial version)

```
int i, num_steps = 100000;
double x, pi, step, sum = 0.0;

step = 1.0/(double) num_steps;

for (i = 0; i < num_steps; i++)
    x = (i + 0.5) * step;
    sum += 4.0/(1.0 + x*x)
}
pi = step * sum;
```

Numerical integration



- Estimate value of π using the integral (SMPD version)

```
int num_steps = 100000;
double pi, step, sum = 0.0;

step = 1.0 / (double) num_steps;

#pragma omp parallel reduction(+:sum)
{
    int i, id = omp_get_thread_num();
    int numthreads = omp_get_num_threads();
    double x;

    for (i = id; i < num_steps; i += numthreads)
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x*x)
}
pi = step * sum;
```

Master / worker pattern



- Problem
 - How can programmers structure their programs if the design is dominated by the need to dynamically balance work among UEs?
- Context
 - Sometimes balancing the load dominates the design
 - Task workloads are highly variable
 - Computations are not mappable onto a simple loop
 - Not applicable if tasks are tightly coupled (share read & write data)

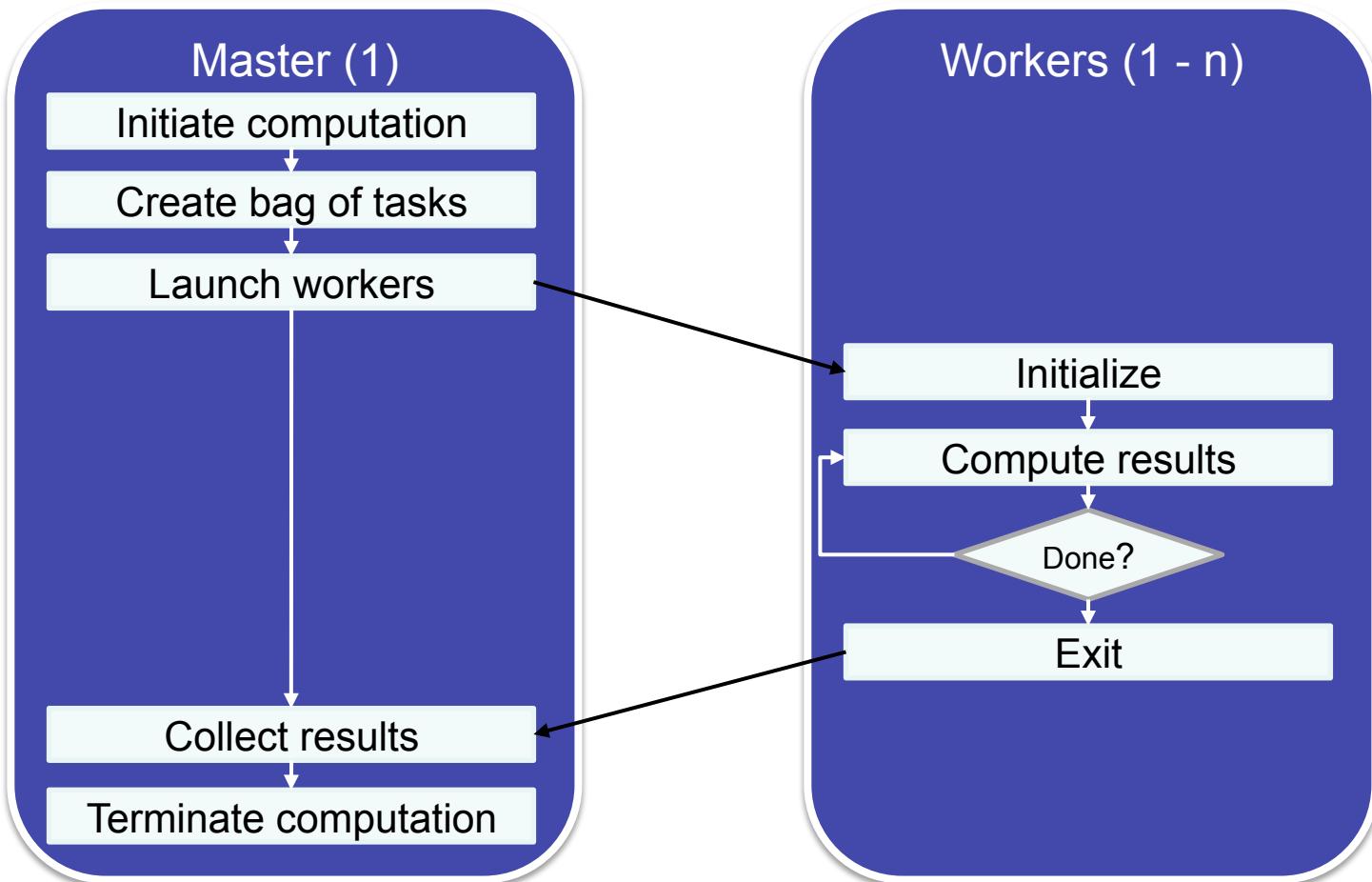
Master / worker pattern



- Forces
 - Work for each task varies unpredictably
 - Load balancing may impose expensive overhead
 - Scheduling should revolve around smaller number of larger tasks
 - Code to produce optimal load balancing maybe error prone
 - Need to make trade-offs between optimal distribution and easy to maintain code

Master / worker pattern

- Solution



Master / worker pattern



- Discussion
 - Have good scalability
 - Management of task bag has overhead
 - Not a problem if tasks have average workload greater than time required for management
 - Not tied to a specific hardware
- Variations
 - Master may turn into a worker after creating tasks
 - Each worker may maintain its own task queue
 - Steal from another worker if local queue is empty

Loop parallelism pattern



- Problem
 - If a serial program has computation intensive loops, how to parallelize it?
- Context
 - Most scientific programs use iterative constructs
 - Program's code is available but has poorly understood algorithm
 - Iterations of the loop should work as mostly independent tasks
 - Less loop carried dependencies
 - OpenMP was primarily created for loop parallelism pattern



Loop parallelism pattern

- Forces
 - Sequential equivalence
 - Incremental parallelism
 - Memory utilization
 - Good performance relies on memory hierarchy of the system
- Solution
 - Find bottlenecks
 - Locate most computationally intensive loops
 - Eliminate most of the loop-carried dependencies
 - Parallelize the loops
 - Split up the iterations among UEs
 - Optimize scheduling

Loop parallelism pattern



- Example

```
int i;
int num_steps = 100000;
double x, pi, step, sum = 0.0;

step = 1.0 / (double) num_steps;

#pragma omp parallel for private(x) reduction(+:sum)
for (i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x*x)
}
pi = step * sum;
```

Fork / join pattern



- Problem
 - Some programs have variable number of parallel tasks at different stage of execution. How to parallelize such programs?
- Context
 - Tasks are generated dynamically (forked) and later terminated (joined)
 - Example:
 - A divide & conquer algorithm, problem is split into multiple subproblems (forked), which may fork more subproblems recursively
 - When all tasks to handle a particular split are finished, they are joined back with parent task that continues further computation

Fork / join pattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Forces
 - Algorithms imply relationship between tasks
 - 1-1 mapping of tasks to UEs is natural, but must be balanced
 - UE creation and destruction is costly

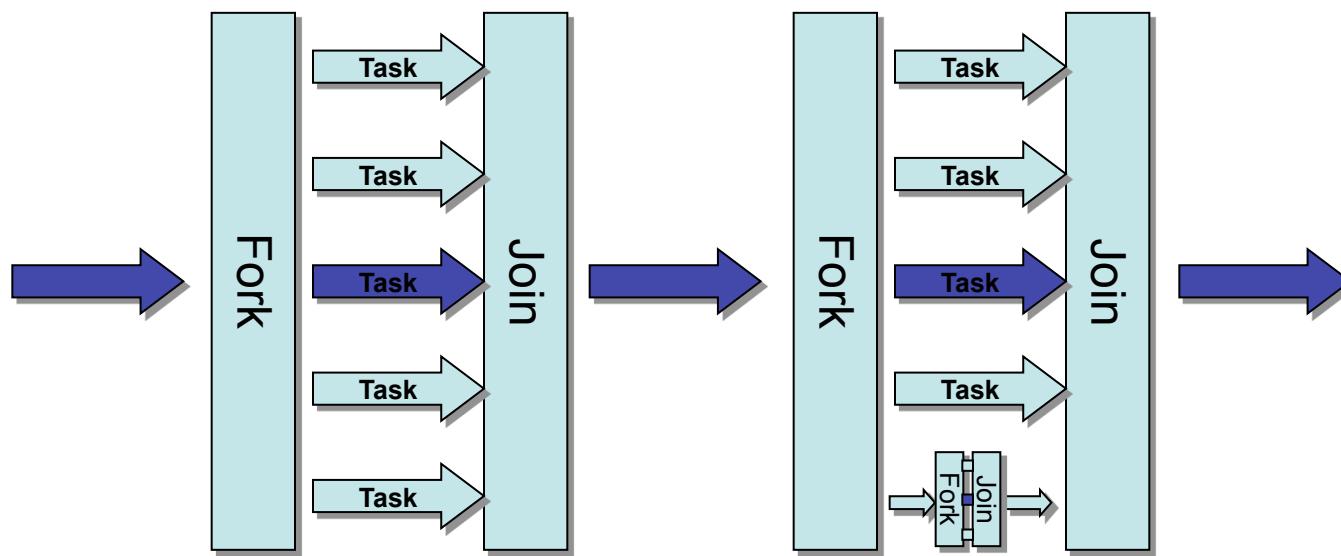


Fork / join pattern

- Solution
 - **Direct task/UE mapping**
 - Simply map each sub-task to a UE
 - Creation of new task means creation of new UE
 - Termination of a task means related UE is destroyed
 - Standard openMP model
 - Creation and destruction of UEs is expensive
- **Indirect task/UE mapping**
 - Avoid dynamic UE creation
 - Create a static set of UEs in the start of a program
 - Mapping of tasks to UEs occur dynamically
 - Complicated to implement
 - Results in efficient program

Fork / join pattern

- Example





Fork / join pattern

- Discussion
 - Divide and conquer pattern use fork/join pattern
 - If UEs are forked just to handle iteration of a single loop in program then it is fork/join pattern
 - Not loop parallelism pattern
 - Master/worker pattern can be used to implement the indirect-mapping solution

Shared data pattern



- Problem
 - How to manage shared data between parallel tasks?
- Context
 - Sometimes data is needed by all the tasks
 - At least one data structure (e.g. tree, list) is accessed by all tasks
 - At least one task modifies the data structure
 - The tasks potentially need to use the modified data structure

Shared data pattern



- Forces
 - The final result should be correct for any ordering of the tasks
 - Overhead for the management of data structure should be kept low for program to run efficiently
 - Techniques for data sharing management may limit the number of tasks
 - Complex data sharing management constructs are difficult to maintain

Shared data pattern



- Solution
 - Be sure this pattern is needed
 - Try to find another parallelism strategy
 - Define an Abstract Data Type (ADT)
 - Should provide a fixed number of operations on data
 - E.g. queues, stacks, trees etc.
 - Implement appropriate concurrency control protocol
 - One at a time execution
 - Noninterfering sets of operations
 - Readers/writers
 - Reduce size of critical section
 - Nested locks
 - Application specific semantic relaxation

Shared data pattern



- Discussion
 - Shared queue and distributed array are specific types of shared data structures.
 - Many problems that use shared data pattern use task parallelism pattern for algorithm structure
- Examples
 - Phylogeny problem
 - Each task handles one node of tree but needs information from other parts of tree as well
 - Gröbner basis problem
 - Computations use pairs of polynomials to generate new polynomials and adding them to master set under some conditions

Shared queue and Distributed array patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Problem
 - How to safely share a queue between parallel tasks?
 - How to safely share an array between parallel tasks?
- Context
 - Many effective parallel algorithms require a queue shared among UEs
 - Master/worker pattern
 - Arrays are fundamental data structures for the most of scientific problems
 - Distribute array in a way that the needed by UE is stored near to it at the right time

Shared queue and Distributed array patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Forces
 - Load balance
 - Effective memory management
 - Clarity of abstraction

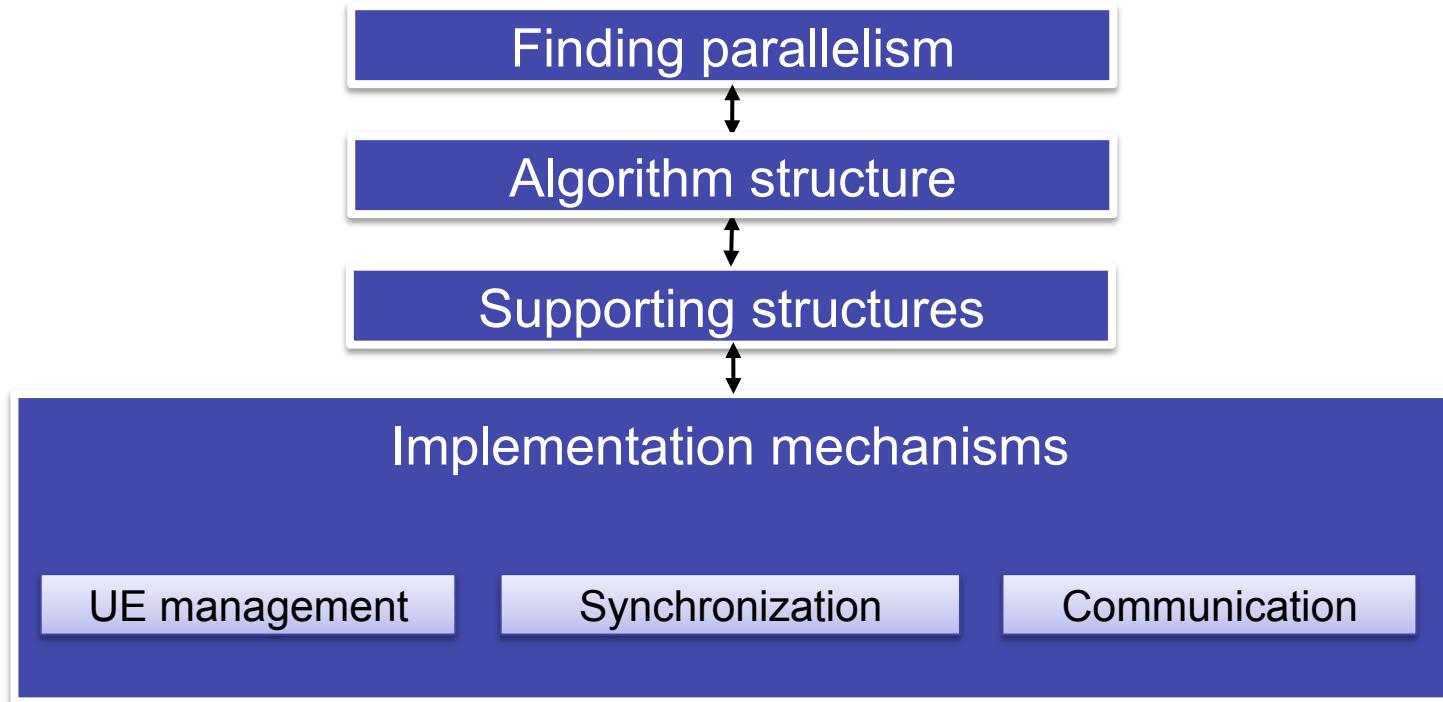
Shared queue and Distributed array patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Solution
 - Shared queues
 - ADT: list of data objects, enqueue, dequeue
 - Mostly available in libraries of languages
 - Java.util.concurrent, Intel TBB etc.
 - Implement shared queue as an instance of shared data pattern
 - Distributed arrays
 - Partition array into blocks 1D, 2D, etc.
 - Map these blocks onto the UEs
 - Each UE should get almost equal amount of work

Implementation mechanisms design space



UE management



- Two type of UEs:
 - Threads
 - Processes
- How to manage (creation/destruction) in different languages

```
Class M extends Thread
{public void run() {code} }

Thread t = new M(); //create thread
t.start(); //launch thread
```

Java

```
#pragma omp parallel
{structured block}
```

openMP

Synchronization



- Enforce a constraint on the order of events occurring in tasks
- Types of synchronization
 - Memory synchronization and fences
 - Guarantees the consistent view of data to all UEs
 - *flush* in openMP, *synchronized* in Java
 - Barriers
 - Every member UE of a collection should arrive to a point before any UE continuing ahead
 - *Barrier* in openMP, *CountDownLatch* in Java
 - Mutual exclusion
 - Some operations must be done by one UE at a time
 - *Critical* in openMP, *lock* in Java

Communication



- Sharing of data between UEs
- Types of communication
 - Message passing
 - A message consisting of data is passed between UEs
 - In shared memory this can be achieved by putting message in shared area and using event synchronization
 - Collective communication
 - Broadcast
 - Barrier
 - Reduction

Summary



- Parallel design patterns are used to parallelize a problem
- Patterns for four spaces
 - Finding parallelism
 - Algorithm structure
 - Supporting structure
 - Implementation mechanisms
- User may need to jump back and forth to get a correct solution
- The solution should be
 - Flexible
 - Efficient
 - Simple
 - scalable



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

PARALLEL PROGRAMMING IN JAVA

Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Basics
 - Creating threads
 - Life cycle of a thread
 - Critical section and race conditions
- Java.util.concurrent:
 - Synchronization of threads
 - Control constructs
 - Performance of control constructs
 - Constructs for wait and notification
 - Additions to Java library
 - Synchronizer
 - Atomic data types
 - Collections
 - Synchronized and concurrent data types



Agenda (Cont.)

- Constructs for asynchronous execution:
 - Callable
 - Future
 - Executor
 - Energy consumption and time of programming styles
- Java memory model
 - Memory model
 - The fundamentals: happens-before ordering
 - Using volatile
 - Recommendations

Creating threads



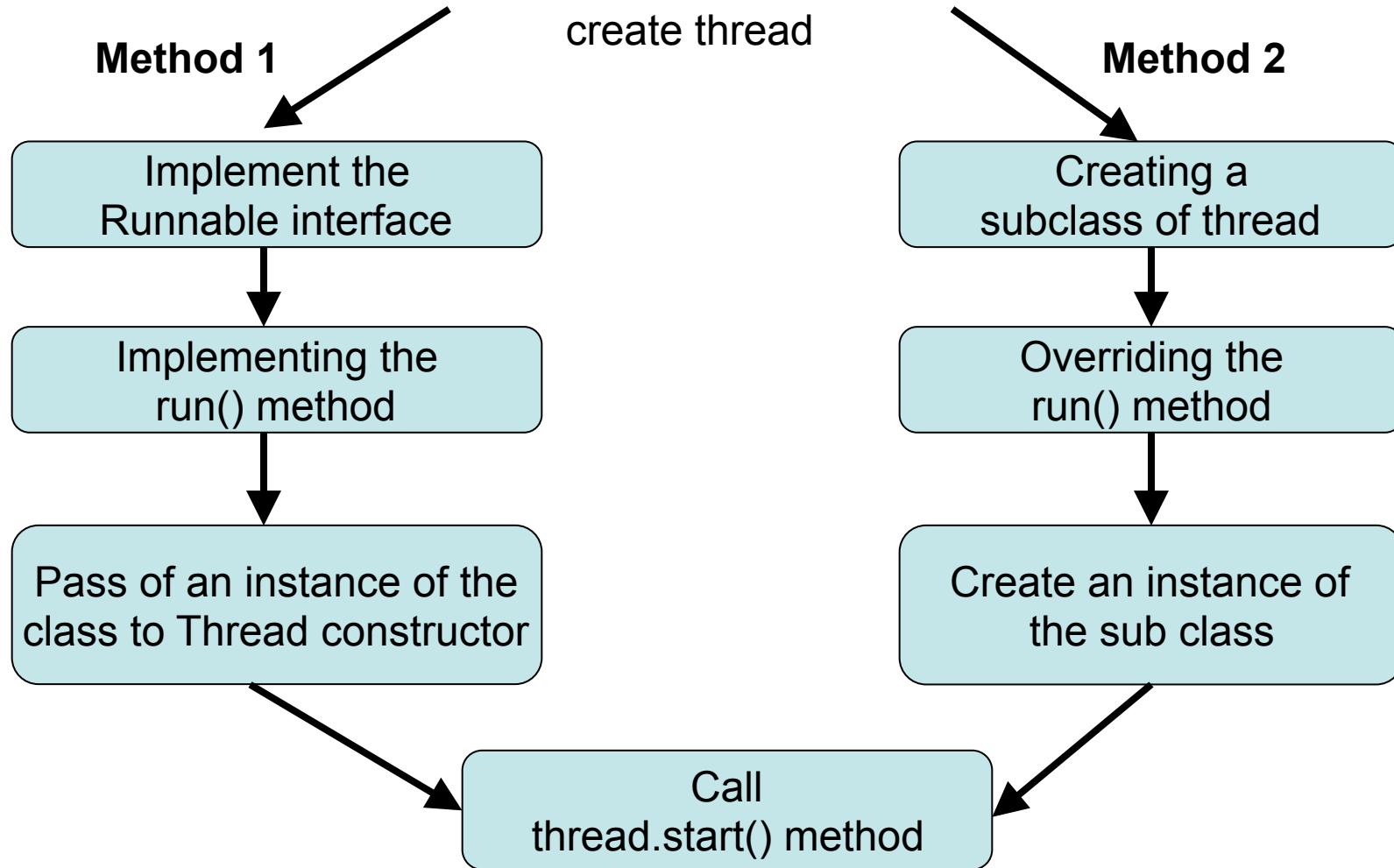
- Since Java 1.0:
 - Built-in classes and interfaces for parallel programming:
 - Interface `java.lang.Runnable`

```
public interface Runnable {  
    public abstract void run();  
}
```

- Class `java.lang.Thread`

```
public class Thread extends Thread {  
    public Thread(String name);  
    public Thread(Runnable target)  
    public void start();  
    public void run(){ code }  
    ...  
}
```

Creating threads



Creating threads-example method 1



- Class that implements Runnable

```
class ComputeRun implements Runnable {  
    long min, max;  
    ComputeRun(long min, long max) {  
        this.min = min; this.max = max;  
    }  
    public void run() {  
        // Parallel task  
    }  
}
```

- Create and launch control thread:

```
ComputeRun c = new ComputeRun(1,20);  
new Thread(c).start();
```

- Start the new thread of control. Only generates the new activity which returns the start() method
- Immediately returns the new control thread continues concurrently.
- No restart: start() must be called only once. not directly invoke run()



Creating threads-example method 2

- Class, which inherits from thread

```
class ComputeThread extends Thread {  
    long min, max;  
    ComputeThread(long min, long max) {  
        this.min = min; this.max = max;  
    }  
    public void run() {  
        // Parallel task  
    }  
}
```

- Create and launch control thread

```
ComputeThread t = new ComputeThread(1,10);  
t.start();
```

- Java does not support multiple inheritance



Life cycle of a thread

- A thread can be in one of six states (query with `getstate()`):

NEW

- Thread has been created, but not yet called `start()`

RUNNABLE

- Thread is running

BLOCKED

- Thread is not running because it's waiting on a resource (e.g. lock or an I/O device)

WAITING

- Thread is waiting for another thread to perform a task

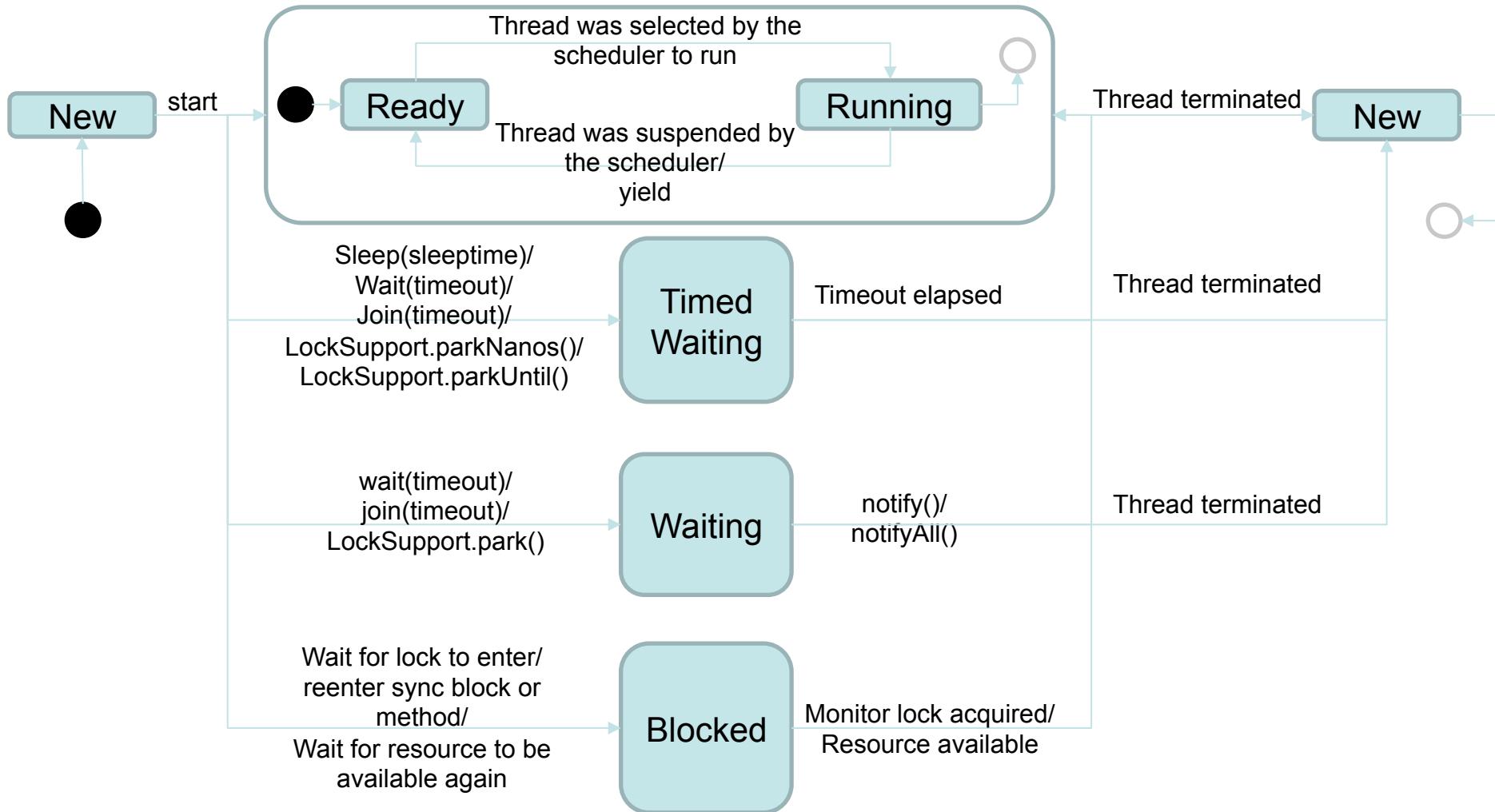
TIMED_WAITING

- For a specified interval of time
 - Transitions back to runnable when the interval expires or when the event occurs

TERMINATED

- Execution completed:
 - Method `run()` normally stopped or by an exception

Life cycle of a thread (flow)



Critical Sections and race conditions



- Inherently, no problem with multiple threads in an application.
- Problem: when multiple threads access same resources (e.g. variables, arrays, objects etc.).

```
public class Counter {  
    protected long count = 0;  
  
    public void add(long value){  
        this.count = this.count + value;  
    }  
}
```

this.count = 0;

A: Reads this.count into a register (0)
B: Reads this.count into a register (0)
B: Adds value 2 to register
B: Writes register value (2) back to memory.

this.count now equals 2

A: Adds value 3 to register
A: Writes register value (3) back to memory.

this.count now equals 3

Some points



- Thread-safe: A piece of code with no race conditions.
- Heap is shared between threads of a process:
 - If we instantiate an object and give it to different threads:
 - All fields declared inside the class are shared.
 - Private, public or protected access modifiers not relevant to sharing!
 - Local variables in methods of the class are also shared!
- Each thread has its own stack



Synchronization of threads-usage

Number	Control constructs	# projects	% of total
1	Synchronized	40	86.96%
2	Lock	19	41.30%
3	ReadWriteLock	15	32.61%
4	ReentrantReadWriteLock	14	30.43%
5	CountDownLatch	12	26.09%
6	ReentrantLock	11	23.91%
7	Semaphore	9	19.57%
8	Condition	5	10.87%
9	CyclicBrarrier	4	8.70%
10	AbstractQueuedSynchronizer	2	4.35%
11	LockSupport	2	4.35%
12	AbstractOwnableSynchronizer	0	0%
13	Exchanger	0	0%
14	Phaser	0	0%

46 Java projects including:

- Casandra: A distributed DBMS
- Neo4J: A graph database
- Consulo IDE: An IDE for many programming languages.
- JetBrains MPS: An environment for language definition.

An empirical study on parallelism in modern open-source projects, M. Kiefer, D. Warzel, W. Tichy, SEPS 2015.

Control constructs (Synchronized)



- Every object has an intrinsic lock
- A thread *owns* the lock between the time it has acquired the lock and released the lock.
- Other threads block
- Two synchronization idioms: *synchronized methods* and *statements*.

```
/*synchronized method*/
synchronized void foo(){
    // Body is critical
    // section
}
```

```
/*synchronized block*/
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Control constructs-Synchronized (Cont.)

- Block oriented
- Cannot be used for constructors
- Unlike synchronized methods, synchronized statements must specify an object that provides the lock

Control constructs-Lock



- Additional lock construct without block orientation

```
public interface Lock {  
  
    void lock();  
  
    void lockInterruptibly() throws InterruptedException;  
  
    boolean tryLock();  
  
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  
  
    void unlock();  
  
    Condition newCondition();  
}
```

Control constructs-Lock

Advantages



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Provides extensive operations:
 - Non-blocking or optimistic attempt to acquire a lock using tryLock().
 - Attempt to acquire the lock that can be interrupted by checking lockInterruptibly().
 - Attempt to acquire a lock that can timeout by checking tryLock(long, TimeUnit).
- Ability to have multiple condition variables
 - E.x producer-consumer:
 - Producer must not produce a new item until the previous item has been consumed
 - Consumer must not consume an item that hasn't been produced
 - Two conditions: itemProduced, itemConsumed

Control constructs-Lock

Advantages (Cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Synchronized blocks are unfair since locking is implicit.
- isLocked(): Check if the lock is held
- getQueuedThreads(): Get list of threads waiting on the lock
- Not limited to block-structure

```
Lock l = ...;  
l.lock();  
try {  
    // access the resource protected by this lock  
} finally {  
    l.unlock();  
}
```

Control constructs-ReadWriteLock



- Interface:

```
public interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```

- Multiple reader threads
- Writes are exclusive
- Useful for parallel algorithms with commonly read and rarely write or update:
 - Actual performance depends on frequency of read-writes, duration of each operation, contention for the data, i.e. number of threads that read-write the data at same time.

Control constructs-ReentrantLock and ReentrantReadWriteLock



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Prioriterizes the thread last successfully locking, but not yet unlocking it.
- There was some problem with `ReadWriteLock` in JDK 5:
 - Multiple readers can lock all writer threads
 - Writers starve if readers enter critical section in a tag-team fashion
- Constructor accepts a fairness parameter:
 - True indicates granting access to longest-waiting thread

Control constructs-Lock

Disadvantages



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- With the increased flexibility comes additional responsibility:
 - Need to wrap lock acquisitions in a try/finally block.
 - The code does not look as pretty as when using synchronized!
- The synchronized keyword can be put in method definitions which avoids the need for a block which reduces nesting.

Control constructs-Performance



- Doug Lea, „*The java.util.concurrent synchronizer framework*”, Science of Computer Programming, Elsevier North-Holland, Inc., 2005, 58, 293-309.
- Benchmark program:
 - A thread calls shared random number generator with probability S, which is protected by the lock.
 - Purpose of randomization:
 - Used to decide whether to lock or not. S=1 indicates a high contentious lock. S=0 indicates per-lock access, i.e. no contention.
 - Code in loops cannot be trivially optimized.
 - Tests on 4 x 86 (Linux 2.4) and 4 UltraSparc machines (Solaris 9), SUN J2SE 5.0 JDK. all programs were carried out before the measurements 20 x.

Control constructs-Performance (Cont.)



- Experiments with different types of locks
 - **Builtin:** Block-based synchronized construct
 - **Mutex:** Own, simple Mutex class that implements an explicit lock.
 - **Reentr:** ReentrantLock
 - Thread that holds the lock, can recall locking methods. This will return immediately without synchronization overhead.
 - `getHoldCount()` specifies how often the thread has locked when you share must call `unlock()` just as often .
 - **fair:** ReentrantLock set in its "fair" mode.
 - Prefers longer waiting threads.

Control constructs-Performance (Cont.)



- Estimates of the overhead time in ns.

Builtin: synchronized construct
Mutex: explicit lock
Reentr: ReentrantLock
fair: ReentrantLock in fair mode

Name	builtin	mutex	reentr	fair	builtin	mutex	reentr	fair
1P	18	9	31	37	521	46	67	8327
2P	58	71	77	81	930	108	132	14967
2A	13	21	31	30	748	79	84	33910
4P	116	95	109	117	1146	188	247	15328
1U	90	40	58	67	879	153	177	41394
4U	122	82	100	115	2590	347	368	30004
8U	160	83	103	123	1274	157	174	31084
24U	161	84	108	119	1983	160	182	32291

Computers

1P (1 × 900 MHz Pentium 3)
2P (2 × 1400 MHz Pentium 3)
2A (2 × 2000 MHz Athlon)
4P (2 × 2400 MHz hyperthreaded Xeon)
1U (1 × 650 MHz Ultrasparc2)
4U (4 × 450 MHz Ultrasparc2)
8U (8 × 750 MHz Ultrasparc3)
24U (24 × 750 MHz Ultrasparc3)

Overhead of synchronization constructs during execution with 1 thread. Time is difference between code synchronization ($S = 1$) vs. code without synchronization constructs ($S = 0$).

Additional expenses (overhead) per lock for $S = 1$ (each thread calls always common generator) and 256 parallel threads.

Constructs for waiting and notification



- Sometimes threads must stop their execution (and release locks), until a specific event occurs and then continue execution
- Methods in `java.lang.Object`

```
public final void wait() throws InterruptedException;  
  
public final void notify();  
  
public final void notifyAll();
```

Constructs for waiting and notification (Cont.)



- Each object maintains an internal queue of waiting threads:
 - When a thread calls the wait method of an object o, then it causes the thread to wait until
 1. Another thread invokes notify() method for this object
 2. Another thread invokes notifyAll() method for this object,
 3. A specified amount of time has elapsed.
 - notifyAll(): If a thread calls it, object o wakes up all threads in the queue
 - notify():
 - Wakes up a single thread
 - If there are multiple threads waiting, one of them is chosen. The choice is arbitrary and depends on the implementation

Constructs for waiting and notification

Wait and NotifyAll



```
public class ConnectionPool {  
  
    private List<Connection> connections =  
        createConnections();  
  
    private List<Connection> createConnections()  
    {  
  
        List<Connection> conns = new  
        ArrayList<Connection>(5);  
  
        for (int i = 0; i < 5; i++) {  
  
            ... add a Connection to conns  
  
        }  
  
        return conns;  
    }  
  
}
```

```
// bad: woken thread can't start until we  
// come out of synchronized block!  
  
    updateStatistics(conn);
```

```
    public Connection getConnection() throws  
        InterruptedException {  
  
        synchronized (connections) {  
  
            while (connections.isEmpty()) {  
  
                connections.wait();  
  
            }  
  
            return connections.remove(0);  
        }  
  
    }  
  
    public void returnConnection(Connection conn) {  
  
        synchronized (connections) {  
  
            connections.add(conn);  
  
            connections.notify();  
        }  
    }  
}
```

Constructs for waiting and notification

Wait and NotifyAll (Cont.)

- A couple of small points:
 - After calling notify(), exit synchronized block ASAP!
 - Don't wait if the list isn't empty: because when you're writing the code, your mind is saying "I need to wait until there's something in the list".
 - Waiting thread can be interrupted: Better to surround it with try and catch.
 - An awoken thread from wait():
 - Doesn't know why it has awoken! So we have to do some checking

Additions to the Java library

java.util.concurrent



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Explicit or fine granular locking is error-prone
- Many data structures are not “thread-safe”: e.g. queues
- From Java 1.5:
 - [java.util.concurrent](#) provides additional classes to parallel programming
- Categories
 - Synchronizer
 - Atomic data types
 - Collections
 - Constructs for asynchronous execution

java.util.concurrent

Synchronizer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Maintains a state
- Lets a thread pass through or forces it to wait depending on the state.
- Five classes aid common special-purpose synchronization idioms:
 - Semaphore
 - Exchanger
 - CyclicBarrier
 - CountDownLatch
 - Phaser

java.util.concurrent

Synchronizer-Semaphore

- Is initialized to a fixed number of "Authorization"
- **acquire** block until a permit is available and then decrements the number of permits
- **Release** increments the number of permits

java.util.concurrent

Synchronizer-Exchanger

- Enables rendezvous and exchange of data between two threads through the exchange method
- Each thread calls exchange with an object (which is to be exchanged) and gets the object passed by the other thread at the rendezvous
- The first incoming thread blocks until it calls second exchange.

java.util.concurrent

Synchronizer-Exchanger (Example)



```
class FillAndEmpty {  
    Exchanger<DataBuffer> exchanger = new Exchanger();  
    DataBuffer initialEmptyBuffer = ...  
    DataBuffer initialFullBuffer = ...  
  
    class FillingLoop implements Runnable {  
        public void run() {  
            DataBuffer currentBuffer = initialEmptyBuffer;  
            try { while (currentBuffer != null) {  
                addToBuffer(currentBuffer);  
                if (currentBuffer.full()) currentBuffer =  
                    exchanger.exchange(currentBuffer); } }  
            catch (InterruptedException ex) {...handle...} }  
    }  
  
    class EmptyingLoop implements Runnable {  
        public void run() {  
            DataBuffer currentBuffer = initialFullBuffer;  
            try { while (currentBuffer != null) {  
                takeFromBuffer(currentBuffer);  
                if (currentBuffer.empty()) currentBuffer =  
                    exchanger.exchange(currentBuffer); } }  
            catch (InterruptedException ex) {...handle...} }  
    }  
    void start() { new Thread(new FillingLoop()).start();  
    new Thread(new EmptyingLoop()).start(); }  
}
```

Each thread passes an object to exchange method and obtains the object passed by the other thread.

idea:

- Buffer can be exchanged between threads
- Thread, which fills the buffer, passes full buffer to thread that flushes buffer
- Thread that flushes buffer, passing empty buffer thread filling buffer

java.util.concurrent

Synchronizer-CyclicBarrier



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Synchronizes a group of n threads
- Threads call await() method which blocks until all n threads reach it.
- It then allows threads to continue their execution (the barrier is reset).

java.util.concurrent

Synchronizer-CyclicBarrier (Example)



```
class Solver {  
    final int N;  
    final float[][] data;  
    final CyclicBarrier barrier;  
  
    class Worker implements Runnable {  
        int myRow;  
        Worker(int row) { myRow = row; }  
        public void run() {  
            while (!done()) {  
                processRow(myRow);  
  
                try {  
                    barrier.await();  
                } catch (InterruptedException ex) {  
                    return;  
                } catch (BrokenBarrierException ex) {  
                    return;  
                }  
            }  
        }  
    }  
}
```

```
public Solver(float[][] matrix) {  
    data = matrix;  
    N = matrix.length;  
    barrier = new CyclicBarrier(N,  
        new Runnable() {  
            public void run() {  
                mergeRows(...);  
            }  
        });  
    for (int i = 0; i < N; ++i)  
        new Thread(new Worker(i)).start();  
  
    waitUntilDone();  
}
```

Sample usage: parallel decomposition design:

1. Worker threads process each row of the matrix and then wait at the barrier until all rows have been processed.
2. Runnable barrier action executes and merges the rows.
3. If merger determines that a solution has been found then done() will return true and each worker will terminate.

java.util.concurrent

Synchronizer-CountDownLatch

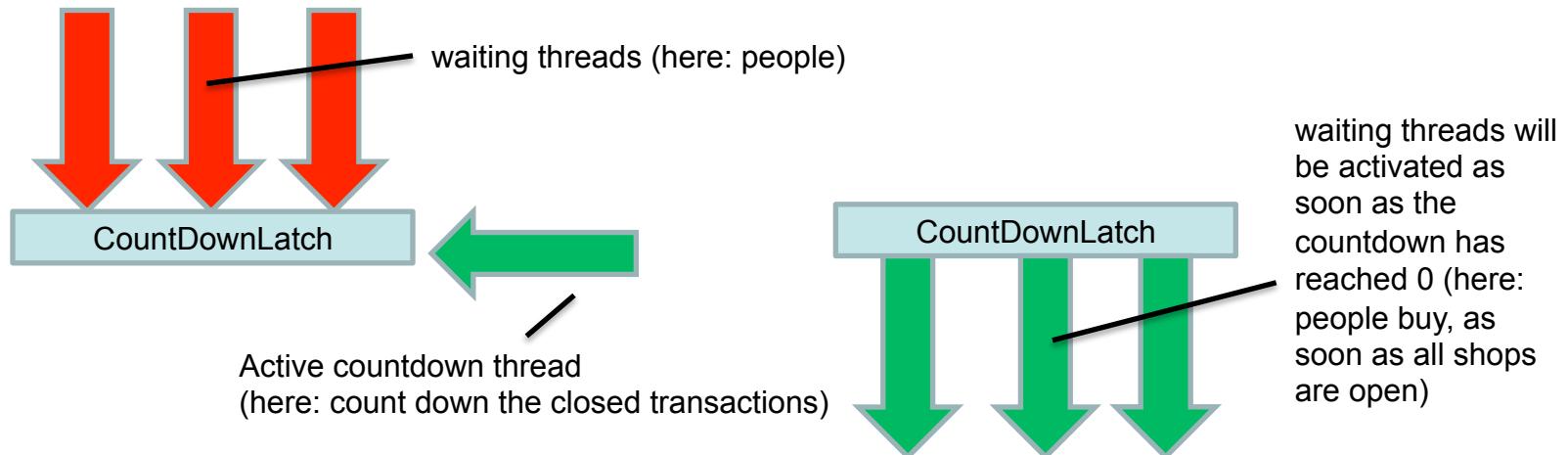


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Synchronizes multiple threads
- Each thread calls await() and blocks until countDown()
- Then is allowed all threads to continue their execution

java.util.concurrent Synchronizer-CountDownLatch (Example)

- People waiting in front of entrance of a shopping mall with several shops
- Initially all shops are closed, but gradually open (CountDown gradually update)
- Only when all shops are open, the front door to the mall opens. Then the people waiting at the Mall can go and do their shopping



java.util.concurrent

Synchronizer-CountDownLatch (Example)



```
public class CountDownLatchSample {  
    private CountDownLatch latch;  
    private int shops = 5;  
  
    private class Shopper implements Runnable{  
        public void run() {  
            try {latch.await();}  
            catch (InterruptedException e) {...}  
            System.out.println("Juhu, shopping!");}  
    }  
  
    private class Shop implements Runnable {  
        public void run() {  
            System.out.println("Opening shop..");  
            latch.countDown();    }  
    }  
  
    public CountDownLatchSample() {  
        latch = new CountDownLatch(shops);  
    }  
}
```

```
private void simulate() {  
    int shopper = 10;  
    for (int i = 0; i < shopper; i++) {  
        Thread t = new Thread(new Shopper());  
        t.start();  
    }  
    for (int i = 0; i < shops; i++) {  
        Thread t = new Thread(new Shop());  
        t.start();  
    }  
}  
  
public static void main(...) {  
    new CountDownLatchSample().simulate();  
}  
} // end class CountDownLatchSample
```

java.util.concurrent



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Categories
 - Synchronizer
 - **Atomic data types**
 - Collections
 - Constructs for asynchronous execution

java.util.concurrent

Atomic data types



- **java.util.concurrent.atomic**
 - Contains classes with atomic operations on data types such as Boolean, integer ...
- Example: [AtomicInteger](#)
 - Atomic executable operations:
 - compareAndSet(int expect, int update)
 - addAndGet(int delta)
 - getAndAdd(int delta)
 - decrementAndGet()
 - getAndIncrement()
 - ...

java.util.concurrent

Atomic data types



Data structure	# projects	% of total
AtomicInteger	28	60.87%
AtomicBoolean	17	36.96%
AtomicLong	16	34.78%
AtomicReference	13	28.26%
AtomicReferenceArray	7	15.22%
AtomicIntegerArray	5	10.87%
AtomicLongArray	4	8.70%
AtomicIntegerFieldUpdater	2	4.35%
AtomicReferenceFieldUpdater	2	4.35%
AtomicLongFieldUpdater	1	2.17%
AtomicMarkableReference	1	2.17%
AtomicStampedReference	0	0.00%

An empirical study on parallelism in modern open-source projects, M. Kiefer, D. Warzel, W. Tichy, SEPS 2015.

java.util.concurrent

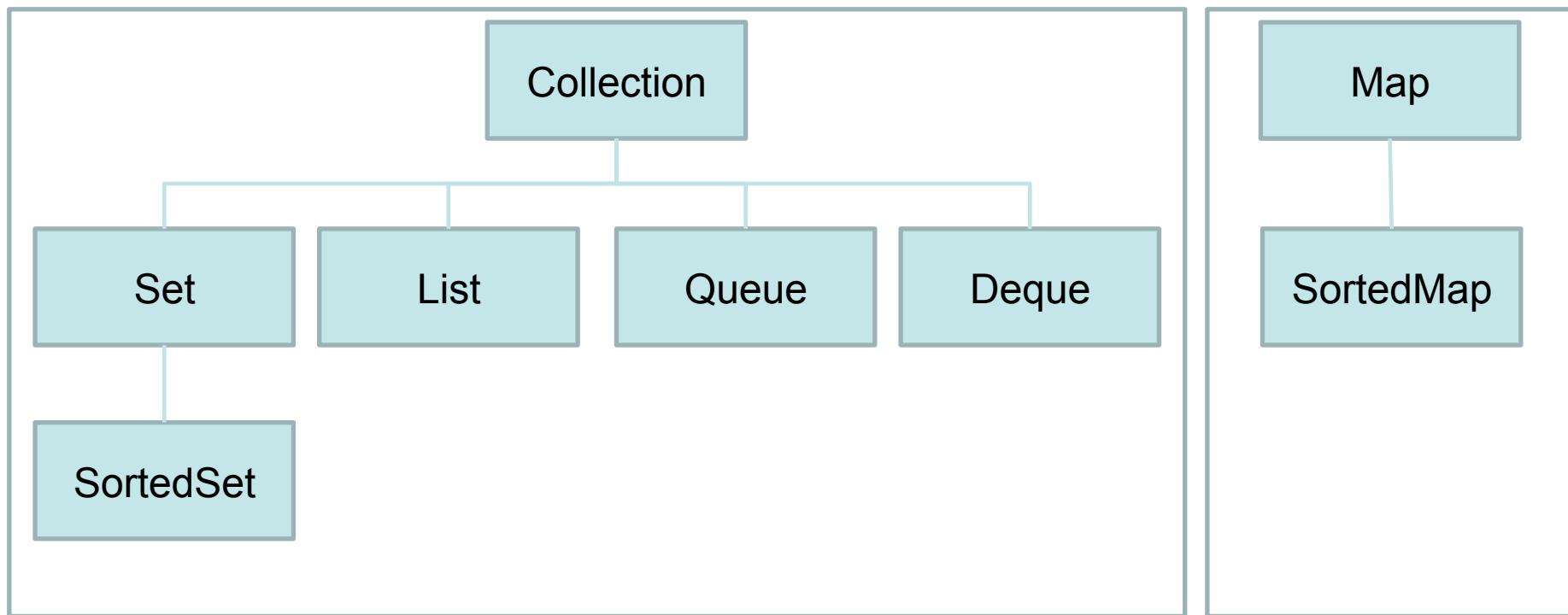


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Providing useful functionality is needed.
- Categories
 - Synchronizer
 - Atomic data types
 - **Collections**
 - Constructs for asynchronous execution

java.util.concurrent

Types of collections



java.util.concurrent

Types of collections (Cont.)



- Set:
 - Un-ordered collection, without duplicates.
 - Three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.
 - HashSet: stores elements in a hash table. Best performance
 - LinkedHashSet: implemented as a combination of hash table and a linked list. Slightly higher performance cost.
 - TreeSet: stores elements in a red-black tree, orders them based on their values. Substantially slower than HashSet.
- List:
 - May contain duplicates
 - Ordered and indexed collection

java.util.concurrent

Types of collections (Cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Queue:
 - Typically, not necessarily, order elements in a FIFO manner.
- Deque:
 - Both as FIFO & LIFO. Elements inserted, retrieved and removed at both ends.
- Map:
 - Is not a true Collection.
 - Maps keys to values
 - Cannot contain duplicate keys.
 - Three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet.

java.util.concurrent Collections-usage



Data structure	# projects	% of total
ConcurrentHashMap	26	56.52%
SynchronizedMap	15	32.61%
CopyOnWriteArrayList	14	30.43%
ConcurrentLinkedQueue	11	23.91%
SynchronizedList	10	21.74%
SynchronizedSet	9	19.57%
CopyOnWriteArraySet	6	13.04%
ConcurrentSkipListSet	3	6.52%
SynchronizedCollection	3	6.52%
ConcurrentLinkedDeque	1	2.17%
ConcurrentSkipListMap	1	2.17%
SynchronizedSortedMap	1	2.17%
SynchronizedSortedSet	1	2.17%
ConcurrentNavigableMap	0	0.00%

An empirical study on parallelism in modern open-source projects, M. Kiefer, D. Warzel, W. Tichy, SEPS 2015.

java.util.concurrent

Concurrency in collections



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Two categories:
 - Synchronized collections
 - Concurrent collections.

java.util.concurrent

Concurrency in collections (Cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Synchronized Collections**

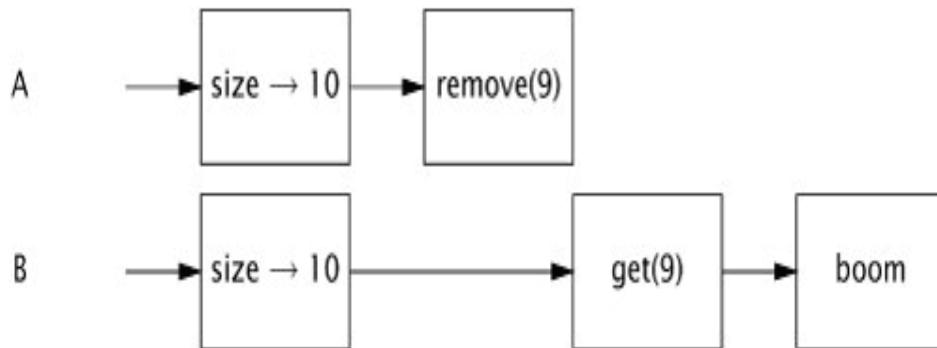
- Collections.synchronizedXxx factory methods.
- Achieve thread safety by synchronizing every public method
- Only one thread at a time can access the collection
- Hold a lock for the duration of each operation
 - Big problem when to iterate over the entire collection
- No read/write access at the same time
- Good for simple inserts and lookups
- Sometimes additional client side locking is required to guard compound actions: iteration, navigation.
- Without locks they may not behave as you expect!

java.util.concurrent Synchronized Collections-problem



```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}
```

```
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```



1. A calls getLast on a Vector with ten elements.
2. B calls deleteLast on ten elements too.
3. getLast throws
ArrayIndexOutOfBoundsException

java.util.concurrent

Synchronized Collections-solution



- Solution: Make getLast and deleteLast atomic by client-side locking!

```
public static Object getLast(Vector list) {  
    synchronized (list) {  
        int lastIndex = list.size() - 1;  
        return list.get(lastIndex);  
    }  
}  
  
public static void deleteLast(Vector list) {  
    synchronized (list) {  
        int lastIndex = list.size() - 1;  
        list.remove(lastIndex);  
    }  
}
```

java.util.concurrent

Concurrency in collections (Cont.)



- **Concurrent Collections:**

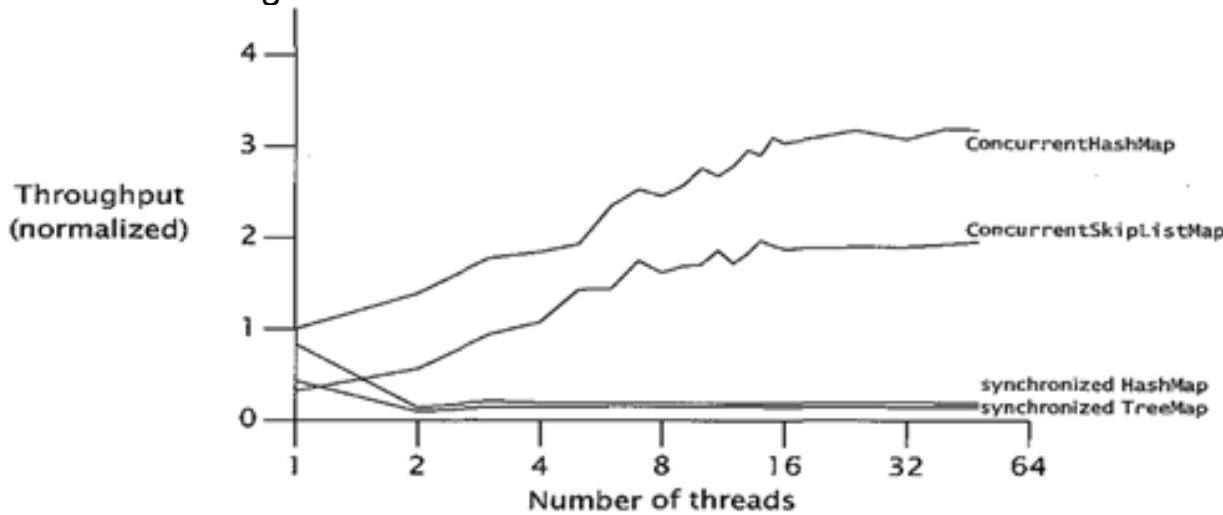
- Instead of a collection wide lock, the collection maintains a list of locks.
- Each lock is used to guard a bucket of the collection.
- For example, for map there are 16 locks and 16 threads can modify the collection at a time.
- In most cases, no collection-wide lock is required to perform operations: retrieval, removal, traversal and iteration.
- Many reading threads can access the map concurrently.
- Higher throughput under concurrent access, with little performance penalty for single threaded access.
 - Serialize accesses to the collection's state.
 - Poor concurrency when multiple threads contend for the collection wide lock.

java.util.concurrent

Concurrent collections-examples



- **ConcurrentHashMap**
 - Thread safe implementation of the java.util.Map interface
 - Does not use Synchronized
 - Allows unlimited parallel read operations without locks
 - For write access, data structure internally divided into segments, which locked when updating for read and write access of another threads.
 - The segments can be resized.



Test scenario:
For N threads concurrently execute a loop that chooses a random key and looks up value corresponding to that key. If the value is found, it is removed with a probability of 0.02. If not, it is added to the map with a probability of 0.6.

java.util.concurrent



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Providing useful functionality is needed.
- Categories
 - Synchronizer
 - Atomic data types
 - Collections
 - **Constructs for asynchronous execution**
 - **Callable**
 - **Future**
 - **Executor**

java.util.concurrent

Constructs for asynchronous execution-Callable



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Similar to Runnable interface but it can return any Object and able to throw Exception
- The callee thread implements the callable interface (i.e. the call method instead of run)
 - Object that implements callable, represents a task that returns a result and may throw an exception
- The calling thread passes an object to an executor using the submit() method and then resumes its execution.
- **Submit** returns a future object
 - Caller read result from future object using get()

```
public interface Callable<V> {  
    public V call() throws Exception;  
}
```

java.util.concurrent

Constructs for asynchronous execution-Future



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Represents result of calculation
- Available immediately or in the future
- `get()`: result of calculation or blocks until result is available.
- `cancel()`: cancels thread execution.

```
public interface Future<V> {  
    public V get();  
    public V get(long timeout, TimeUnit unit);  
    public boolean cancel(boolean mayInterruptIfRunning);  
    public boolean isCancelled();  
    public boolean isDone();  
}
```

java.util.concurrent

Constructs for asynchronous execution-Future (Example)



```
class CallableExample implements Callable<String> {  
    public String call() {  
        String result = "job done";  
        // do something  
        return result;  
    }  
}
```

```
ExecutorService es = Executors.newSingleThreadExecutor();  
Future<String> f = es.submit(new CallableExample());  
//in the meantime, do something  
try {  
    String callableResult = f.get();  
} catch (InterruptedException ie) {...}  
catch (ExecutionException ee) {...}
```

java.util.concurrent

Constructs for asynchronous execution-Executor



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Before Executor framework in JDK 1.5: developer responsibility for thread management activities, e.g. thread creation and scheduling!
- Executor: a simple interface that supports launching new task.
 - contains the method execute
 - Tasks are stored in internal queues
- ExecutorService: a sub interface of Executor, supports future and/or periodic execution of tasks and methods such as submit, shutdown, invokeAll, ...

java.util.concurrent

Constructs for asynchronous execution-ExecutorService



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Executors: is a factory which provides various implementations of ExecutorService:
 - SingleThreadExecutor: Has exactly a worker thread
 - FixedThreadPoolExecutor:
 - Thread pool with a fixed number of threads.
 - Keeps all threads running, i.e. occupies resources, until explicitly terminated
 - CachedThreadPoolExecutor:
 - Creates a thread pool, to number of threads as needed.
 - Threads that have not been used for 60 seconds are terminated and removed from the cache.
 - Improves performance of programs that execute many **short-lived** asynchronous tasks
 - ScheduledThreadPoolExecutor:
 - Enables periodic execution of tasks or with certain delay.

java.util.concurrent

ExecutorService-ForkJoinPool



- Different from other kinds of ExecutorService
 - It employs work-stealing: All threads find and execute tasks submitted to the pool.
 - Efficient when tasks spawn other subtasks
 - The pool maintains number of active threads dynamically by adding, suspending, or resuming internal worker threads
 - Designed for work that can be broken into smaller pieces recursively.
 - Your code should look similar to:

```
if (my portion of is smaller than sequential threshold)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

java.util.concurrent

Executors-example



```
public class WorkerThread implements Runnable {  
    private String command;  
    public WorkerThread(String s){  
        this.command=s;  
    }  
  
    @Override  
    public void run() {  
  
        System.out.println(Thread.currentThread().getName()  
        ()+' Start. Command = '+command);  
        processCommand();  
  
        System.out.println(Thread.currentThread().getName()  
        ()+' End.');
```

}

```
    }  
  
    private void processCommand() {  
        try {  
            Thread.sleep(5000); //Pause 5 seconds  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    @Override  
    public String toString(){  
        return this.command;  
    }  
}
```

```
public class SimpleThreadPool {  
  
    public static void main(String[] args) {  
        ExecutorService executor =  
        Executors.newFixedThreadPool(5); 1  
        for (int i = 0; i < 10; i++) { 2  
            Runnable worker = new WorkerThread('' +  
            i);  
            executor.execute(worker); 3, 4  
        }  
        executor.shutdown();  
        while (!executor.isTerminated()) {  
        }  
        System.out.println('Finished all threads');  
    }  
}
```

1. Pool of 5 worker threads
2. Submitting 10 jobs to this pool
3. Start 5 jobs and other jobs will be in wait state
4. When a job finishes, another one is started from the wait queue

java.util.concurrent

Which programming style should I use?



```
class Main {  
    int counter = 0; int coords[DATAN];  
    void main() {  
        for (int i = 0; i < THREADN; i++)  
            (new Bucket()).start();  
    }  
    class Bucket extends Thread {  
        ...  
        public void run() { while(counter<DATAN)  
        { dowork  
        () ; }}  
        public void dowork() {  
            int start;  
            synchronized (Main.this) {  
                if (counter >= DATAN) return;  
                start=counter; counter+=DATAN/TASKN;  
            }  
            for (int j = 0; j < DATAN/TASKN; j++) {  
                render(coords[start + j]);  
            } //end for  
        }}}
```

Thread Style

Executor Style

ForkJoin Style

G. Pinto, F. Castor, Y. D. Liu "Understanding energy behaviors of thread management constructs"

```
class Main {  
    int counter = 0; int coords[DATAN];  
    void main() {  
        ExecutorService es = Executors.  
        newFixedThreadPool(THREADN);  
        for (int i = 0; i < TASKN; i++)  
            es.execute(new Bucket()); }  
    class Bucket extends Thread {  
        ...  
        public void run() { dowork(); }  
    }}}
```

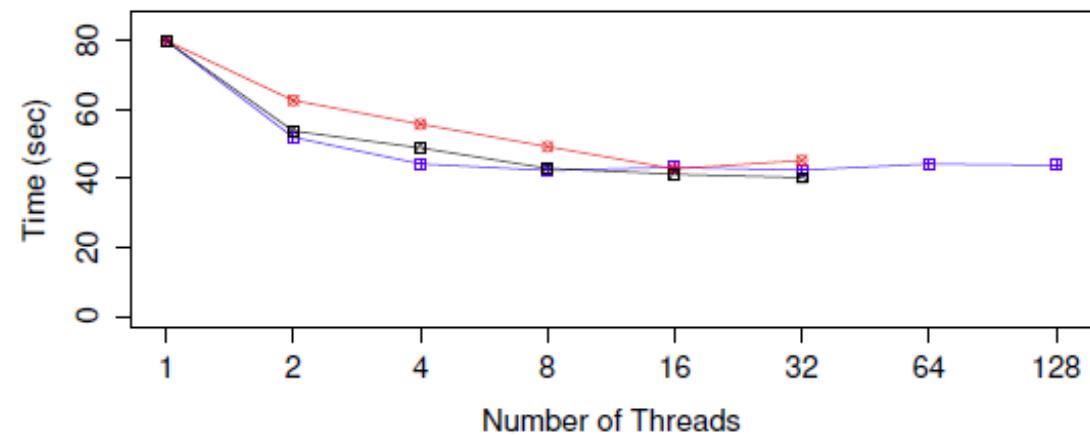
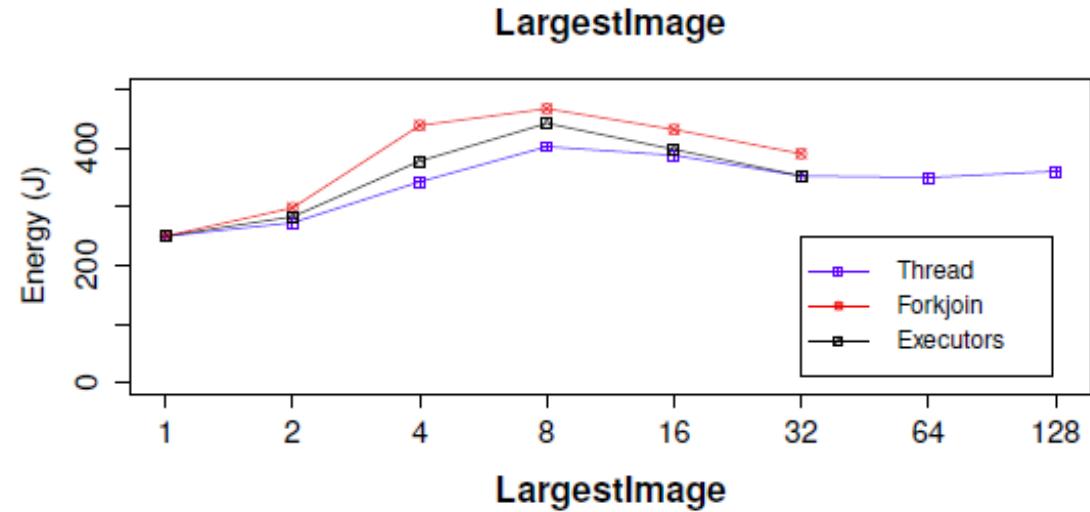
```
class Main {  
    int counter = 0; int coords[DATAN];  
    void main() {  
        (new ForkJoinPool(THREADN)).submit(new Bucket(1));  
    }  
    class Bucket extends RecursiveAction {  
        ...  
        int taskcounter;  
        Bucket(int taskcounter) {  
            this.taskcounter = taskcounter;  
        }  
        public void compute() {  
            if (taskcounter <= TASKN) {  
                (new Bucket(taskcounter+1)).fork();  
                dowork();  
            }}}}
```



- Thread management constructs consume energy differently:
 - I/O-bound programs: Thread style is best, ForkJoin style is worst.
 - Embarrassingly parallel: the opposite holds.

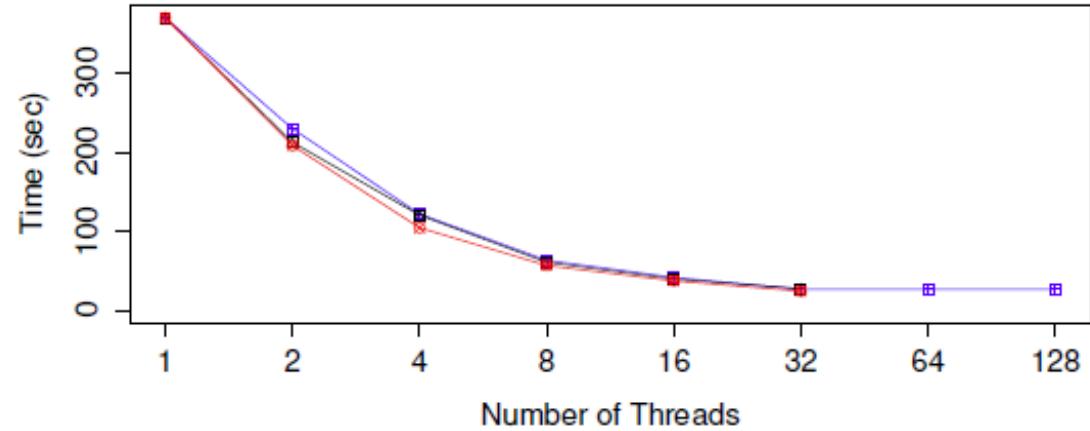
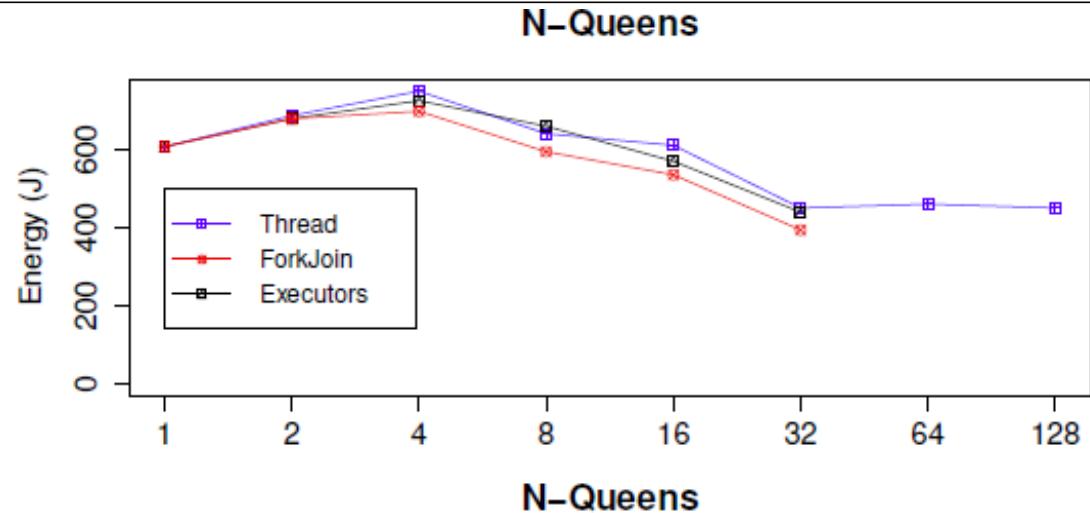
LargestImage:

- DaCapo benchmark suite
- I/O intensive
- recursive file system search.



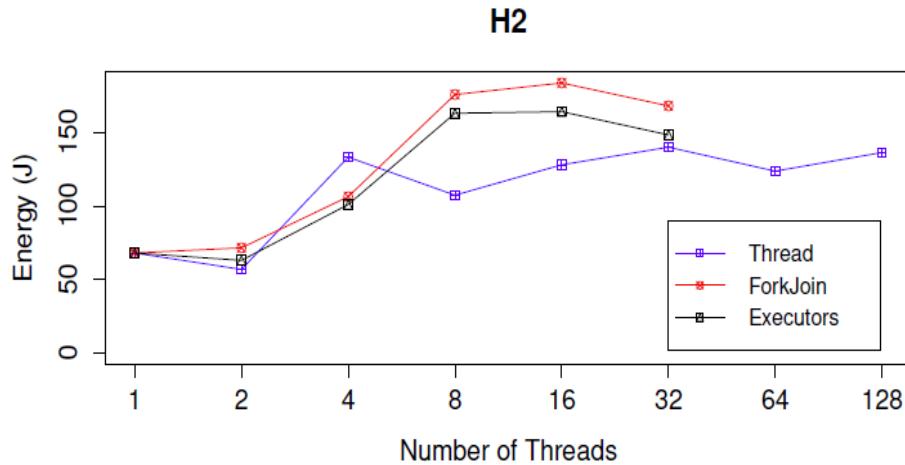
N-queens:

- RAW benchmark suite
- CPU-intensive
- no synchronization points but uses one atomic variable



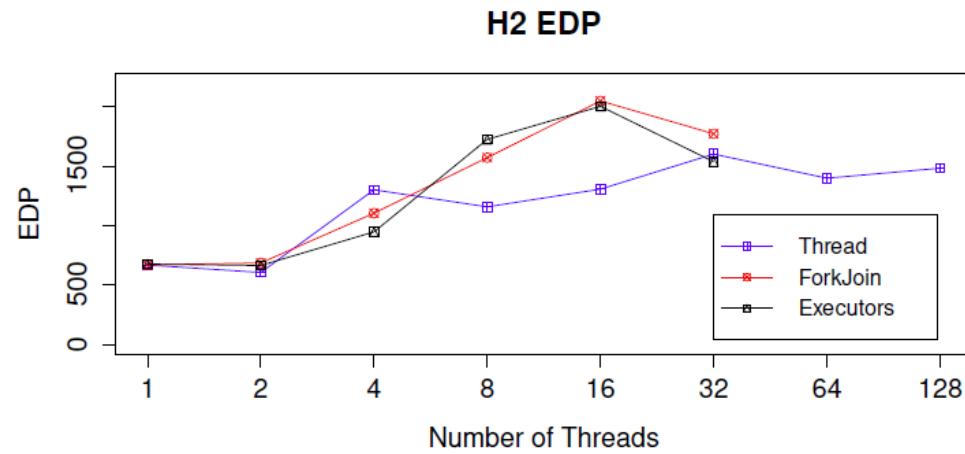
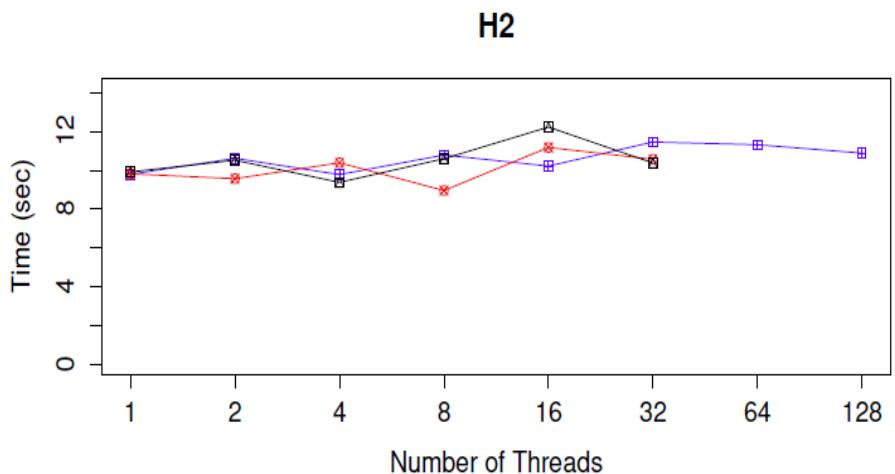


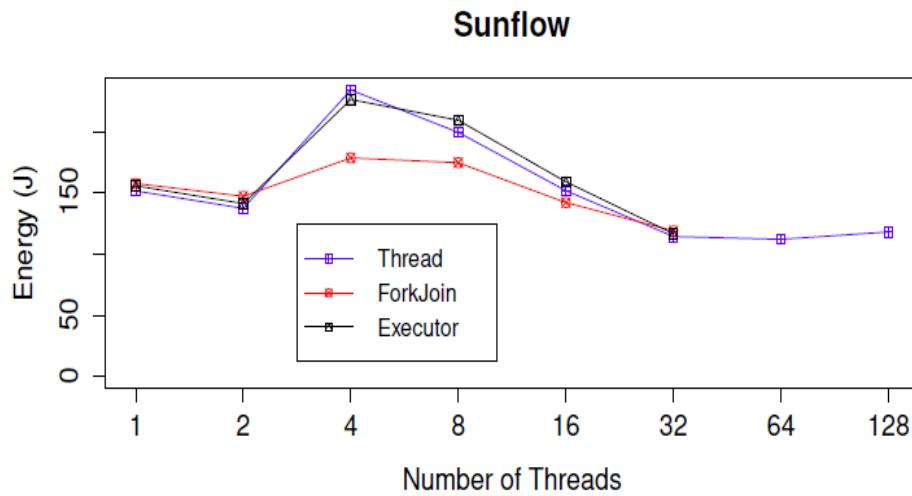
- Faster is not greener:
 - Sequential execution often consumes least energy
 - Parallel execution leads to improved energy/performance trade-off for nonembarrassingly-serial programs.



H2:

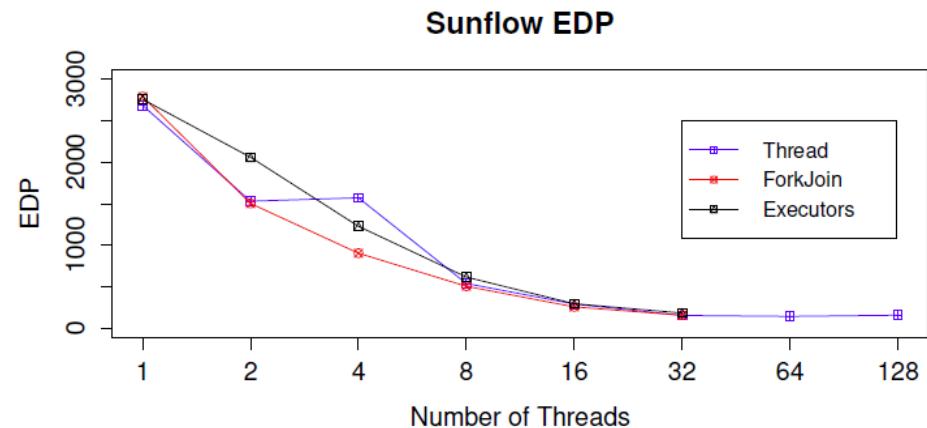
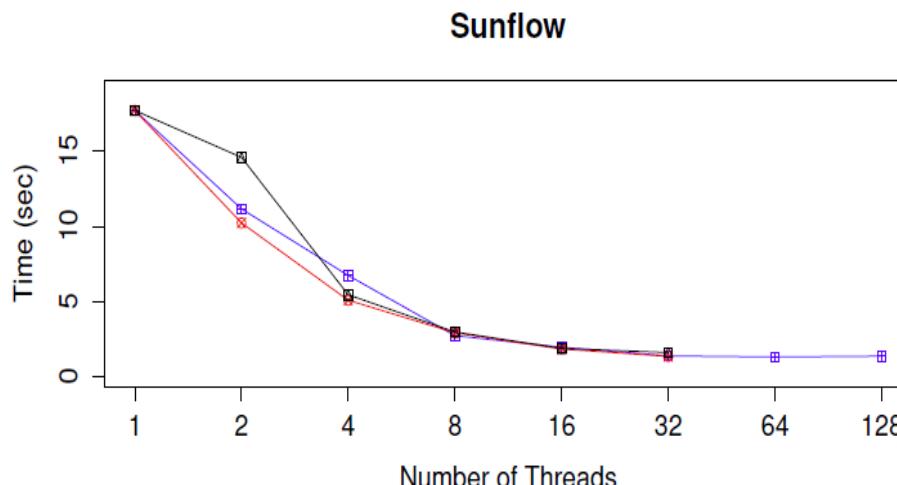
- Memory bound
- Highly sequential
- Not considerable benefit when parallelized but consumes much energy!
- Better to run it sequentially
- No an interesting EDP





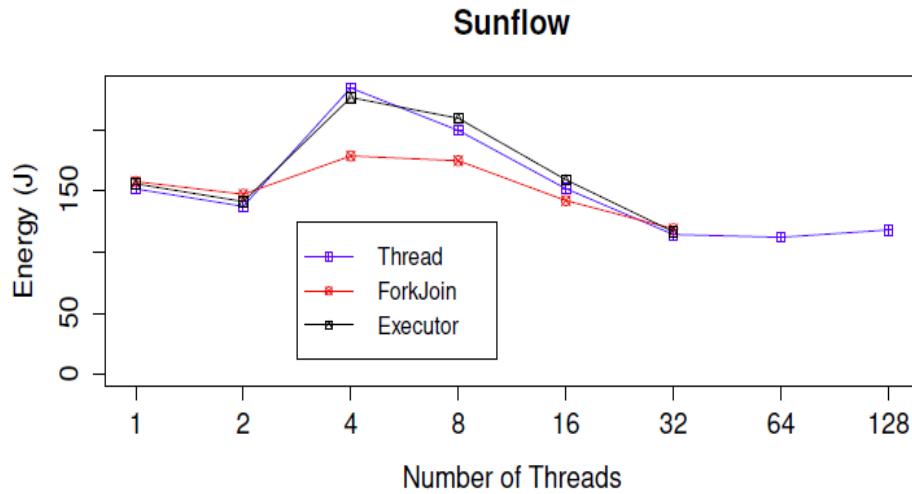
Sunflow:

- Renders a set of images using ray tracing
- Memory bound
- Embarrassingly parallel
- Very good EDP



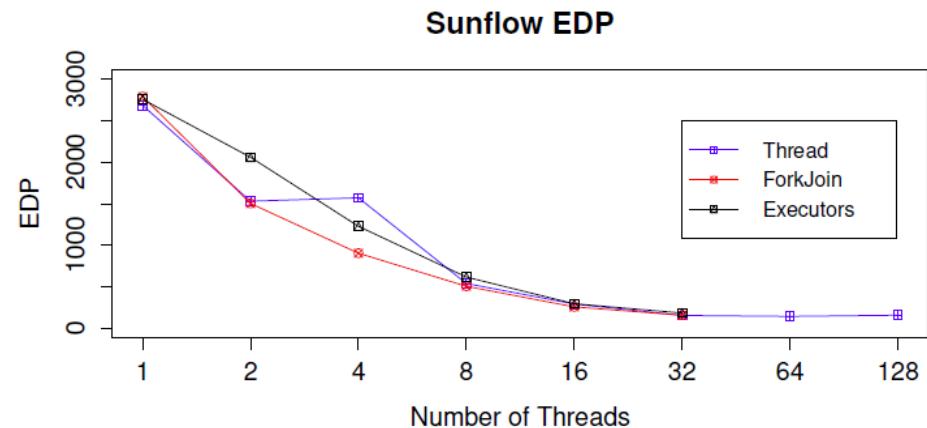
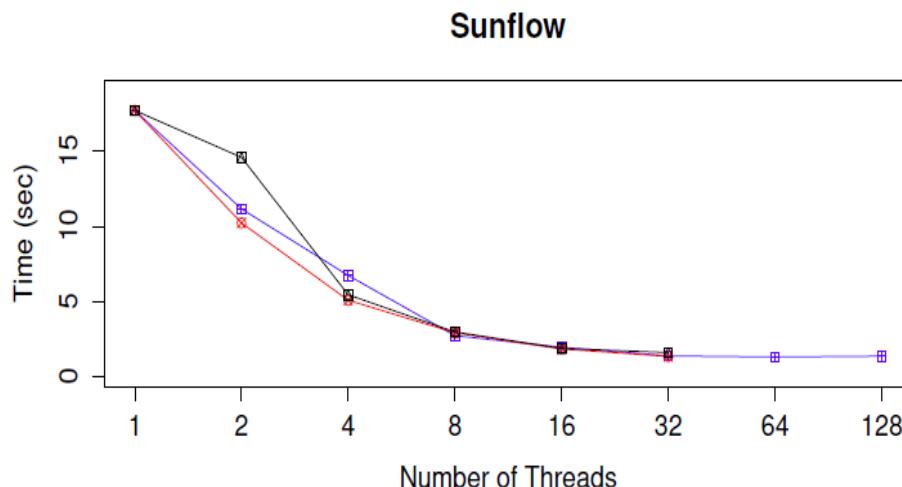


- ForkJoin style:
 - Energy consumption of ForkJoin style is sensitive to the degree of parallelism latent in the benchmark. Outperforms other style if benchmarks are embarrassingly parallel or when the benchmark needs to join during execution or after completion. Underperforms in presence of more serial benchmarks.



Sunflow:

- Renders a set of images using ray tracing
- Memory bound
- Embarrassingly parallel
- Very good EDP



java.util.concurrent

- There are lots of synchronization primitives, concurrent data structures, concurrency programming styles etc.
- Different impact on energy consumption and execution time.
- Which one to use?



Parallel patterns



- Consider the opposite way!
- Find pattern in your algorithm
- Implementation comes next

Pattern	# projects
Master-worker	29
Producer-consumer	10
Pipeline	8
Divide&conquer	3
Parallel Loop	1

An empirical study on parallelism in modern open-source projects, M. Kiefer, D. Warzel, W. Tichy, SEPS 2015.

Parallel patterns

Task parallelism



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- We have already talked about patterns in details.
- Concurrency in tasks:
 - Divide-and-conquer:
 - Embarassingly parallel
 - Tasks join during execution or after completion
 - ForkJoinPool
 - Master-worker: One thread, called the master thread creates a bunch of worker threads.
 - Embarassingly parallel
 - Some dependencies
 - ExecutorService

Parallel patterns

Pipeline or producer-consumer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Concurrency in flow of data:
 - Produce-consumer:
 - Threads exchange data
 - Exchanger
 - Pipeline:
 - More than two threads communicating
 - Exchanger
 - Threads synchronize based on events:
 - CyclicBarriers or CountDownLatch

Java memory model



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Memory model
- The fundamentals: happens-before ordering
- Using volatile
- Recommendations

Memory model



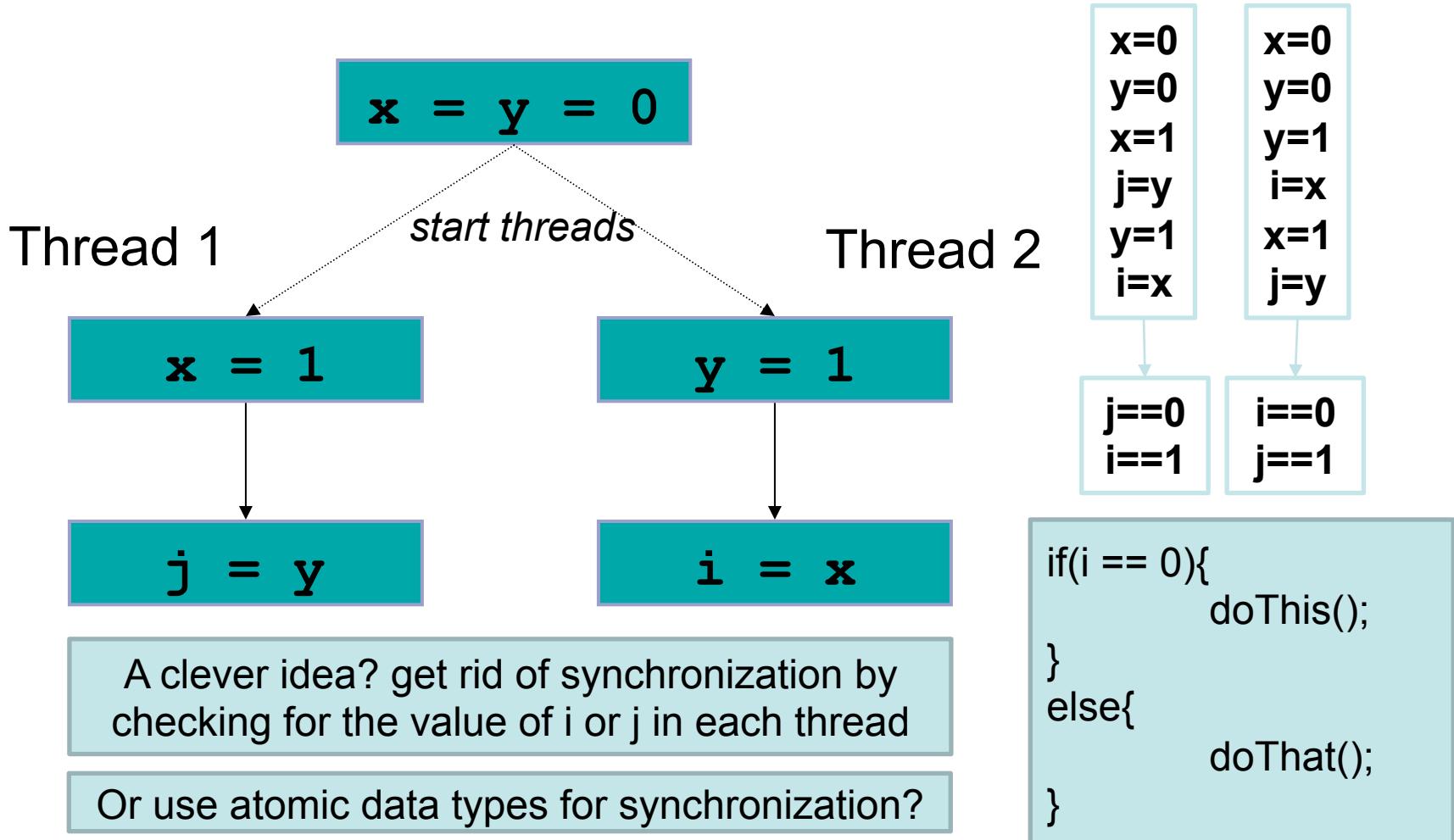
- Given a program and an execution trace of that program
 - It determines whether the execution trace is a legal execution of the program
- describes possible behaviors of a program
- provides the freedom to perform a myriad of code transformations:
 - reordering of actions
 - removal of unnecessary synchronization
- Free to implement the program in any way as long as the result of the execution is predictable

Memory model

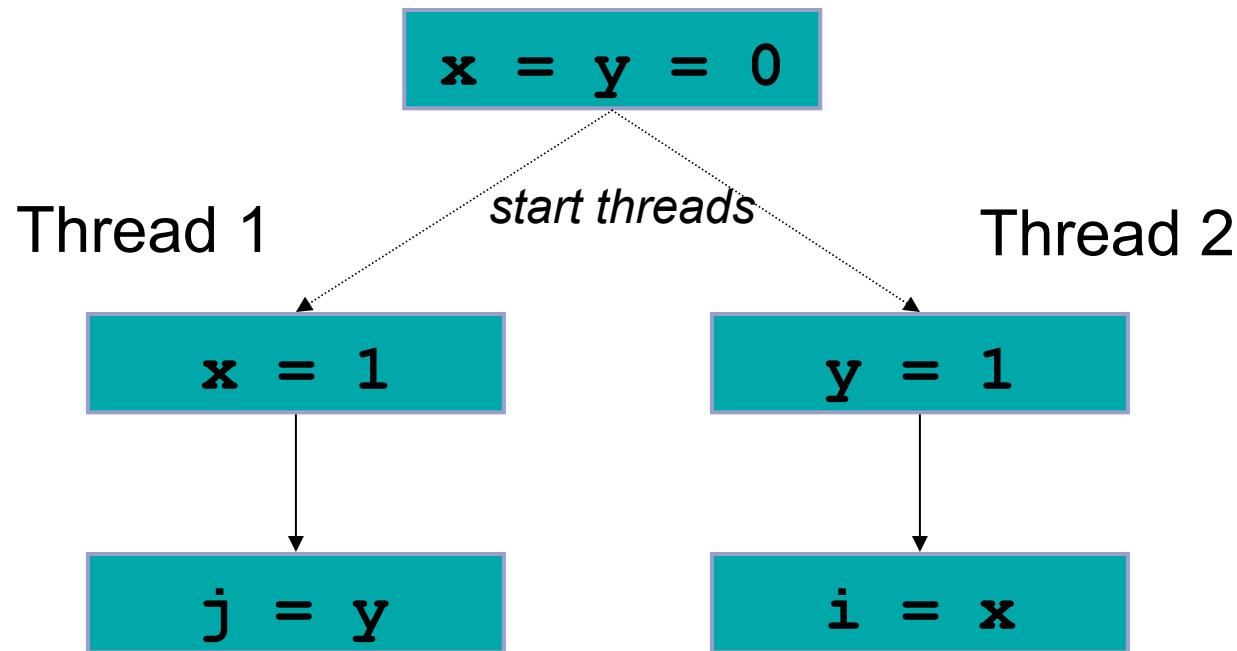


- Simple rules for single-thread execution
 - Optimize the code as long as as-if-serial semantics are not violated
 - As-if-serial semantic: result of thread execution is guaranteed to be the same as it executed statements in the order that appear in the program
- Parallel threads may see different values for the same shared data!
 - In the absence of synchronization
 - Different optimizations: reordering

Memory model

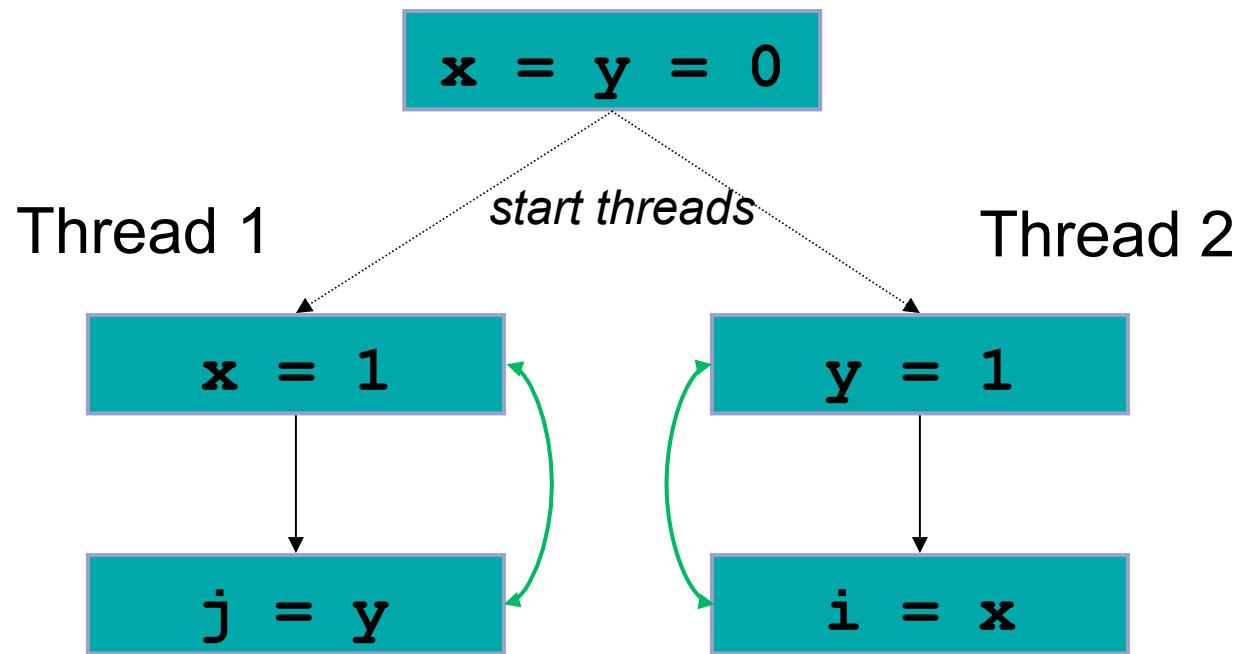


Memory model



Can this result in $i = 0$ and $j = 0$? YES!

Memory model



How can $i = 0$ and $j = 0$?

How can this happen?

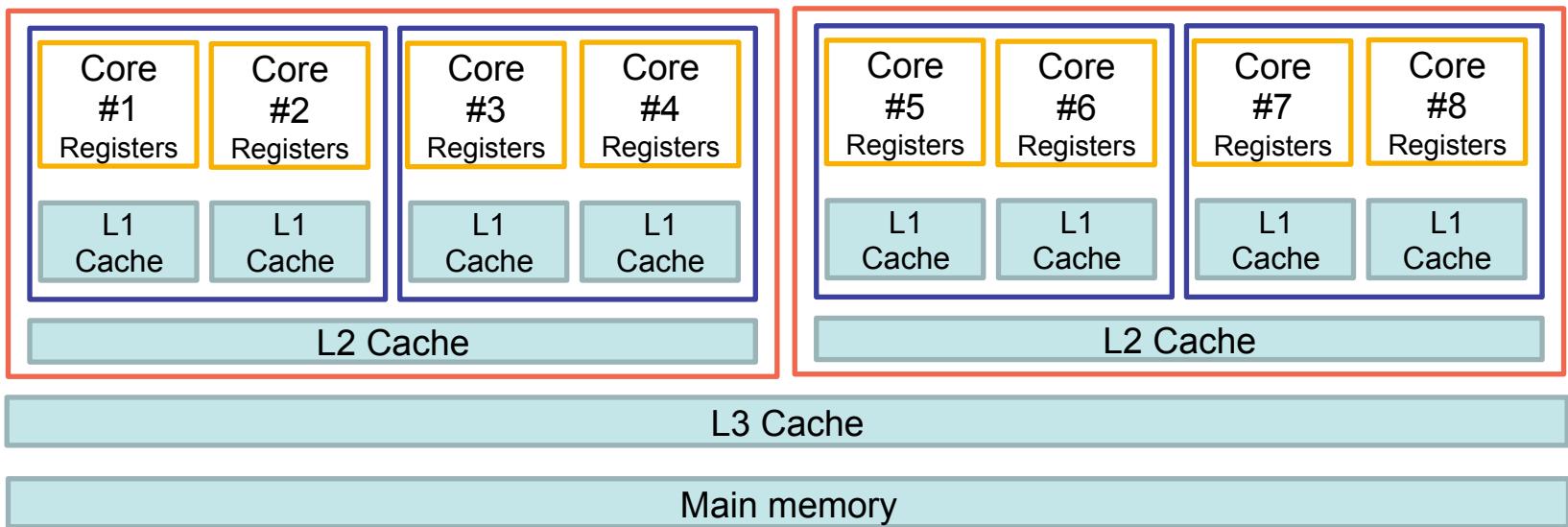


- Compiler can reorder statements
 - Or keep values in registers
- Processor can reorder them
 - On multi-processor, values not synchronized to global memory
- Memory model is designed to allow aggressive optimization
 - including optimizations no one has implemented yet
 - Good for performance
 - Bad for your intuition about insufficiently synchronized code

Memory hierarchy



- Parallel threads may see different values for the same shared data!
 - Cores have their own local caches
 - Generally undesirable to maintain threads perfectly in sync
 - Performance greatly degrades!





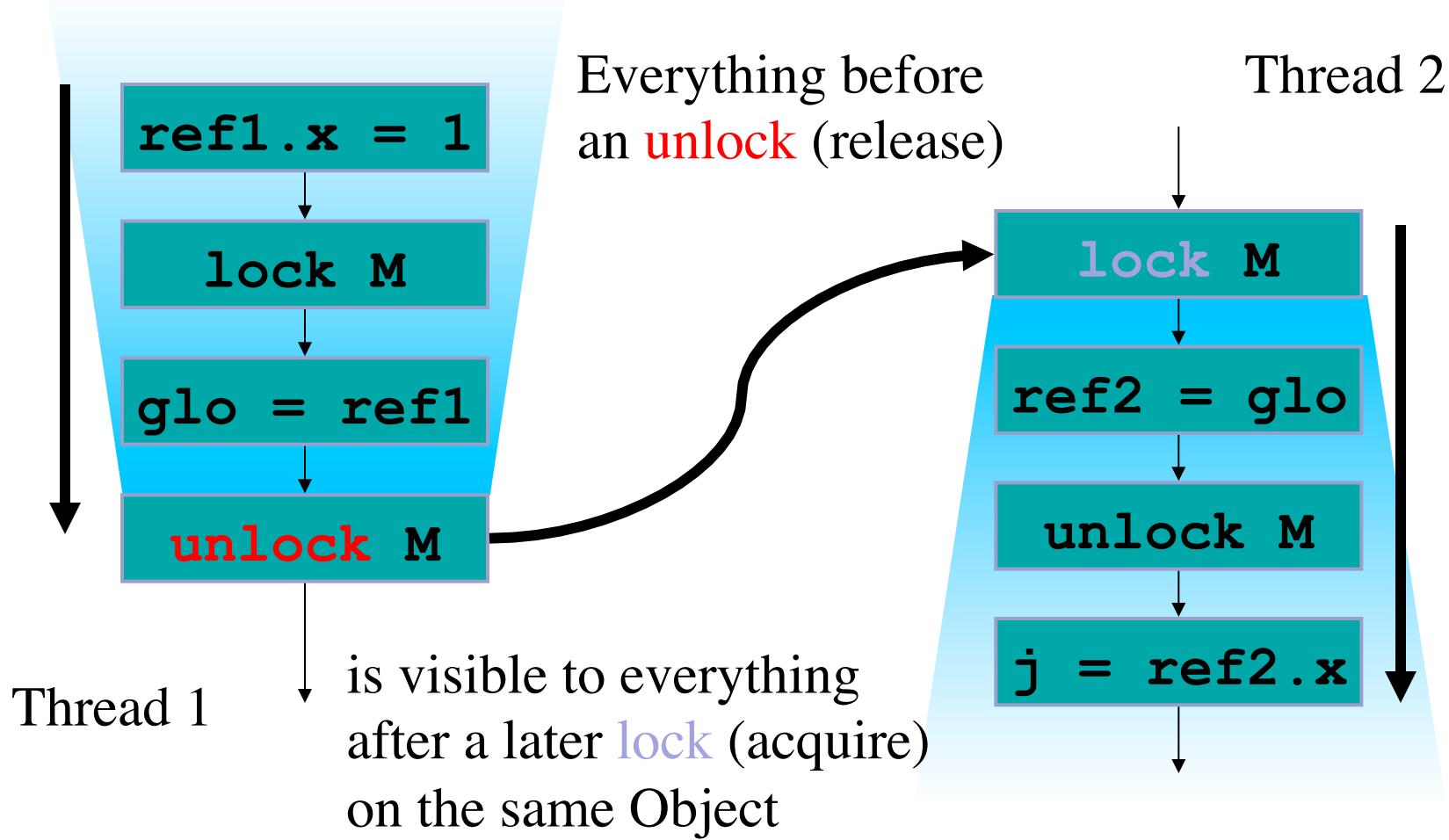
Memory model

The fundamentals: happens-before ordering

Using volatile

Recommendations

Synchronization, blocking and visibility





Happens-before rules

- Program order: Actions in threads follow program order
- Lock: An unlock on a lock happens-before every subsequent lock on the same lock.
- Thread start: Call to `Thread.start` happens-before every action in the started thread.
- Thread termination: Termination of a thread happens-before a join with the terminated thread
- Transitivity: If A happens before B, and B happens-before C, then A happens before C.
- `util.concurrent`: Placing an object into any concurrent collection happens-before the access or removal of that element from the collection



Memory model

The fundamentals: happens-before ordering

Using volatile

Recommendations

Volatile fields



- If a field could be simultaneously accessed by multiple threads, and at least one of those accesses is a write
- Two choices:
 - use synchronization
 - make the field volatile
 - gives essential JVM machine guarantees
- What does volatile do?
 - reads and writes go directly to memory
 - not cached in registers
 - volatile reads/writes cannot be reordered

Happens-before rule for volatile

- A volatile write happens-before all following reads of the same variable
- A volatile write is similar to a unlock
- A volatile read is similar to a lock

More notes on volatile



- Incrementing a volatile is not atomic
 - Increment is a compound action: read/modify/write
 - if threads try to increment a volatile at the same time, an update might get lost
- Consider using `util.concurrent.atomic` package
 - Atomic objects work like volatile fields
 - but support atomic operations such as increment and compare and swap
- Volatile reads are very cheap
 - volatile writes cheaper than synchronization
- No way to make elements of an array be volatile



Scope

The fundamentals: happens-before ordering

Using volatile

Recommendations

These are building blocks

- If you can solve your problems using the high level concurrency abstractions provided by `util.concurrent`
 - do so
- Performance of the `util.concurrent` abstractions is amazing and getting better
- Understanding the memory model, and what release/acquire means in that context, can help you devise and implement your own concurrency abstractions
 - and learn what not to do

Designing Fast Concurrent Code



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Make it right before you make it fast
- Reduce synchronization costs
 - avoid old Collection classes (`Vector`, `Hashtable`)
- Use `java.util.concurrent` classes
- Avoid lock contention
 - Reduce lock durations



Software Engineering for Multicore Systems

Dr. Ali Jannesari

SOFTWARE ENGINEERING & CONCURRENCY PATTERNS

Outline



- Software engineering vs. computational efficiency
- Concurrency Patterns
 - Active Object
 - Monitor Object
 - Half-Sync/Half-Async
 - Leader/Follower
 - Thread Local Storage

Software engineering (SE) review in a glance



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Def: Software engineering is a profession dedicated to designing, implementing, and modifying software so that it fulfills:

- Flexibility
- Extensibility
- Usability
- Modularity
- Maintainability
- Affordability

Computational Efficiency vs. High Level SE



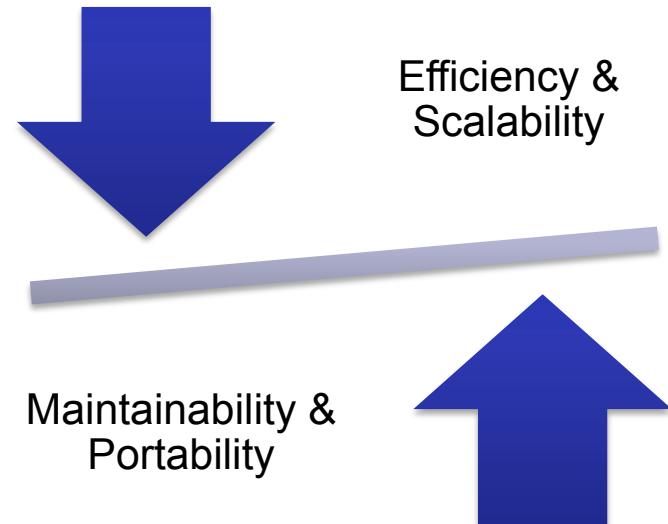
- Computational efficiency and high level SE metrics are in direct contrast
- Trade-off between SE metrics and computational efficiency goal

Efficiency Goals	Software Engineering Goals
Low level programming	High level programming
No high level pattern	OOP
Top goal: Higher performance	Top goal: Higher SE metrics

Solution?



- In multicore programming both SE metrics and computational efficiency should be considered together (trade-off)
- Separation of concerns:
 - Low level code
 - Using imperative programming style
 - Compute intensive code section
 - Need high amount of parallelism
 - High level code
 - Using OOP or other SE principles
 - High level functionalities that is less crucial in terms of performance



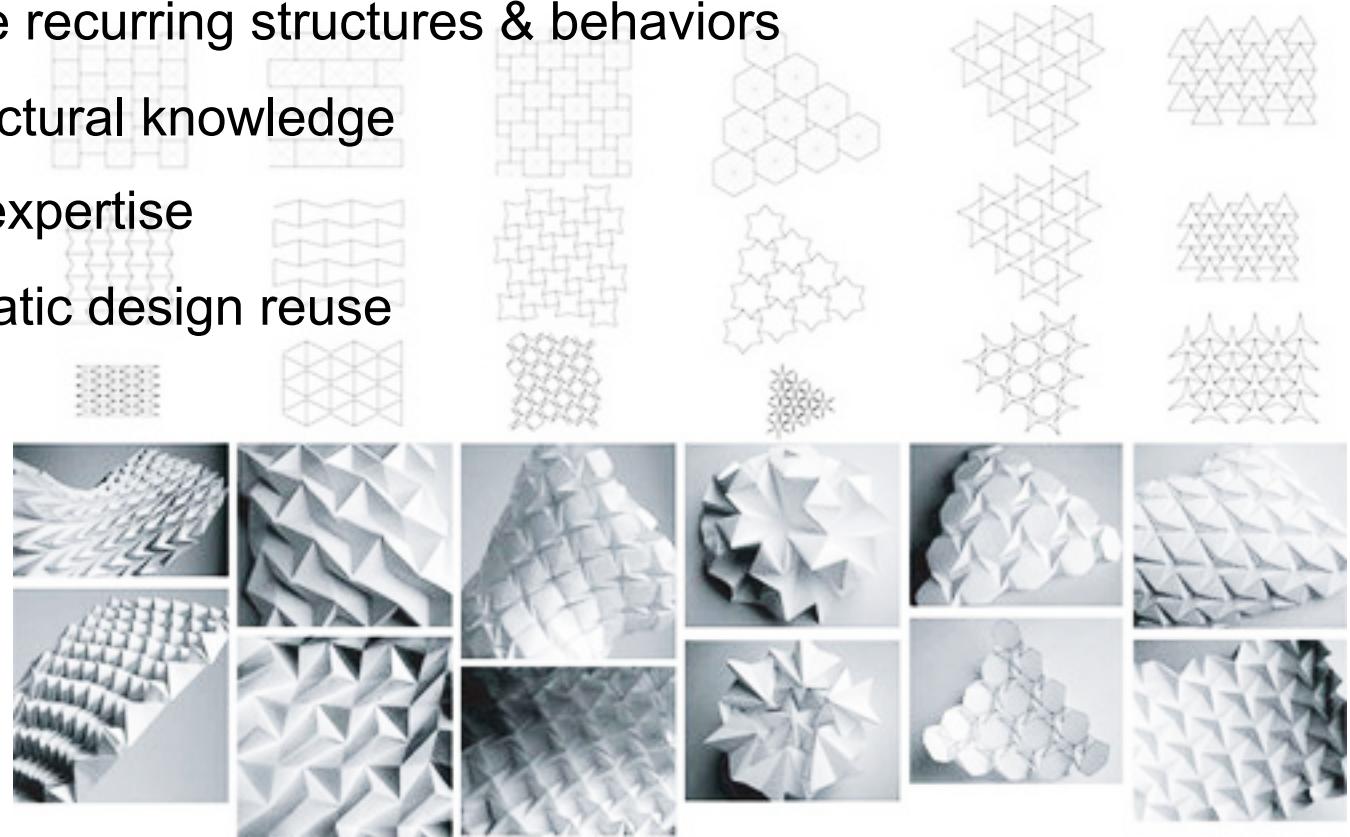
Disregarding Software Engineering?

- Low level programming is required for high performance application, but it doesn't mean that you should write the multi-threaded program by your own way
- Parallel and concurrent software development bring additional complexity (deadlock, synchronization, debugging)
- There are some patterns specific for concurrency development
- Following these guidelines can provide maximum performance along with code maintainability

Patterns Concept



- Map design problems to proven solutions
- Identify & name recurring structures & behaviors
- Convey architectural knowledge
- Codify design expertise
- Enable systematic design reuse



Different Categories of Patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Concurrency architecture patterns
 - Dealing with concurrency in mind
 - How to transform application to concurrent environments
- Parallel design patterns
 - Algorithmic view of code parallelization
 - Organization of tasks and data parallelization

Motivation for Concurrent Software



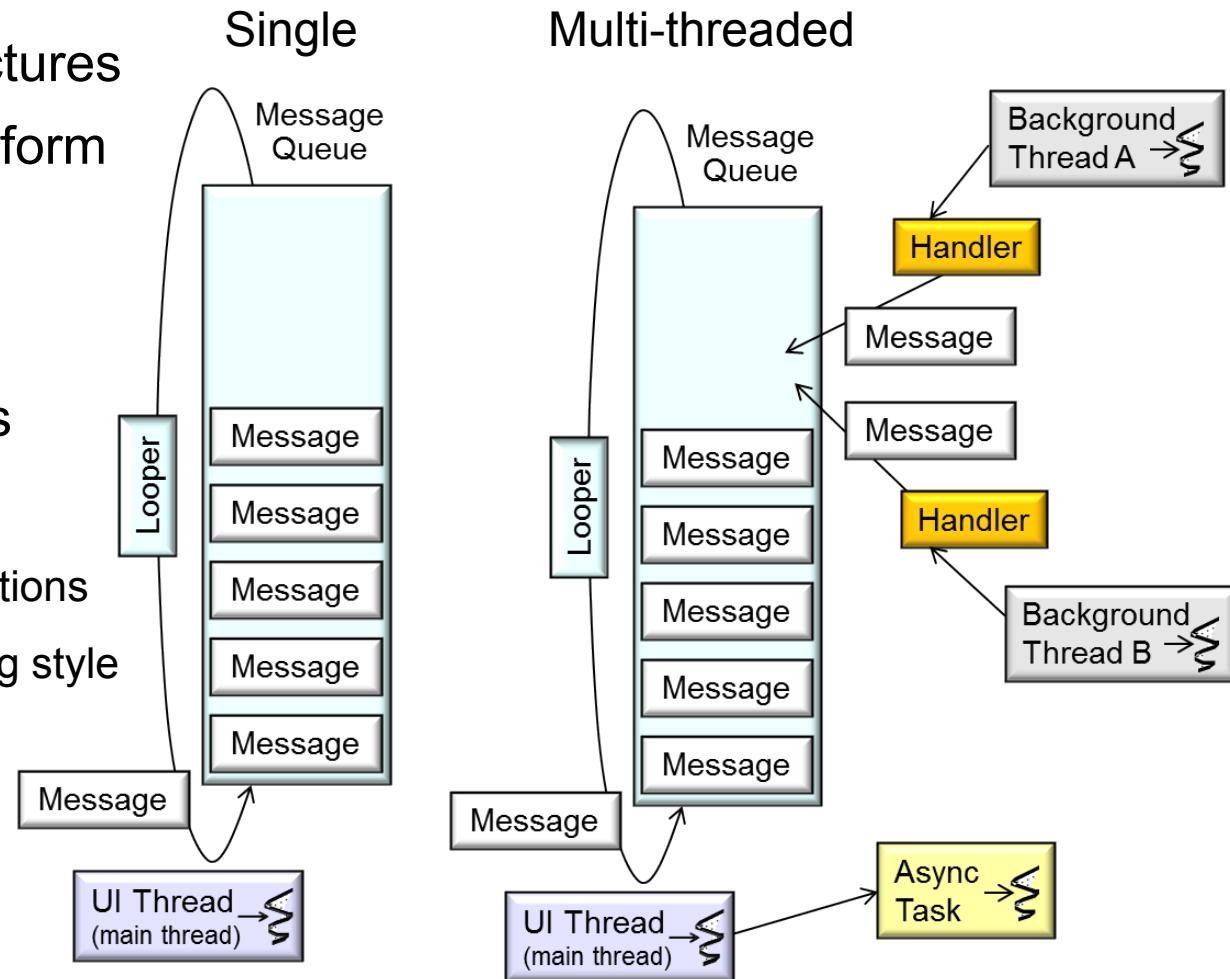
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Leverage hardware/software advances
 - e.g., multi-core processors & multi-threaded operating systems, virtual machines, & middleware
- Simplify program structure
 - e.g. allowing blocking operations
- Increase performance
 - Parallelize computations & communications
- Improve response-time
 - e.g., don't starve the UI thread

Motivation for Concurrent Software – Example



- Classic single architectures architectures can't perform blocking operations
- Concurrency simplifies program structure
 - Allowing blocking operations
 - Simplifying programming style



Concurrency Patterns



- Five important patterns that address concurrency architecture and design issue for components, subsystem and applications
- Choice of concurrency architecture
 - Significant impact on the design and performance of multi-threaded applications
 - No single concurrency architecture is suitable for all workload conditions, software and hardware platform
- Concurrent patterns could be categorized into different group of applicability:

Sharing resources	High-level concurrency architecture	Certain inherent complexities
Active Object	Half-sync/Half-Async	Thread-Specific Storage
Monitor Object	Leader/Follower	

Concurrency Patterns

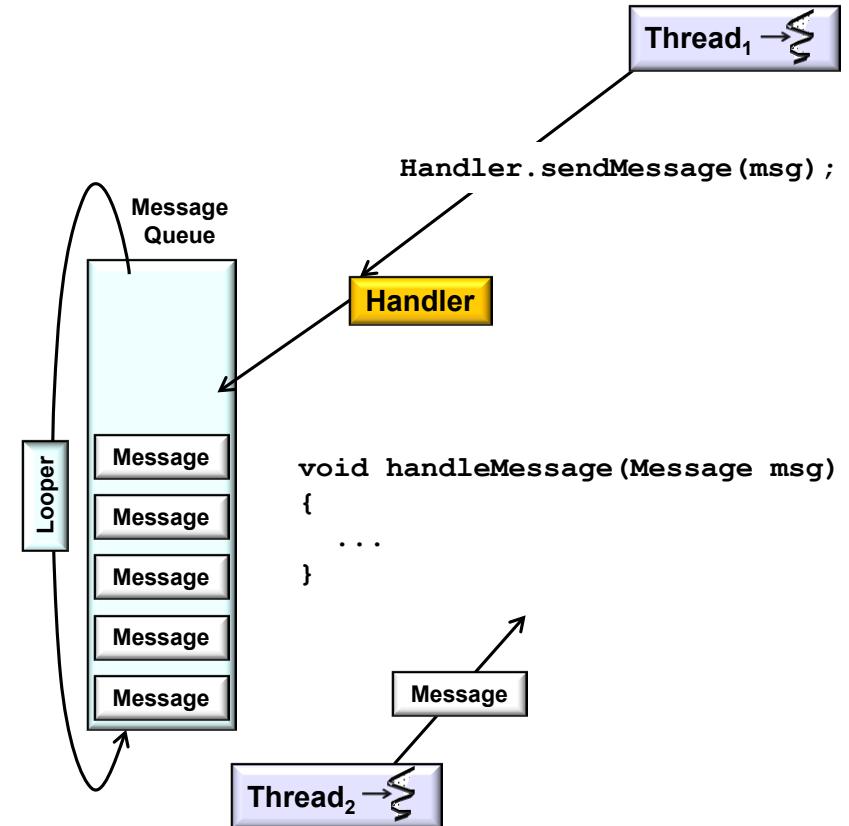
Active Object Pattern



Challenge: Invoking methods in another thread

- **Context**

- Client threads that access object's methods running in separate threads of control
- e.g. A background thread invoking sendMessage() on a Handler associated with UI thread
- Generally, any thread that interact via Handlers/Messages



Concurrency Patterns

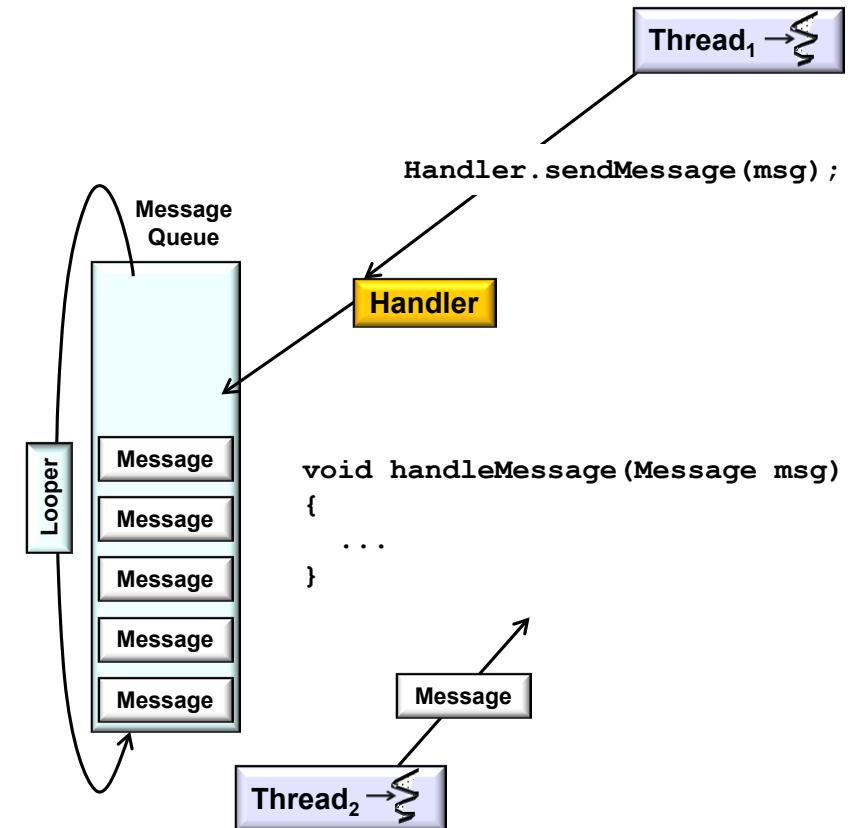
Active Object Pattern (cont.)



Challenge: Invoking methods in another thread

- **Problems:**

- Leveraging concurrency transparently
- Ensuring that threads do not get blocked after invoking remote methods
- Making synchronized access to shared objects easy and intuitive to program



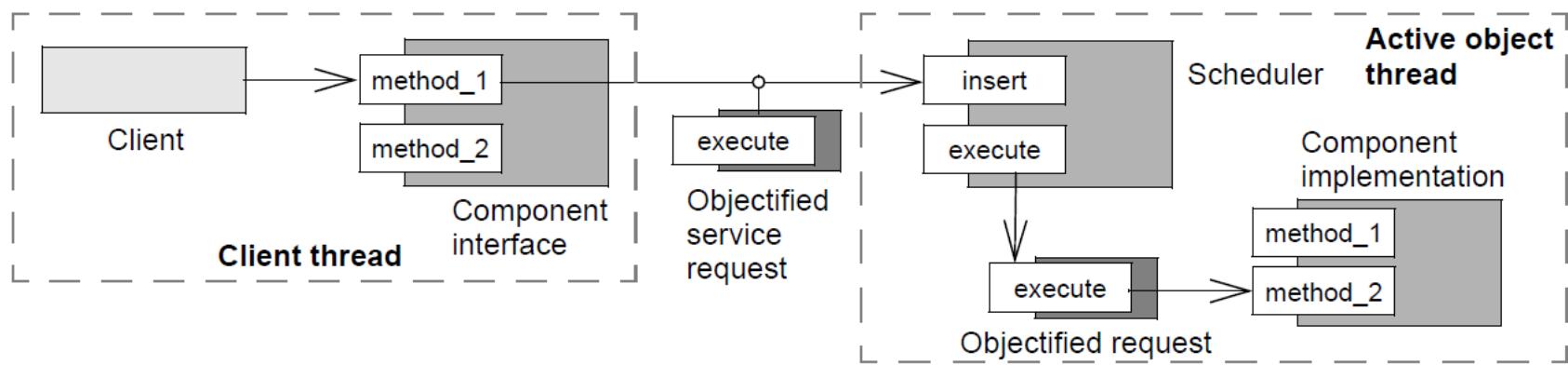
Active Object Pattern (cont.)



Challenge: Invoking methods in another thread

- **Solution: Active Object Pattern**

- Decouples method execution from its invocation (e.g. asynchronous method invocation, callbacks etc.)
- To avoid race conditions, incoming client requests are queued and handled by a scheduler
- The scheduler picks a queued object and makes it run its logic
- It is object's responsibility to know what to do when it gets invoked



Concurrency Patterns

Active Object Pattern (cont.)

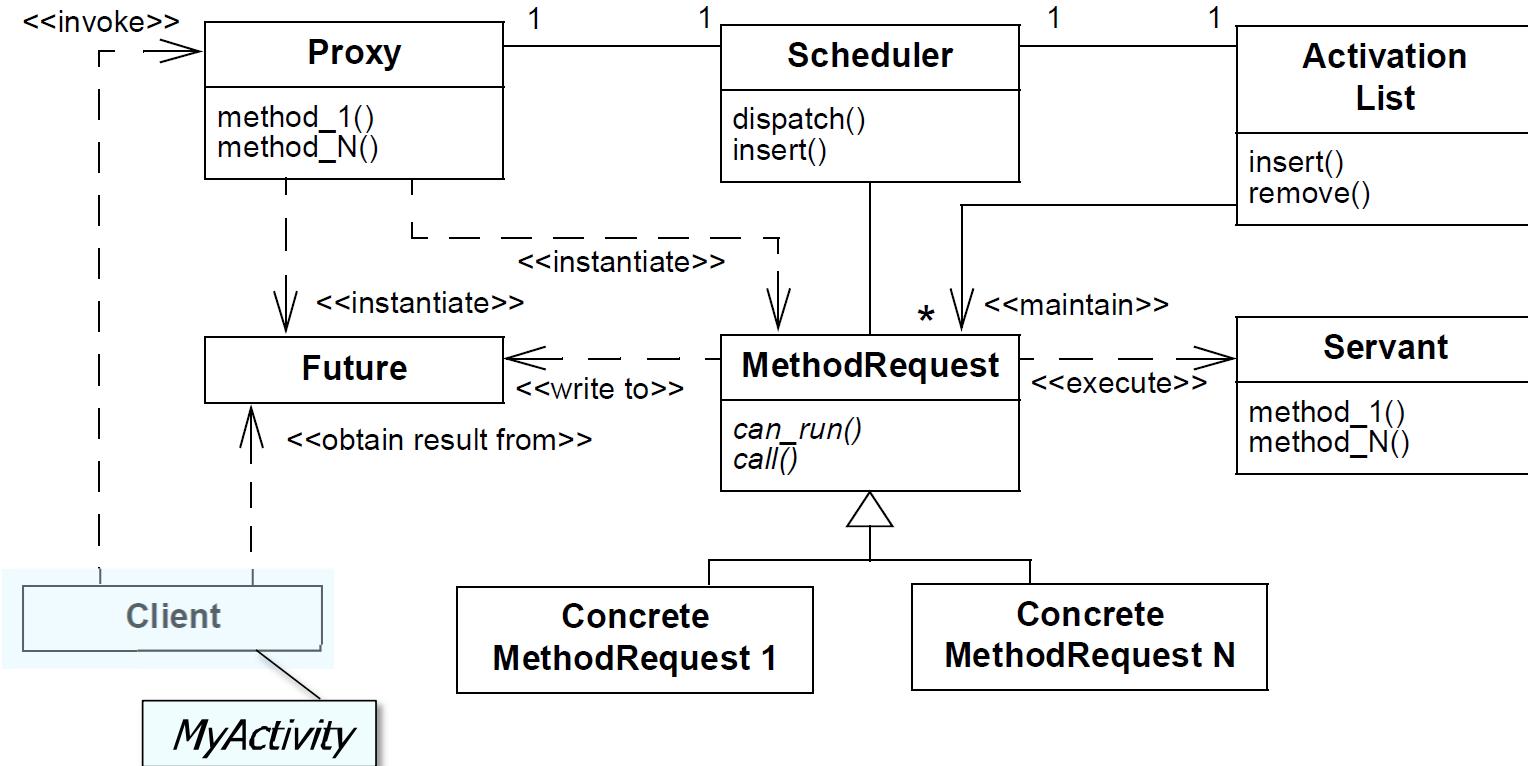


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Key Components:
 - **Proxy:** provides interface the clients can use to submit their requests
 - **Activation List:** a queue of pending client requests
 - **Scheduler:** decides which request to execute next
 - **Active Object:** implements the core business logic
 - **Callback:** contains execution result (i.e. a promise or a future)

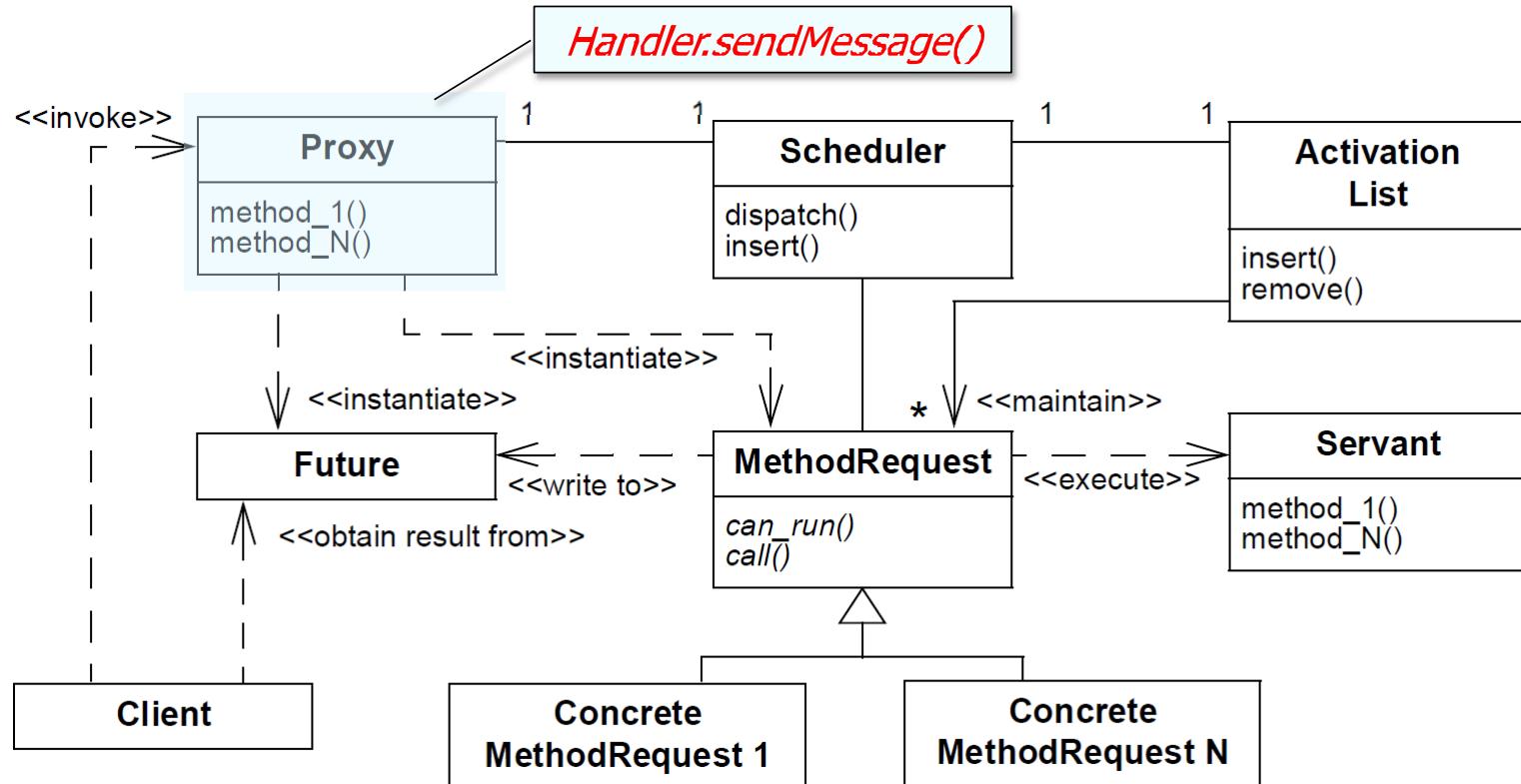
Concurrency Patterns

Active Object Pattern (cont.)



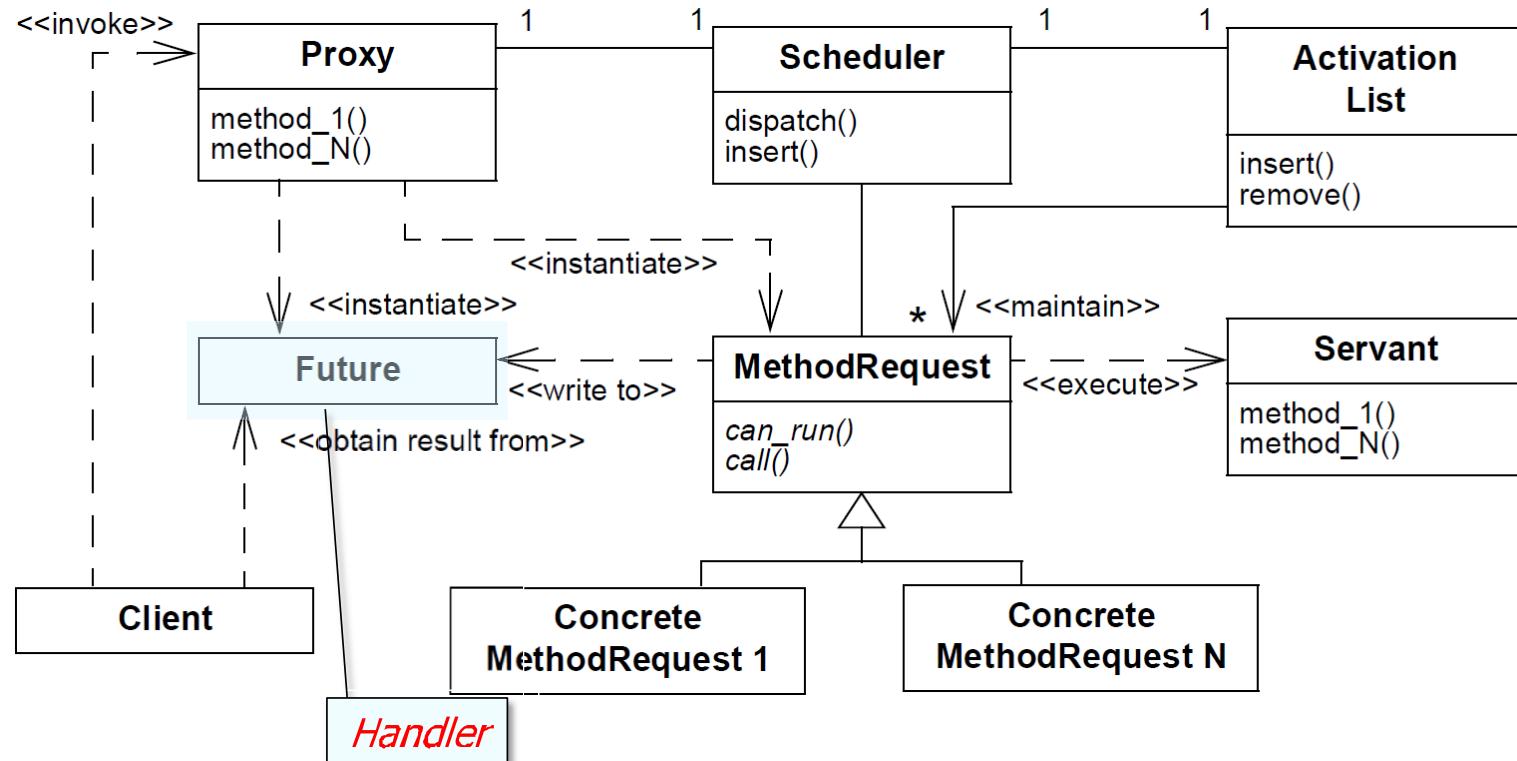
Concurrency Patterns

Active Object Pattern (cont.)



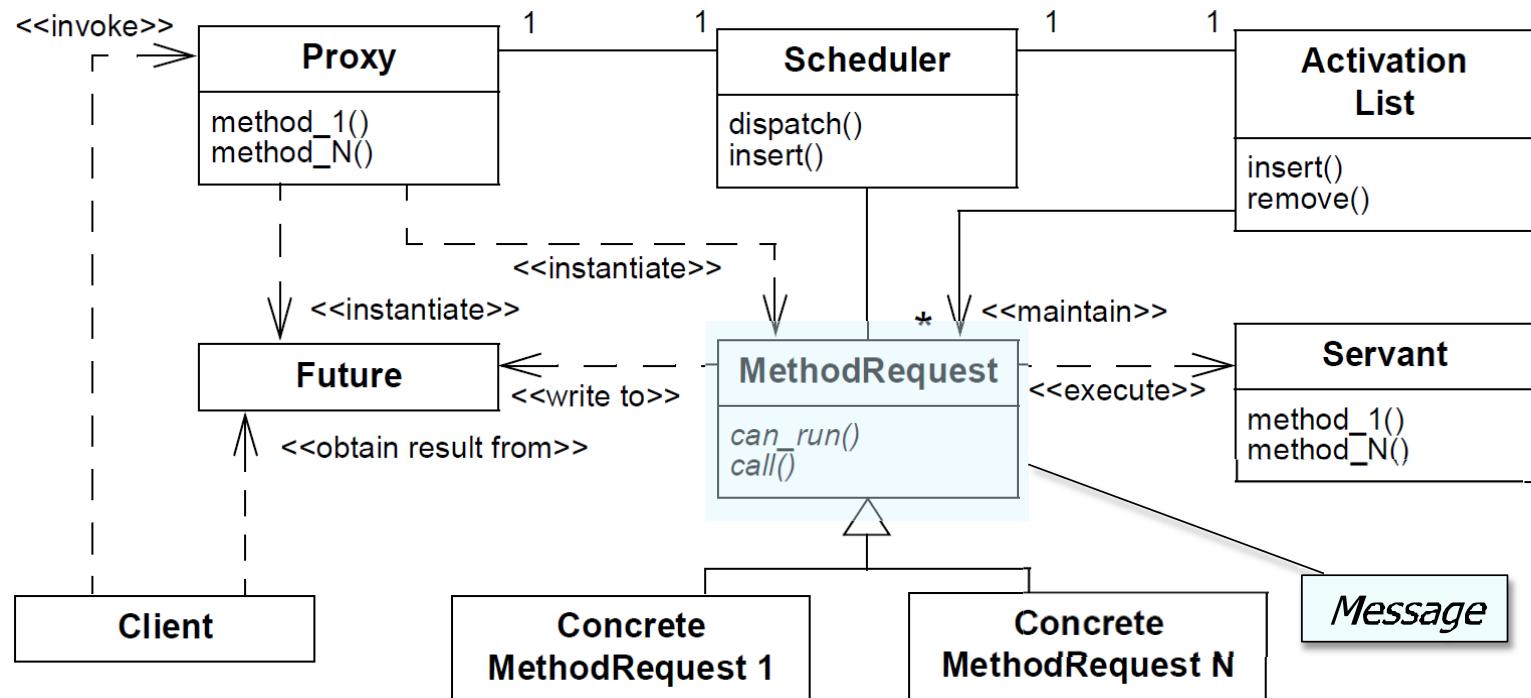
Concurrency Patterns

Active Object Pattern (cont.)



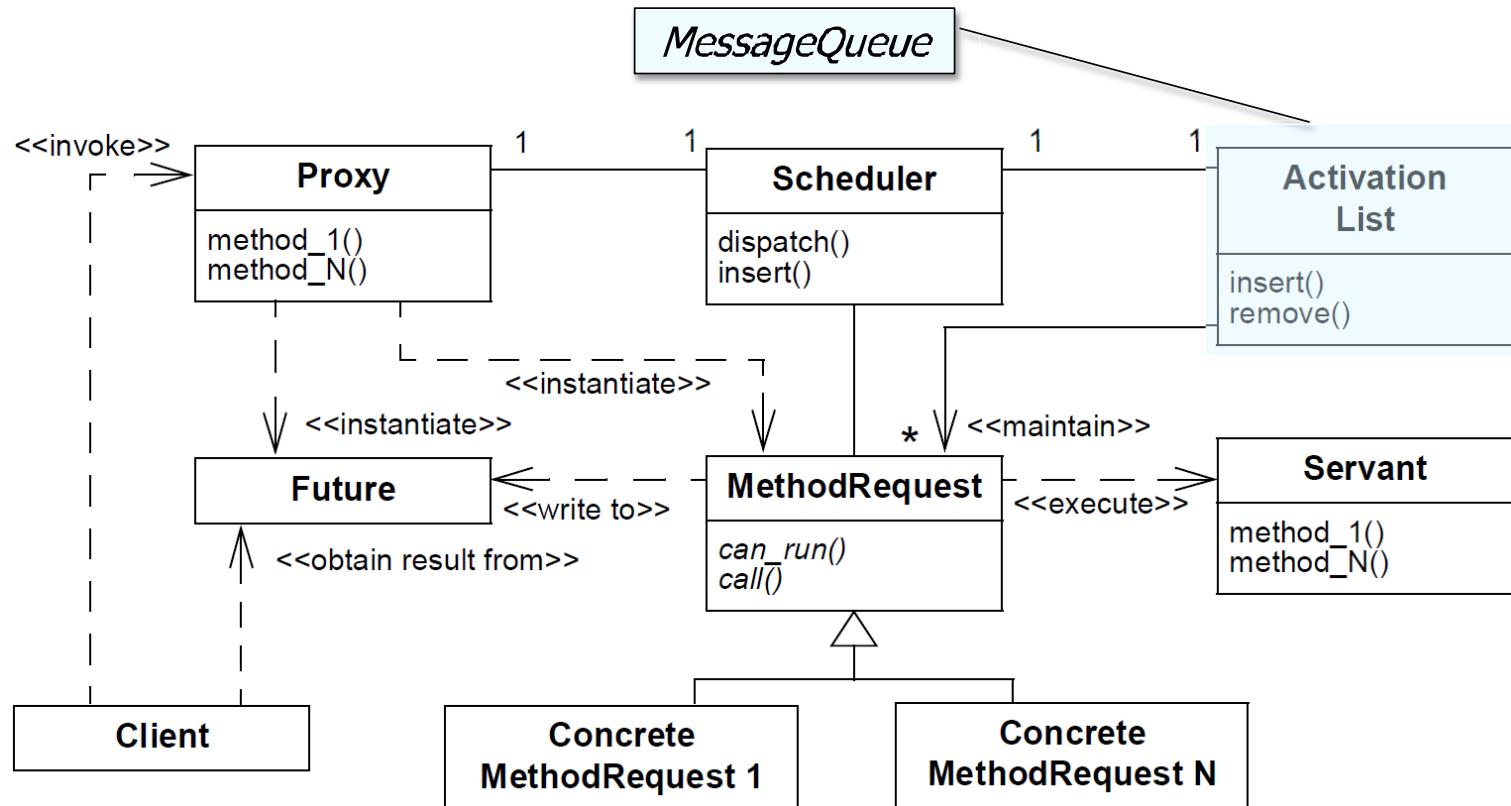
Concurrency Patterns

Active Object Pattern (cont.)



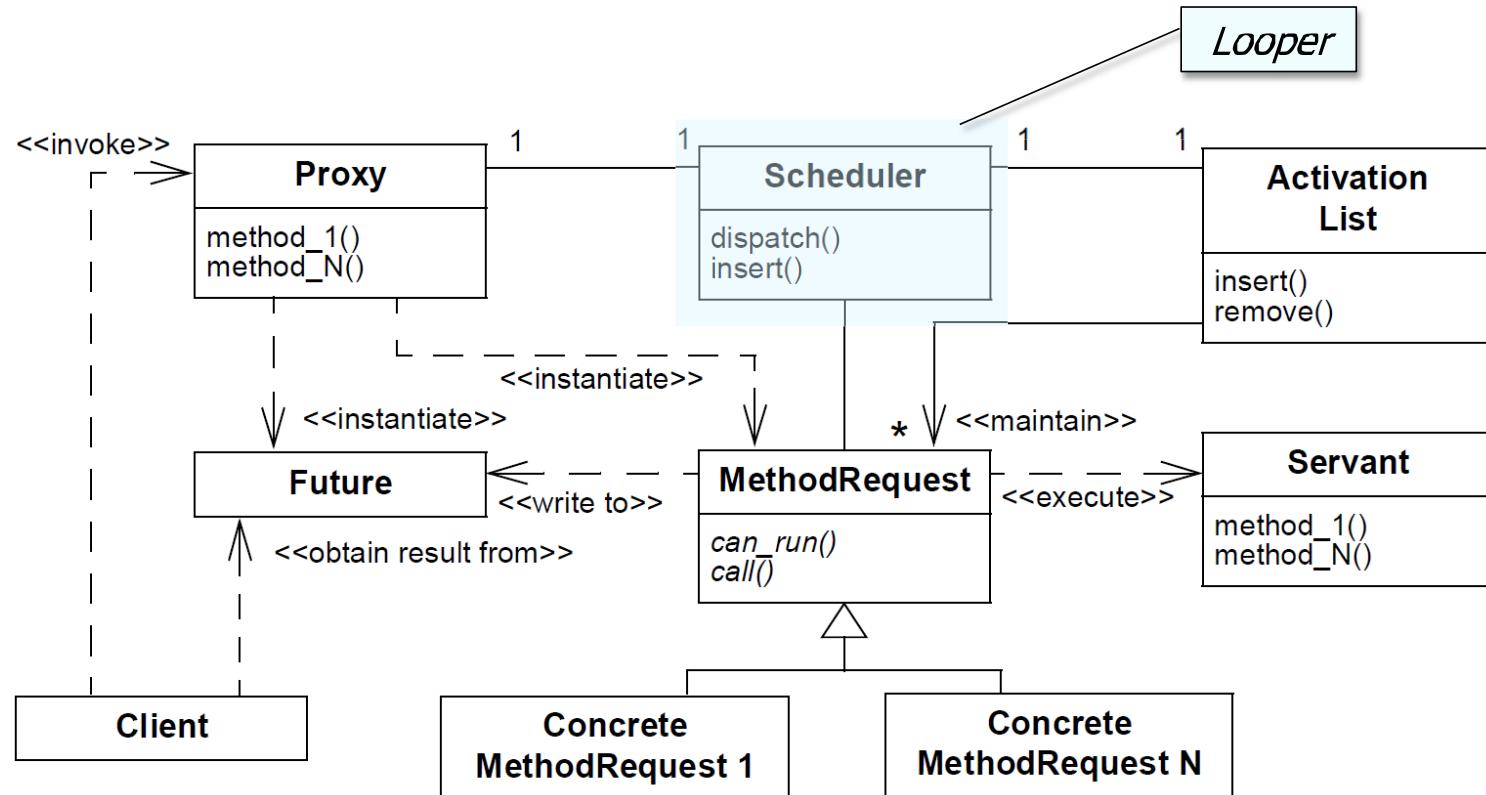
Concurrency Patterns

Active Object Pattern (cont.)



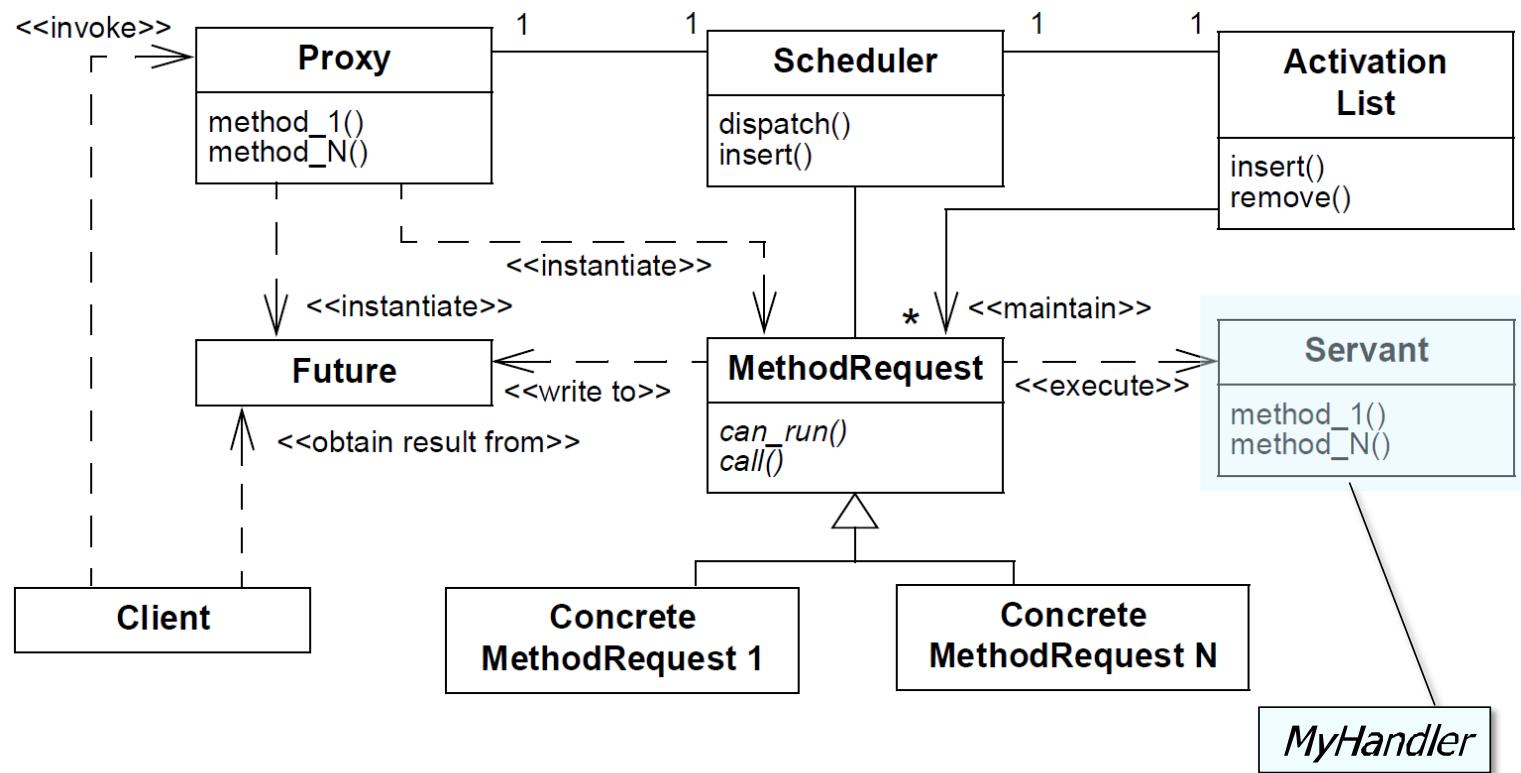
Concurrency Patterns

Active Object Pattern (cont.)



Concurrency Patterns

Active Object Pattern (cont.)



Concurrency Patterns

Active Object Pattern (cont.)

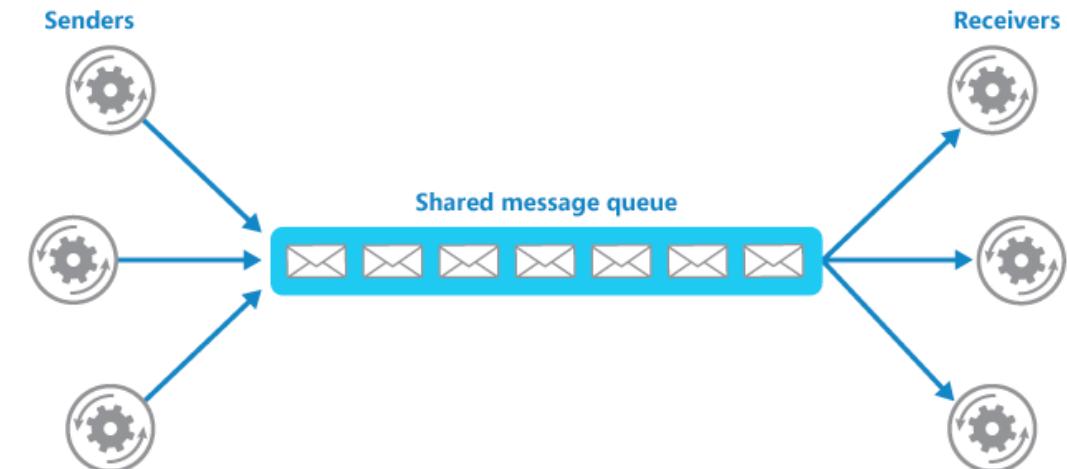


- When to use:
 - When an object's interface methods should define its concurrency boundaries
 - Not to have to acquire or release lock explicitly
 - When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
 - When an object's methods may block during their execution
 - download a big file or long running computation
 - When multiple client method requests can run concurrently on an object
 - When method invocation order might not match method execution order

Concurrency Patterns

Active Object Pattern (cont.)

- Real application:
 - Android - asynchronous background service sending messages to the UI thread
 - Any kind of a queue / message delivery system



Concurrency Patterns

Active Object Pattern (cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Advantages:**
 - Reduced code complexity
 - Once pattern's mechanics are in place, the code can be treated as single-threaded
 - No need for additional synchronization
 - Concurrent requests are serialized and handled by a single internal thread
- **Drawbacks:**
 - Performance overhead
 - Sophisticated scheduling, spinning and request handling can be expensive in terms of memory (lead to non-trivial context switching)
 - Programming overhead
 - Requires a small framework (self-contained but multiple components)

Concurrency Patterns

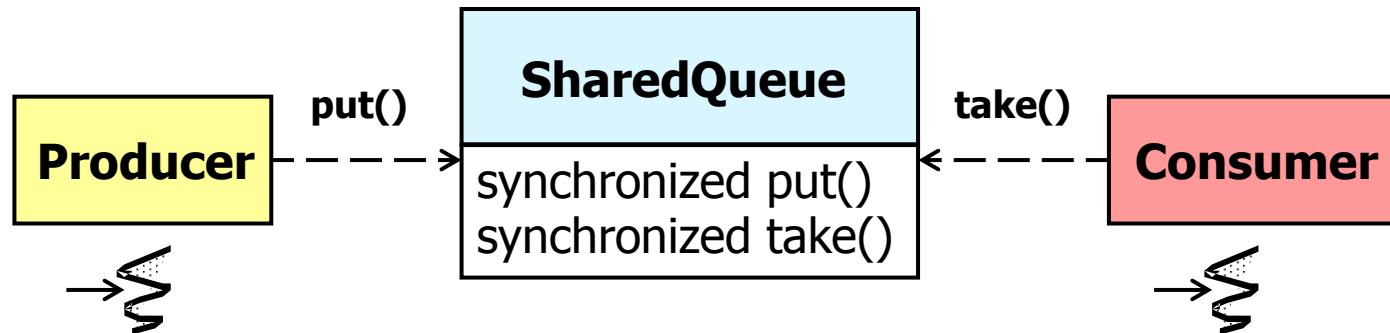
Monitor Object Pattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Challenge: Implementing a Blocking Queue

- Context:
 - Concurrent applications or services that need to coordinate interactions between producer and consumer threads via a shared queue

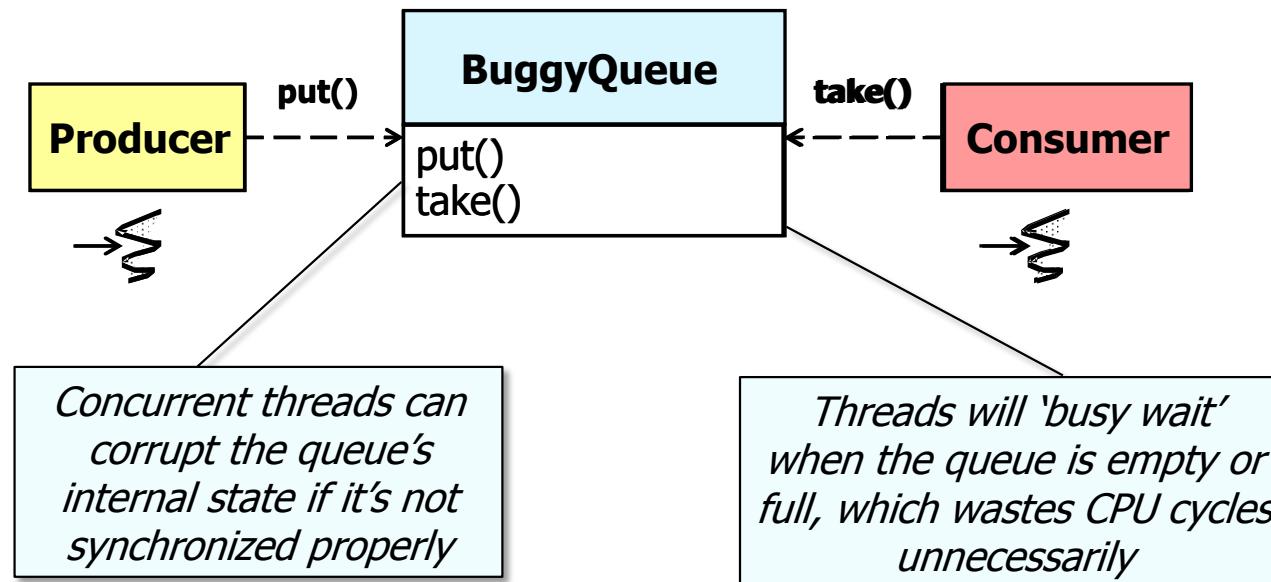


Monitor Object Pattern (cont.)



Challenge: Implementing a Blocking Queue

- **Problems:**
 - Naïve implementations introduce race conditions or “busy waiting” when multiple threads try to update items in the shared queue



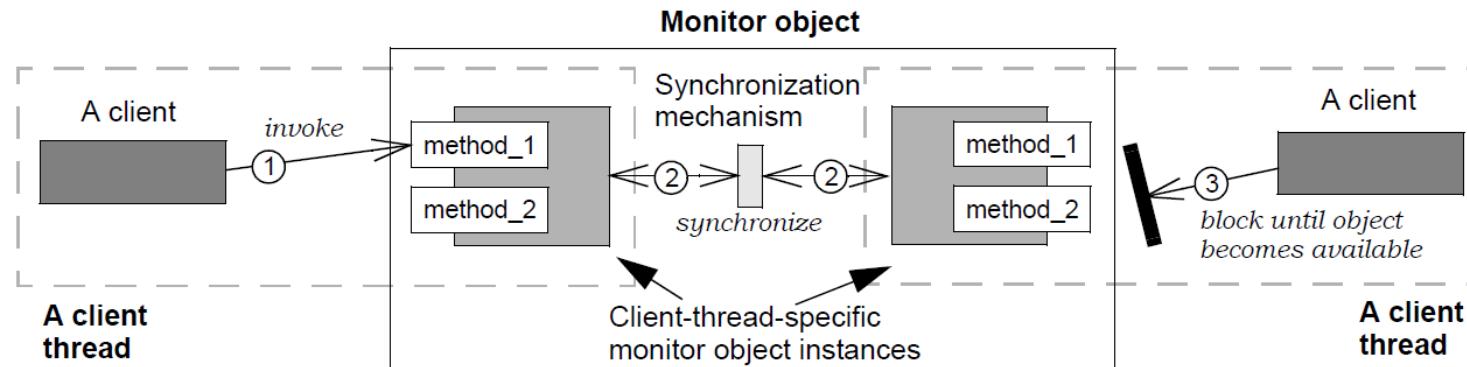
Monitor Object Pattern (cont.)



Challenge: Implementing a Blocking Queue

- **Solution: Monitor Object Pattern**

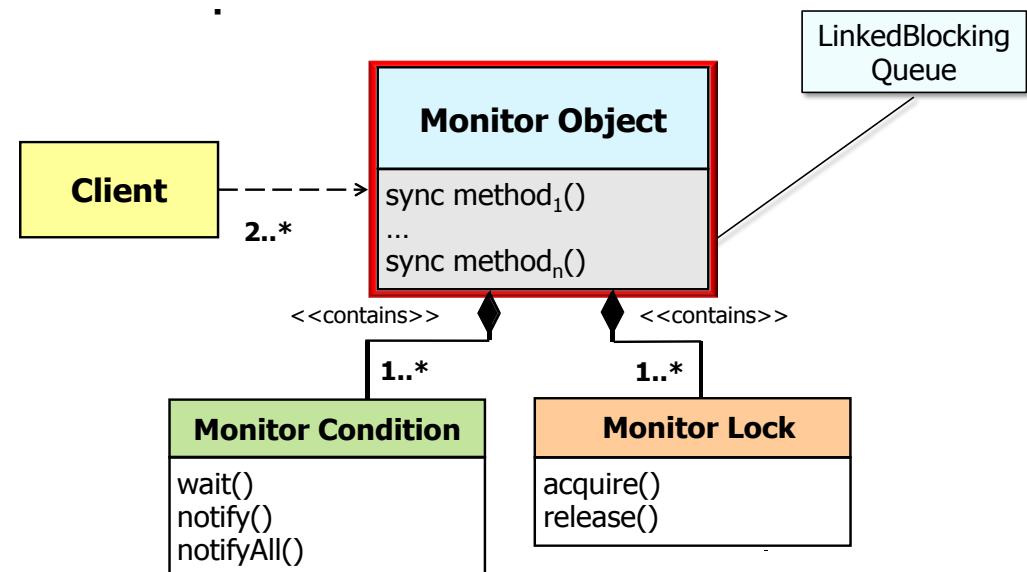
- Mainly revolves around synchronization
- Concurrent threads (clients) can only use the object via a set of synchronized methods (only one method can run at a time)
- Typically a synchronized method watches for a certain condition
- There is no significant performance overhead either, since inefficient busy-waiting loops (polling) are replaced with notifications



Monitor Object Pattern (cont.)



- Key Components:
 - **Monitor Object:** exposes synchronized methods as the only means of client access
 - **Synchronized Methods:** guarantee thread-safe access to the internal state
 - **Monitor Lock:** used by the synchronized methods to serialize method invocation
 - **Monitor Condition:** caters for cooperative execution scheduling



Monitor Object Pattern (cont.)



- When to use:
 - When an object's interface methods can define its synchronization & scheduling boundaries
 - Extension of OOP (protect an objects data from uncontrolled concurrent changes)
 - When only one methods a time should be active within an object
 - When objects should be responsible for transparent method serialization
 - Tedious and error-prone for clients to work with sync mechanisms
 - When an object's methods interact cooperatively via multiple threads
 - Object-specific invariants must hold as threads suspend and resume their execution

Concurrency Patterns

Monitor Object Pattern (cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Advantages:
 - **Simplified synchronization:** All of the hard work is offloaded to the object itself, clients are not concerned with synchronization issues
 - **Cooperative execution scheduling:** Monitor conditions are used to suspend / resume method execution
 - **Reduced performance overhead:** Notifications over inefficient polling
- Drawbacks:
 - **Synchronization tightly coupled with core logic:** Synchronization code blends into the business logic which breaks the principle of separation of concerns
 - **Nested monitor lockout problem:** An endless wait for a condition to become true can occur when a monitor object is nested into another kind of its own

Concurrency Patterns

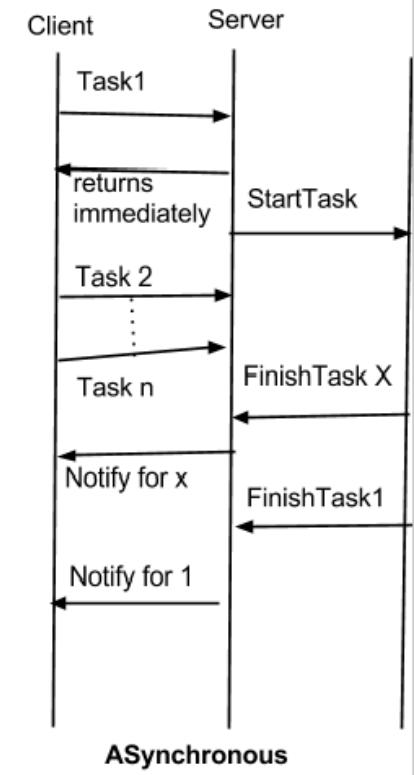
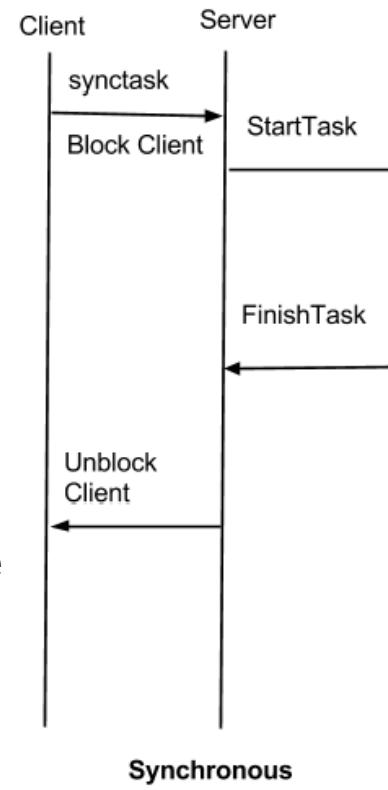
Half-Sync / Half-Async Pattern



Challenge: Combining Sync & Async Processing

- **Context:**

- A concurrent system that performs both asynchronous & synchronous processing services that must communicate
- A multi-threaded downloader application is a good example





Challenge: Combining Sync & Async Processing

- **Problems:**
 - Services that want the **simplicity of synchronous** processing shouldn't need to address the **complexities of asynchronous** part
 - Sync & Async processing services should be able to communicate without complicating their programming model or degrading their performance

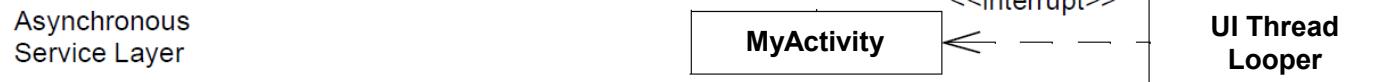
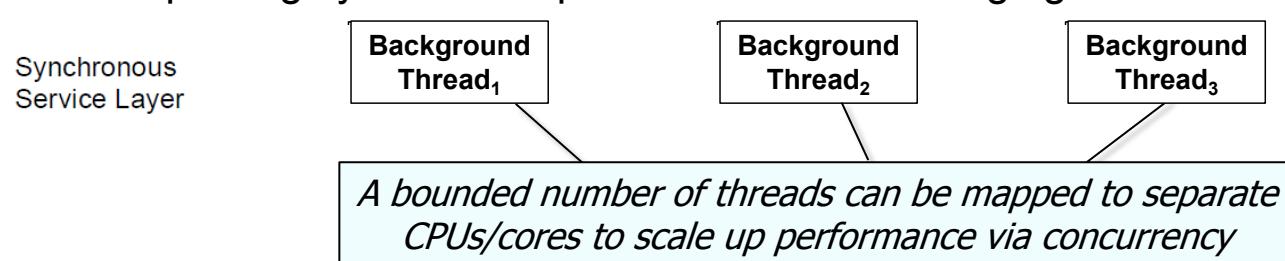
Half-Sync / Half-Async Pattern (cont.)



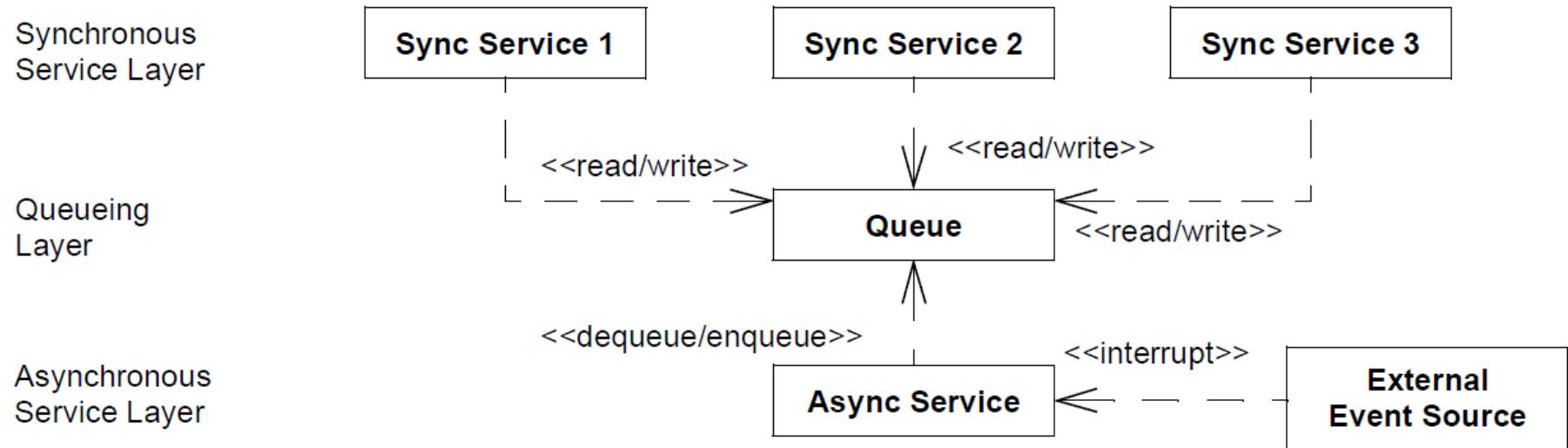
Challenge: Combining Sync & Async Processing

- **Solution: Half-Sync/Half-Async Pattern**

- Decompose the services in the system to 2 layers: Sync and Async
- Main thread doesn't block on incoming client requests and long-running operations are offloaded to a dedicated synchronous layer
- Processing results are delivered by the means of callbacks
- A decent queuing system is required to handle messaging between the two layers



- Key Components:
 - **Synchronous Service Layer:** deals with long-running tasks, implements the core business logic
 - **Queuing Layer:** a request queue, doesn't block the caller
 - **Asynchronous Service Layer:** dispatches incoming requests to the queue

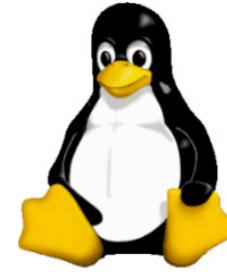


Half-Sync / Half-Async Pattern (cont.)



- When to use:
 - When it is necessary to make **performance efficient & scalable**, while also ensuring that the use of concurrency simplifies, rather than complicates programming
 - When there are constraints on certain types of operations in certain contexts:
 - short-duration vs. long-duration
 - blocking vs. non-blocking
 - etc.

- Application:
 - largely used in operating systems (hardware interrupts, application management)
 - Android - AsyncTask (file downloads, ...)



Half-Sync / Half-Async Pattern (cont.)



- Advantages:
 - **Reduced code complexity:** Synchronized services focus solely on the core logic
 - **Separation of concerns:** Each of the layers is relatively self-contained and serves a different purpose
 - **Centralized communication:** The queuing layer mediates all communication – no moving parts flying around
- Drawbacks:
 - **Performance overhead:** "Boundary-crossing penalty" – context switching, synchronization, data transfer, ...
 - **Harder to debug:** Asynchronous callbacks make testing and debugging less straightforward
 - **Benefit questionable:** Higher-level application services may not benefit from asynchronous I/O (depends on framework / OS design)

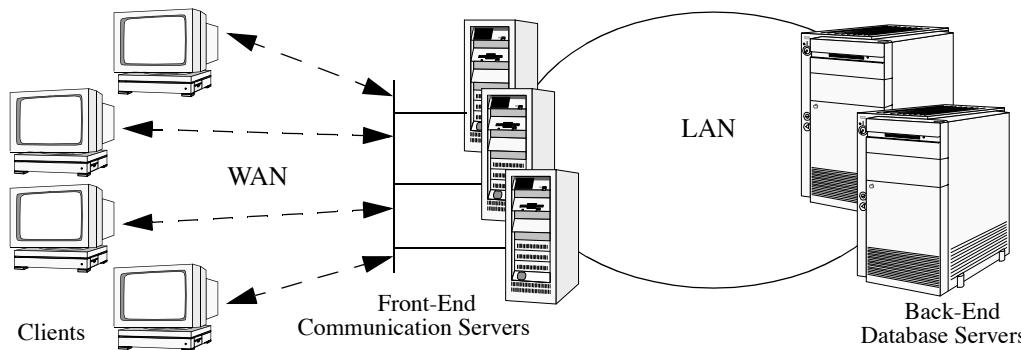
Concurrency Patterns

Leader/Followers Pattern



Challenge: Processing Numerous Types of Inputs Concurrently

- **Context:**
 - An event driven application where multiple services requests arriving on a set of event sources must be processed efficiently by multiple threads that share the event sources
 - e.g. an online transaction processing (OLTP) system



Leader/Followers Pattern (cont.)



Challenge: Processing Numerous Types of Inputs Concurrently

- **Problem:**
 - Associating a thread for each event source may be infeasible due to scalability limitation
 - Allocating memory dynamically for each request incur significant overhead due to context-switching, synchronization and cache coherency
 - As happen in Half-Sync/Half-Async pattern thread pool
 - Multiple threads that demultiplex events on a shared set of event sources must coordinate to prevent race conditions

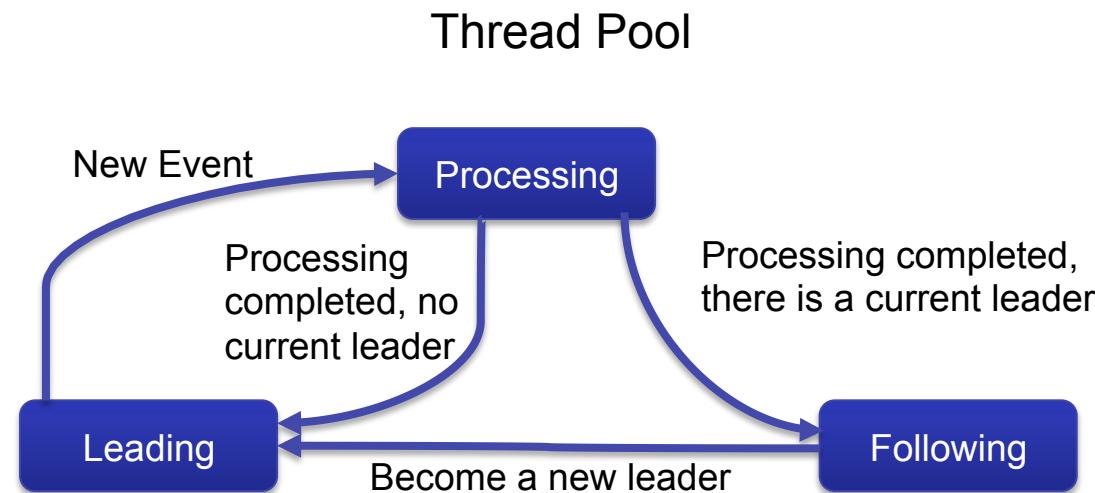
Leader/Followers Pattern (cont.)



Challenge: Processing Numerous Types of Inputs Concurrently

- **Solution: Leader/Followers Pattern**

- Multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources



Concurrency Patterns

Leader/Followers Pattern (cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

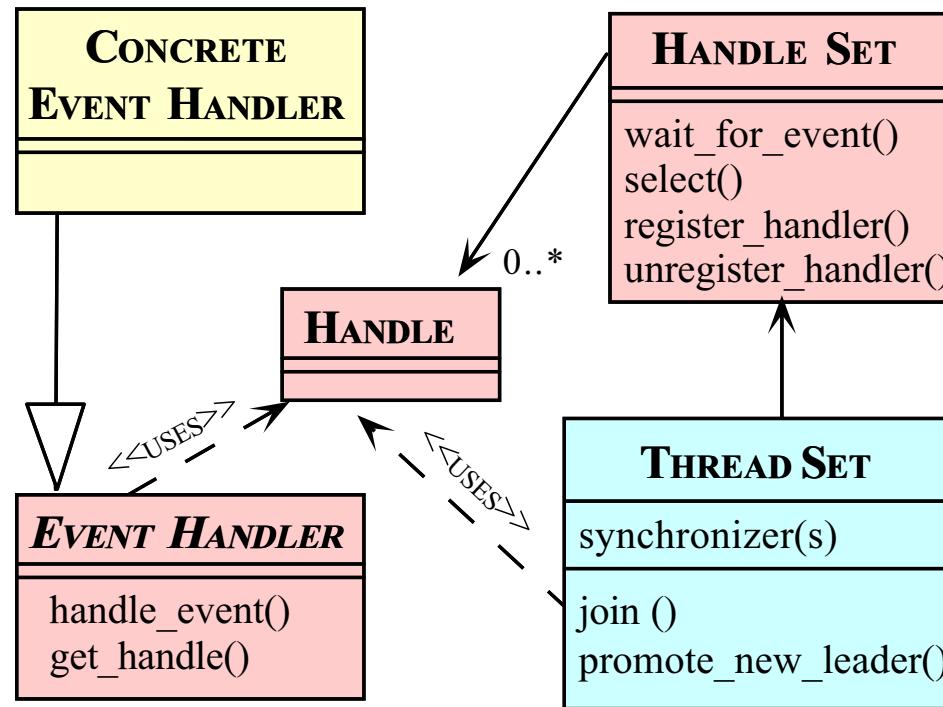
- Key Components:
 - **Handle & Handle sets**
 - Identifies a source of events
 - Can queue up events
 - Handle set is collection of handles
 - E.g. network connections provided by OS
 - **Event Handler**
 - Defines an interface for processing events that occur on a handle
 - **Concrete Event Handler**
 - Specializes the event handler
 - Defines an application service
 - Processes events received on a handle
 - Runs in a processing thread
 - **Thread pool set**
 - Set of threads take turns in leader/follower/processing roles
 - Contains synchronizer and implement a protocol for coordinating

Concurrency Patterns

Leader/Followers Pattern(cont.)



- Key Components:



Concurrency Patterns

Leader/Followers Pattern(cont.)



- Example: OLTP Servers
 - **Handles:** each network connection is a source of events represented in a server by a separate socket handle
 - 2 types of events: CONNECT and READ
 - Use select() event demultiplexer, which identifies handles whose event sources have pending event
 - Applications can invoke I/O operations on these handles without blocking the calling threads
 - **EventHandlers**
 - Concrete event handlers in OLTP front-end servers receive and validate remote client requests
 - Forwards requests to back-end database servers

Concurrency Patterns

Leader/Followers Pattern(cont.)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Advantages:
 - No communication between the threads are necessary, no synchronization, no shared memory (no locks, mutexes) are needed
 - More ConcreteEventHandlers can be added without affecting any other EventHandler
 - Minimizes the latency because of the multiple threads
- Drawbacks:
 - Complex and hard to implement (a set of follower threads waiting)
 - Lack of flexibility (no explicit queue, hard to discard or reorder events)
 - Input I/O can become a bottleneck (allowing only a single thread at a time to wait on the handle set)

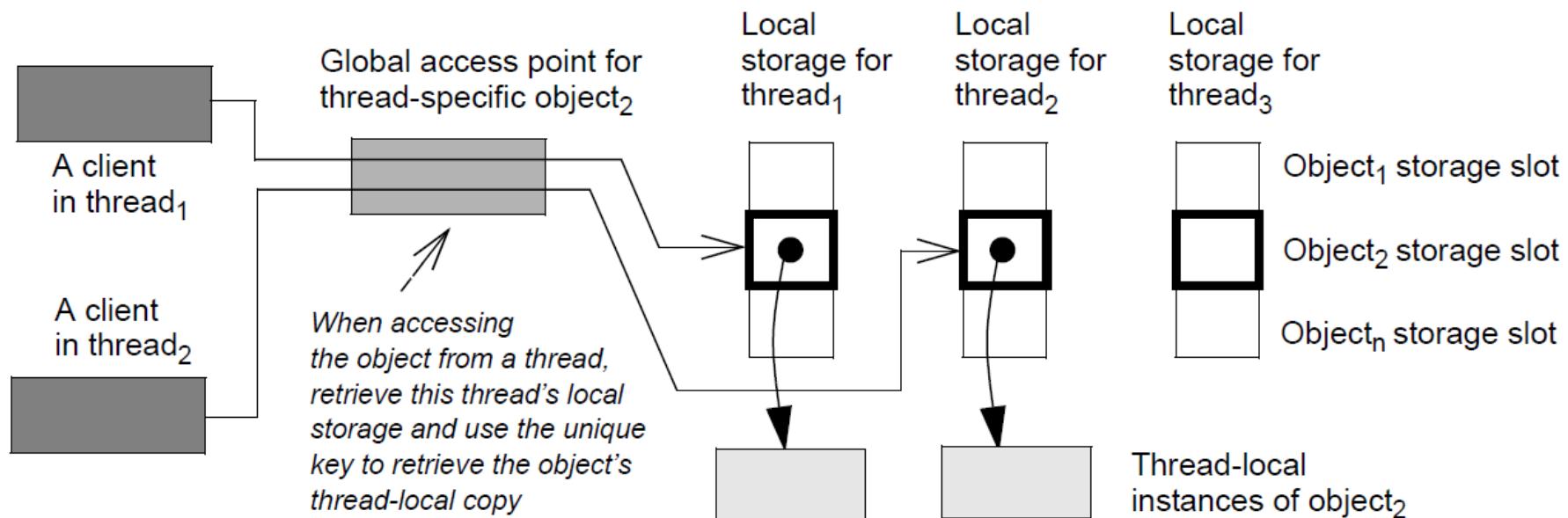
Concurrency Patterns

Thread-Specific Storage Pattern



Challenge: Ensuring One Specific Variable per Thread

- Allows multiple threads to use one ‘logically global’ access point to retrieve an object that is local to a thread, without incurring locking overhead on each object access.



Thread-Specific Storage Pattern (cont.)



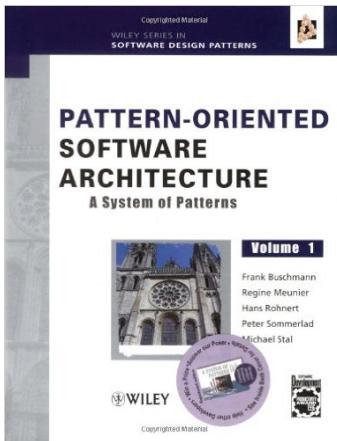
- This pattern is basically available in modern programming languages like C++11 and Java.
- In C++11 you can create such a local storage via `thread_local` identifier.

Further References

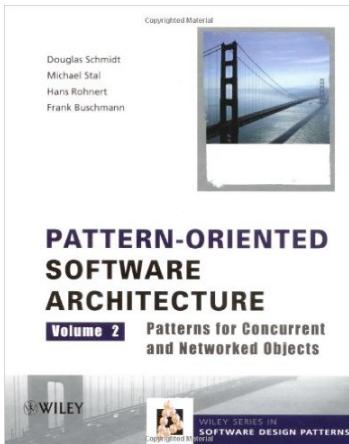


TECHNISCHE
UNIVERSITÄT
DARMSTADT

[POSA1]: **Pattern-Oriented Software Architecture Volume 1: A System of Patterns**, *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal*, Wiley Publications, 1996



[POSA2]: **Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects**, *Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann*, Wiley Publications, 2000





TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

PARALLEL PROGRAMMING IN .NET

Outline

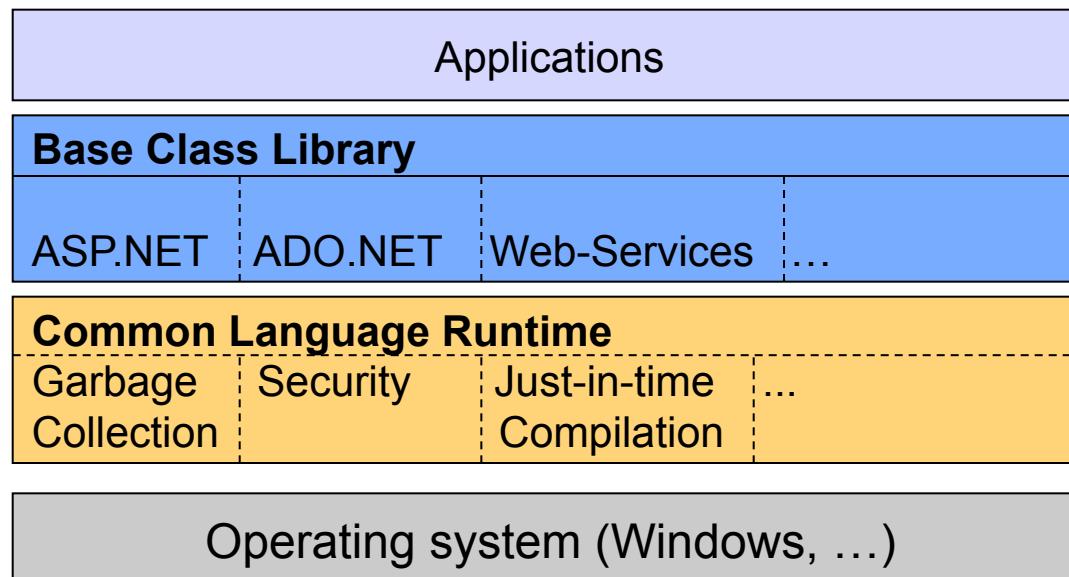


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Overview of .NET
- Parallel Programming in .NET
 - Win32 Application Programming Interface (Win32 API)
 - Common Language Runtime (CLR)
 - .NET Threading Library
 - Task Parallel Library (TPL)
 - Parallel Language Integrated Query (PLINQ)

Overview of .NET

- The .NET Framework is the core of .NET technologies
- Contains a runtime environment and a object-oriented class library for both Windows- and Web-based applications



Overview of .NET (2)

- The runtime environment is based on a virtual machine (similar to Java) with its own intermediate representation (IR): the Common Intermediate Language (CIL)
 - All .NET languages are translated into CIL
 - CIL code will be translated into target machine code right before execution (just-in-time compilation)
 - CIL code guarantees interoperability between different languages and portability of the code

Overview of .NET (3)

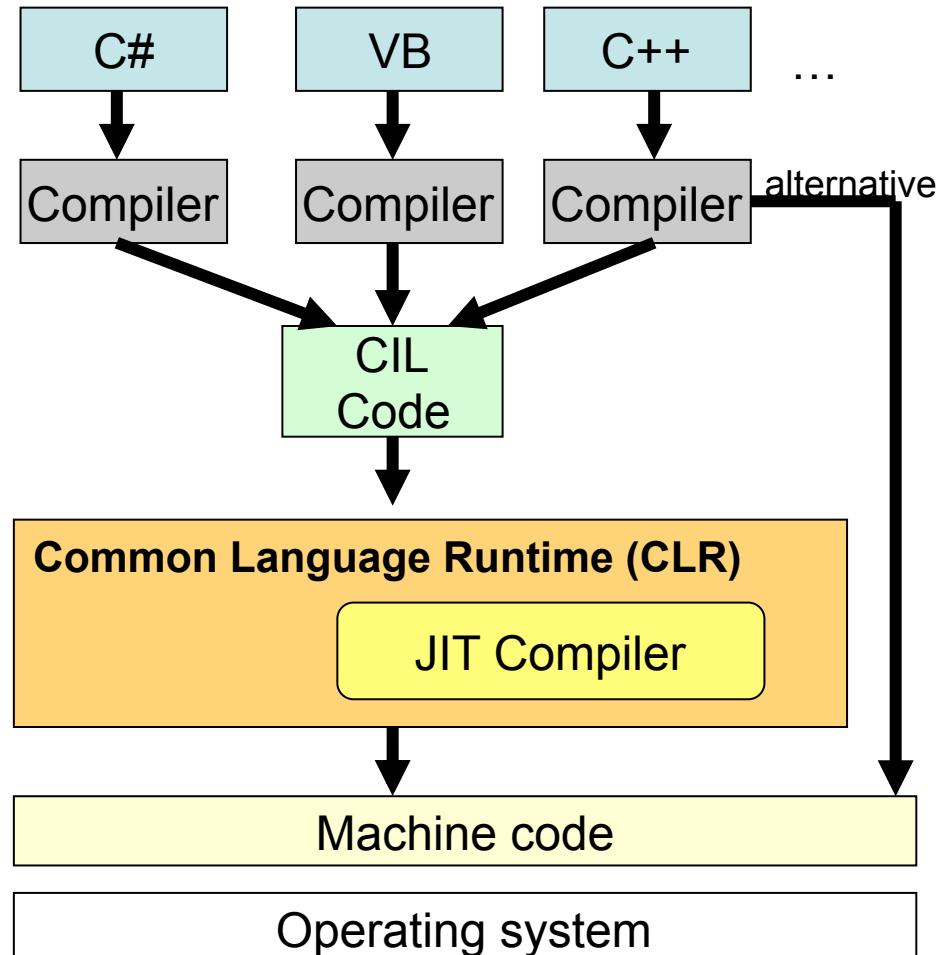


C# `if (a > b) max = a;
else max = b;`

CIL `IL_0004: ldloc.0
IL_0005: ldloc.1
IL_0006: ble.s IL_000c
IL_0008: ldloc.0
IL_0009: stloc.2
IL_000a: br.s IL_000e
IL_000c: ldloc.1
IL_000d: stloc.2`

Intel-
Code

```
mov ebx, [-4]  
mov edx, [-8]  
cmp ebx, edx  
jle 17  
mov ebx, [-4]  
mov [-12], ebx  
...
```



A short introduction to delegate



- A *delegate* is a type that represents references to methods with a particular parameter list and return type
 - When you instantiate a delegate, you can associate its instance with any method (static or dynamic instance) with a compatible signature and return type
 - You can invoke (or call) the method through the delegate instance
 - Similar to function pointers in C, but type-safe
 - Delegates allow methods to be passed as parameters

An example of delegate in C#



```
class Program {  
    // Definition of a delegate  
    public delegate double ProcessOperation(double dblVar1, double dblVar2);  
  
    static void Main(string[] args) {  
        ProcessOperation process; // instance of the delegate  
        if(opt == "A")  
            process = new ProcessOperation(Addition);  
        else if(opt == "S")  
            process = new ProcessOperation(Subtraction);  
        //...  
        double result = process(input1, input2); // call the delegate  
    }  
    public static double Addition(double x, double y) {return x + y;}  
    public static double Subtraction(double x, double y) {return x - y;}  
}
```

An example of delegate in C# (2)



```
delegate void SampleDelegate(string message); // declare a delegate void (*fp)(string)

class MainClass {

    // define a function

    static void SampleDelegateMethod(string message) { Console.WriteLine(message); }

    static void Main() {

        // instantiate the delegate with the pre-defined function

        SampleDelegate d1 = SampleDelegateMethod;

        // instantiate the delegate with an anonymous function

        SampleDelegate d2 = delegate(string message) { Console.WriteLine(message); };

        // call delegate instances d1 and d2

        d1("Hello");

        d2(" World");

    }

}
```

Parallel programming in .NET



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Parallel Language Integrated Query (PLINQ)

declarative approach

Implemented on top of LINQ and TPL

Task Parallel Library (TPL)

Data- & Task-Parallelism (imperative approach)

Base Class Library
including .NET Threading Library

Common Language Runtime (CLR)

Win32 Application Programming Interface (API)

Operating system (Windows)

Win32 Application Programming Interface (Win32 API)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Interface to low-level abstraction between applications and operating system
- Can be directly called in applications
- Written in C
- Implementations are in a set of dynamic linked libraries (DLLs)

Parallel-related functionalities of Win32 API



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Creating and destroying processes and threads
- Critical sections, locking mechanism
- Atomic operations
- Waiting and signals
- Thread pool management

Parallel programming in .NET



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Parallel Language Integrated Query (PLINQ)

declarative approach

Implemented on top of LINQ and TPL

Task Parallel Library (TPL)

Data- & Task-Parallelism (imperative approach)

Base Class Library
including .NET Threading Library

Common Language Runtime (CLR)

Win32 Application Programming Interface (API)

Operating system (Windows)

Common Language Runtime (CLR)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- The virtual machine executing CIL code
- Mapping from CLR-internal threads to OS threads
 - using Win32 API
 - A CLR-internal thread can be mapped to
 - A Windows thread
 - A *fiber*
 - A lightweight thread (stack and a subset of registers)
 - A thread can schedule multiple fibers, but only one at a time
 - Not visible to Windows

Common Language Runtime (CLR) (2)

- Creating and destroying threads are still expensive
- CLR manages threads using the concept of *thread pool*
 - Each process has a thread pool
 - When a thread pool is initialized, a number of (initially inactive) threads are created
 - Work will be submitted into a queue
 - The thread pool assign work to an idle thread



Example: Thread pool

```
using System;
using System.Threading;

public static class Program{
public static void Main(){
    Console.WriteLine("Mainthread - before");
    ThreadPool.QueueUserWorkItem(Compute);
    Console.WriteLine("Mainthread - after");
}

// this method will be executed by
// definition of the function must include
// delegate void WaitCallback(Object)

private static void Compute(Object state)
{
    Console.WriteLine("compute from: " + state);
    Thread.Sleep(1000); //simulate work
    // When this function returns, the thread
    // waiting status in thread pool
}
}
```

Examples of possible outputs:

Mainthread - before

Mainthread - after

compute from: state=5

or

Mainthread - before

compute from: state=5

Mainthread - after

The different order comes from asynchronous execution. The scheduling is determined by the operating system.

Asynchronous processing in thread pool



- Using the "Asynchronous Programming Model" - APM
 - For example, I/O can be done asynchronously with computations
 - How to know an asynchronous task is finished?
 - Wait-until-finish
 - Polling
 - Callback



Example: Wait-until-finish

```
using System;
using System.IO;
using System.Threading;

Public static class Program{
public static void Main(){
    FileStream fs=new FileStream(@"C:\file.txt", FileMode.Open,
                                FileAccess.Read, FileShare.Read, 1024,
                                FileOptions.Asynchronous);
    Byte[] data = new Byte[100];
    // initialize asynchronous reading
    IAsyncResult ar=fs.BeginRead(data,0,data.length,null,null);

    // do something while reading...
    // block until asynchronous reading is finished
    Int32 bytesRead=fs.EndRead(ar);

    // close file
    fs.Close();
}}
```

receipt ↴

```
public interface IAsyncResult{
    Object AsyncState
    WaitHandle AsyncWaitHandle
    Boolean IsCompleted
    Boolean IsCompletedSynchronously}
```

Not cheap: If there is not enough work to do in parallel with reading, then the thread must wait I/O to finish



Example: Polling

```
using System;
using System.IO;
using System.Threading;

Public static class Program{
public static void Main(){
    FileStream fs=new FileStream(@"C:\file.txt", FileMode.Open,
                                FileAccess.Read, FileShare.Read, 1024,
                                FileOptions.Asynchronous);
    Byte[] data = new Byte[100];
    // initialize asynchronous reading
    IAsyncResult ar=fs.BeginRead(data,0,data.length,null,null);

    while (!ar.IsCompleted){
        Console.WriteLine("Operation nicht fertig; warte noch");
        Thread.Sleep(10); // save some CPU time for other threads
    }

    // get result (this time it will not block)
    Int32 bytesRead=fs.EndRead(ar);

    fs.Close();
}}
```

Polling is easy to implement, but it still waste some CPU time to ask the status of asynchronous tasks.

AsyncCallback



- Applications that can do other work while waiting for the results of an asynchronous operation should not block waiting until the operation completes
- Use an **AsyncCallback** delegate to process the results of the asynchronous operation in a separate thread
 - An asynchronous call puts the task into the queue of the thread pool
 - Once the task is done, it puts another task into the queue for processing the results
 - The task of processing the results will be executed by a thread in the thread pool, not necessarily the calling thread
 - The thread executes the callback provided by the programmer to process the results



Example: AsyncCallback (1/2)

```
using System;
using System.IO;
using System.Threading;

Public static class Program{
    // make s_data reachable by both Main() and the callback
    private static Byte[] s_data = new Byte[100];

    public static void Main(){
        // print the current thread ID
        Console.WriteLine("Main thread ID={0}",
            Thread.CurrentThread.ManagedThreadId);

        FileStream fs=new FileStream(@"C:\file.txt", ...);
        //initiere asynchrones Lesen
        //übergeb FileStream fs an Callback-Methode ReadIsDone
        IAsyncResult ar = fs.BeginRead(s_data,0,s_data.length,ReadIsDone,fs);

        // do other work, not care about the reading anymore...
    }
}
```



Callback and the state will be used in processing results



Example: AsyncCallback (2/2)

```
public static void ReadIsDone(IAsyncResult ar){  
    // print the current thread ID  
    Console.WriteLine("ReadIsDone thread ID={0}",  
                      Thread.CurrentThread.ManagedThreadId);  
  
    // extract FileStream from IAsyncResult object  
    FileStream fs=(FileStream) ar.AsyncState;  
  
    // retrieve results  
    Int32 bytesRead=fs.EndRead(ar);  
  
    // close file  
    fs.Close();  
  
    // process results...  
}  
}
```

EndRead() is now in callback.

Example of possible outputs:

Main thread ID=1

ReadIsDone thread ID=4

Note: ReadIsDone can be executed in a thread other than the main thread.



Parallel Language Integrated Query (PLINQ)

declarative approach

Implemented on top of LINQ and TPL

Task Parallel Library (TPL)

Data- & Task-Parallelism (imperative approach)

Base Class Library
including .NET Threading Library

Common Language Runtime (CLR)

Win32 Application Programming Interface (API)

Operating system (Windows)

Thread in C#



- A thread object in C# is an abstraction of an CLR-internal thread (remember the threads and locks model?)
- Associate a method with an thread object to execute the method
 - Method signature: `void MethodName () ;`
 - The method can be either static or instance, public or private, from the same object or a different object

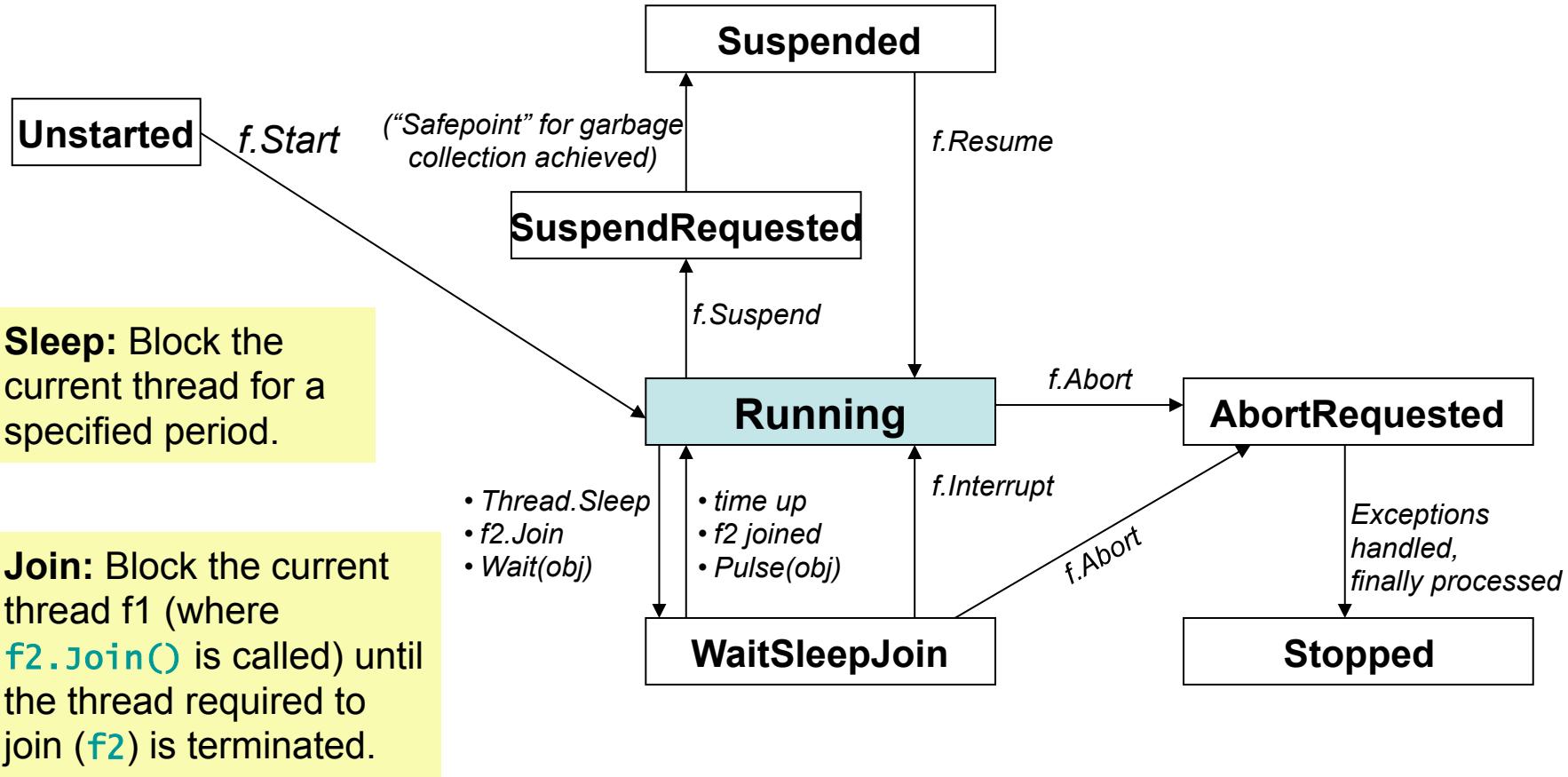


Example: Thread in C#

```
using System;
using System.IO;
using System.Threading;

class ThreadTest {
    static void Main() {
        Thread t = new Thread (
            delegate(){ Console.WriteLine ("hello!"); }
        );
        // execute in a new thread
        t.Start();
    }
}
```

Statuses and transitions of a thread in C#





- Provides a mechanism that synchronizes access to objects
 - Calling `Monitor.Enter(obj)` blocks the current thread until it gets the lock on `obj`
 - The lock of an object can be obtained by the same thread multiple times (recursive lock)
 - If a thread has locked `obj` n times, it must unlock `obj` n time to fully release the lock
 - Calling `Monitor.Exit(obj)` release the lock on `obj` (the lock must be held by the current thread when calling `Monitor.Exit()`)

More about .NET threading library



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Thread execution priority
- Named thread (e.g., for debugging purpose)
- Exception handling
- ...

Parallel programming in .NET



Parallel Language Integrated Query (PLINQ)

declarative approach

Implemented on top of LINQ and TPL

Task Parallel Library (TPL)

Data- & Task-Parallelism (imperative approach)

Base Class Library
including .NET Threading Library

Common Language Runtime (CLR)

Win32 Application Programming Interface (API)

Operating system (Windows)

Overview of Task Parallel Library (TPL)



- Task: An asynchronous operation
 - Resembles a thread or thread pool work item, but at a higher level of abstraction
 - Task parallelism refers to one or more *independent* tasks running concurrently
- Benefits
 - More efficient and more scalable use of system resource
 - More programmatic control than is possible with a thread or work item

TPL is available since .NET Framework 4.0. Some features requires .NET Framework 4.5 or higher.

Overview of Task Parallel Library (TPL) (2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- The programmer is responsible to explore parallelism
- Does not guarantee parallel execution
- Tasks are queued to thread pool
- Scheduler implements work-stealing

Due to the fact that tasks are independent from each other!

TPL Parallel class: Parallel.For



```
for (int i = 0; i < 100; i++) {           // sequential
    a[i] = a[i] * a[i];
}
```

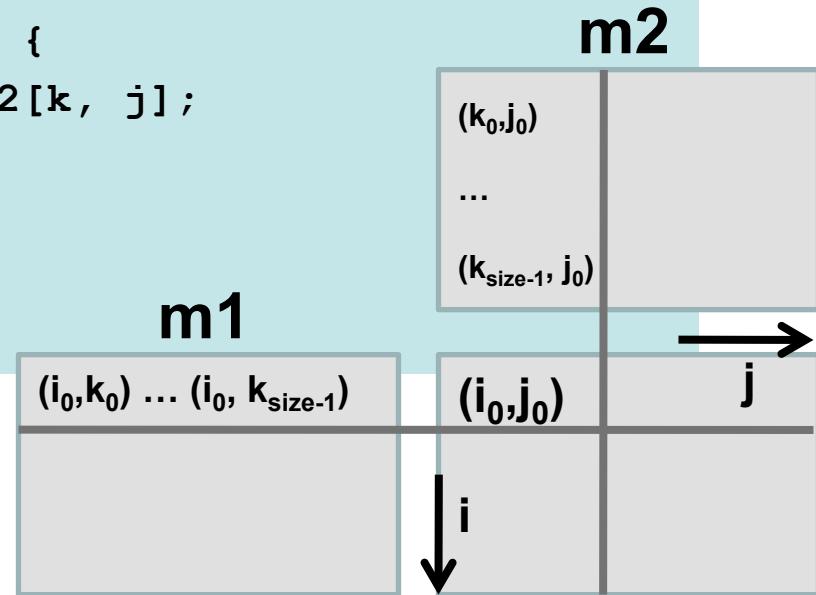
```
Parallel.For (0, 100, delegate(int i){ // parallel
    a[i] = a[i] * a[i];
});
```

- Loop body is passed into Parallel.For as an anonymous delegate
- The loop has to be an DOALL loop: No loop-carried dependencies!

Example: Matrix multiplication (1/3)



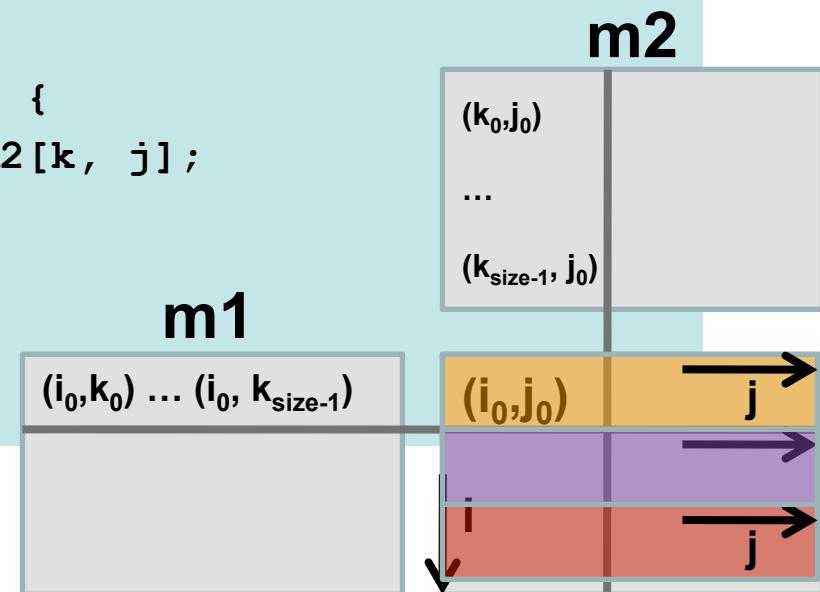
```
void SeqMatrixMult(int size,
                    double[,] m1, double[,] m2,
                    double[,] result) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            result[i, j] = 0;
            for (int k = 0; k < size; k++) {
                result[i, j] += m1[i, k] * m2[k, j];
            }
        }
    }
}
```



Example: Matrix multiplication (2/3)



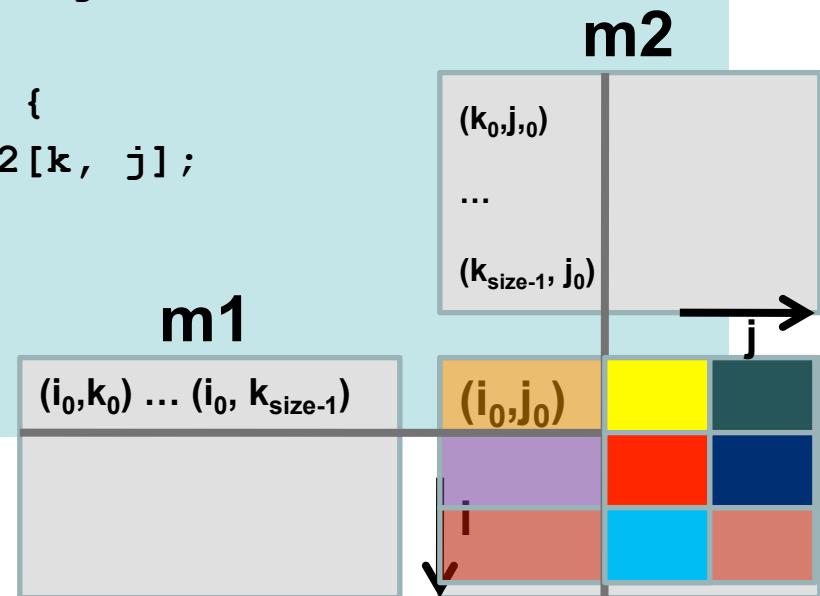
```
using System.Threading.Tasks;  
  
void ParMatrixMultV1(int size,  
                      double[,] m1, double[,] m2,  
                      double[,] result) {  
  
    Parallel.For(0, size, delegate(int i){  
        for (int j = 0; j < size; j++) {  
            result[i, j] = 0;  
            for (int k = 0; k < size; k++) {  
                result[i, j] += m1[i, k] * m2[k, j];  
            }  
        }  
    });  
}
```



Example: Matrix multiplication (3/3)

```
using System.Threading.Tasks;
void ParMatrixMultV2(int size,
                     double[,] m1, double[,] m2,
                     double[,] result) {
    Parallel.For(0, size, delegate(int i) {
        Parallel.For(0, size, delegate(int j) {
            result[i, j] = 0;
            for (int k = 0; k < size; k++) {
                result[i, j] += m1[i, k] * m2[k, j];
            }
        });
    });
}
```

Can the inner-most loop be parallelized using `Parallel.For`?



Mutual exclusion using Lock



```
// sequential
int sum = 0;
for(int i = 0; i < 100; i++) {
    if (isPrime(i)) sum += i;
}
```

```
// parallel(?)
int sum = 0;
Parallel.For(0, 100, delegate(int i) {
    if (isPrime(i)) {
        lock(this){ sum += i; }
    }
});
```

Using Parallel.ForEach for reduction



- An accumulative operation on a set of values
 - Each worker thread calculates a partial result locally without locking the shared data structure
 - One or more worker threads combine local results with mutual exclusion

```
int sum = 0;
int[] sequence = ...;

Parallel.ForEach(sequence,
    () => 0,                                // sequence
    (x, loopState, partial) => { return x + partial; } // initial partial result
    (localPartial) => {                         // loop body
        lock(lockObject) { sum += localPartial; }
    });
return sum;
```

Performance of Parallel.For



- Each delegate creates a class, and the class is instantiated into objects at runtime
 - In V1, delegate objects are created for the outer loop
 - In V2, delegate objects are created for both the outer loop and the inner loop

Since each task only has a small amount of work to do, the overhead of creating delegate objects may outweigh the benefit of parallelizing the inner loop
- Creating tasks also has overhead

If the amount of work for each task is too small, `Parallel.For` may actually slow the program down.

Parallel.For: Behind the scene



- Statically divide iterations into blocks, and encapsulate each block as a task

The division is based on available resources, e.g., the number of cores. This is automatically done.
- Automatically push tasks into the queue of a thread pool
- Automatic load balancing (e.g., work stealing)
- Manage the number of worker threads dynamically

Parallel.For: Exception handling



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- If any iteration triggered an exception, all tasks created by the same **Parallel.For** construct abort
- The exception will be forwarded to the calling thread so that the user can handle the exception

TPL: Parallel.Invoke



- The user specify one or more methods that should run in parallel
- The runtime handles all thread scheduling details
- Does not guarantee the methods actually run in parallel

```
using System.Threading.Tasks;

Parallel.Invoke(() => {
    Console.WriteLine("Begin first task...");
    GetLongestWord(words);
}, // first Action, static method
delegate() {
    Console.WriteLine("Method=gamma, Thread={0}",
        Thread.CurrentThread.ManagedThreadId);
} // second action, anonymous delegate
); // parallel.invoke
```

TPL Task class



- Represents an asynchronous operation

```
using System;
using System.Threading.Tasks;

public class Example {
    public static void Main() {
        Task t = Task.Run( () => {
            int ctr = 0;
            for (ctr = 0; ctr <= 1000000; ctr++) {}
            Console.WriteLine("Finished {0} loop iterations",
                ctr);
        } );
        t.Wait();
    }
}
```

TPL TaskFactory class



- Provides support for both creating and scheduling Task objects
 - Scheduling here means manage dependencies between tasks
Scheduling of how tasks are mapped onto threads in thread pool are handled by **TaskScheduler** class
 - If tasks do not form task graph, it is recommended to use **Task.Run** to create new tasks

TPL: Futures



- A *future* in .NET is a task that returns a value

Implements the concept of futures in functional programming.

- **Task<TResult>**

```
Task<int> futureB = Task.Factory.StartNew<int>(() => F1(a));
```

- Get result by retrieve **.Result** member

```
int f = F4(futureB.Result, d);
```

- Access **.Result** may throw exceptions

TPL: Continuation Tasks



- Continuation tasks make the dependencies among futures

```
TextBox myTextBox = ...;

int futureB = Task.Factory.StartNew<int>(() => F1(a));
int futureD = Task.Factory.StartNew<int>(() => F3(F2(a)));

int futureF = Task.Factory.ContinueWhenAll<int, int>(
    new[] { futureB, futureD },
    (tasks) => F4(futureB.Result, futureD.Result));

futureF.ContinueWith((t) =>
    myTextBox.Dispatcher.Invoke(
        (Action) (() => { myTextBox.Text = t.Result.ToString(); })))
    );
```

Parallel programming in .NET



Parallel Language Integrated Query (PLINQ)

declarative approach

Implemented on top of LINQ and TPL

Task Parallel Library (TPL)

Data- & Task-Parallelism (imperative approach)

Base Class Library
including .NET Threading Library

Common Language Runtime (CLR)

Win32 Application Programming Interface (API)

Operating system (Windows)

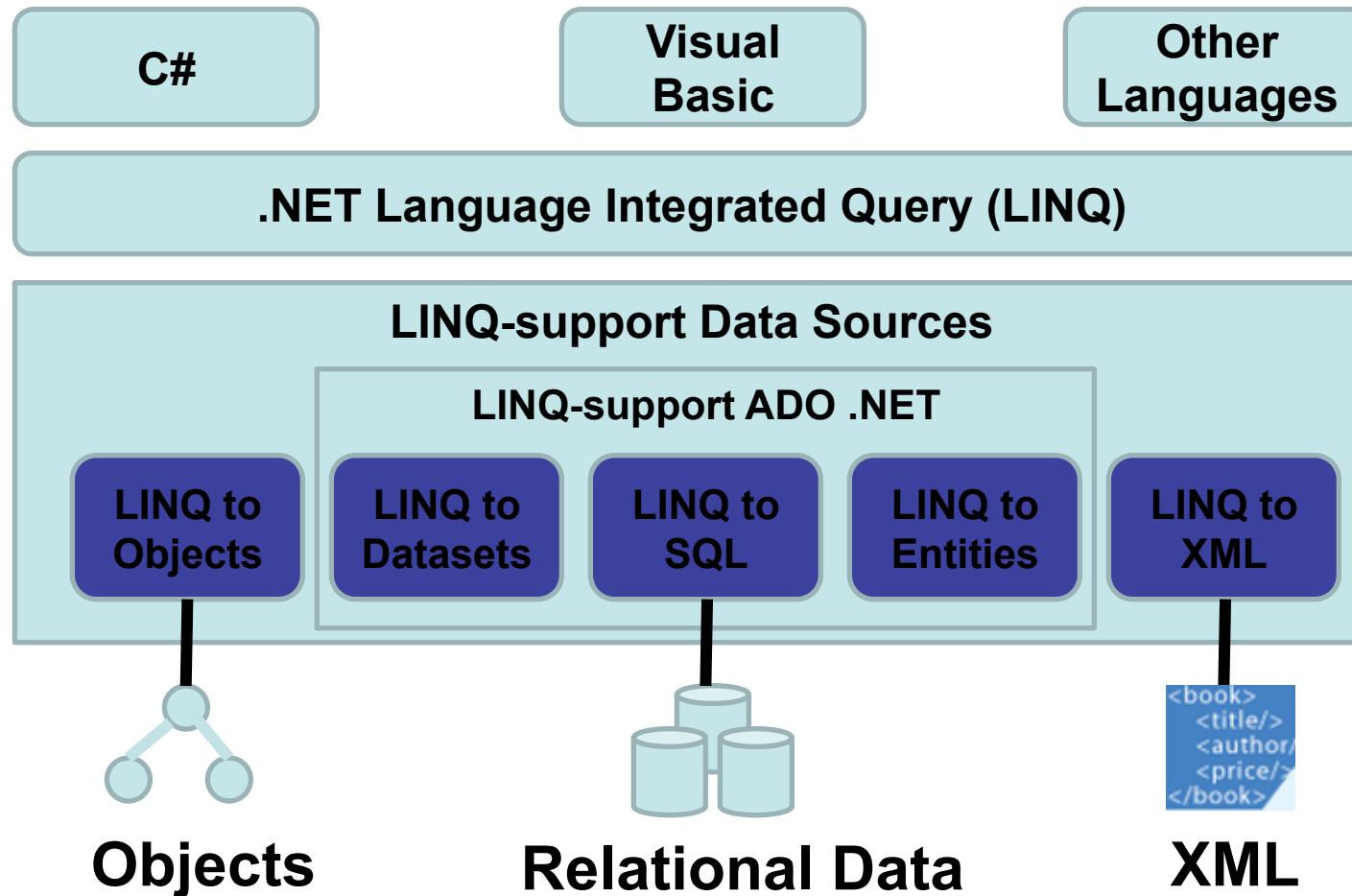
Overview of LINQ



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Combination of different programming paradigms
- Declarative query as integrated component of programming languages (e.g., C#, VB)
 - Specify “what should be done”, not “how”
- Operation for traversal, filtering, map, projection, ...
 - Applicable on data-based arrays, XML, SQL, ...
 - Can be replaced by your own implementations

Overview of LINQ (2)



Example: LINQ in C# (1/2)



```
using System;
using System.Query;
using System.Collections.Generic;

class app {
    static void Main() {
        string[] names = { "Burke", "Connor", "Frank", "Everett",
                           "Albert", "George", "Harris", "David" };

        IEnumerable<string> expr = from s in names where s.Length == 5
                                    orderby s
                                    select s.ToUpper();
    }
}
```

- Lazy execution: query is not executed until the enumerable object is used.

Declarative query on array-type data

Outputs:
BURKE
DAVID
FRANK



Example: LINQ in C# (2/2)

```
using System;
using System.Query;
using System.Collections.Generic;

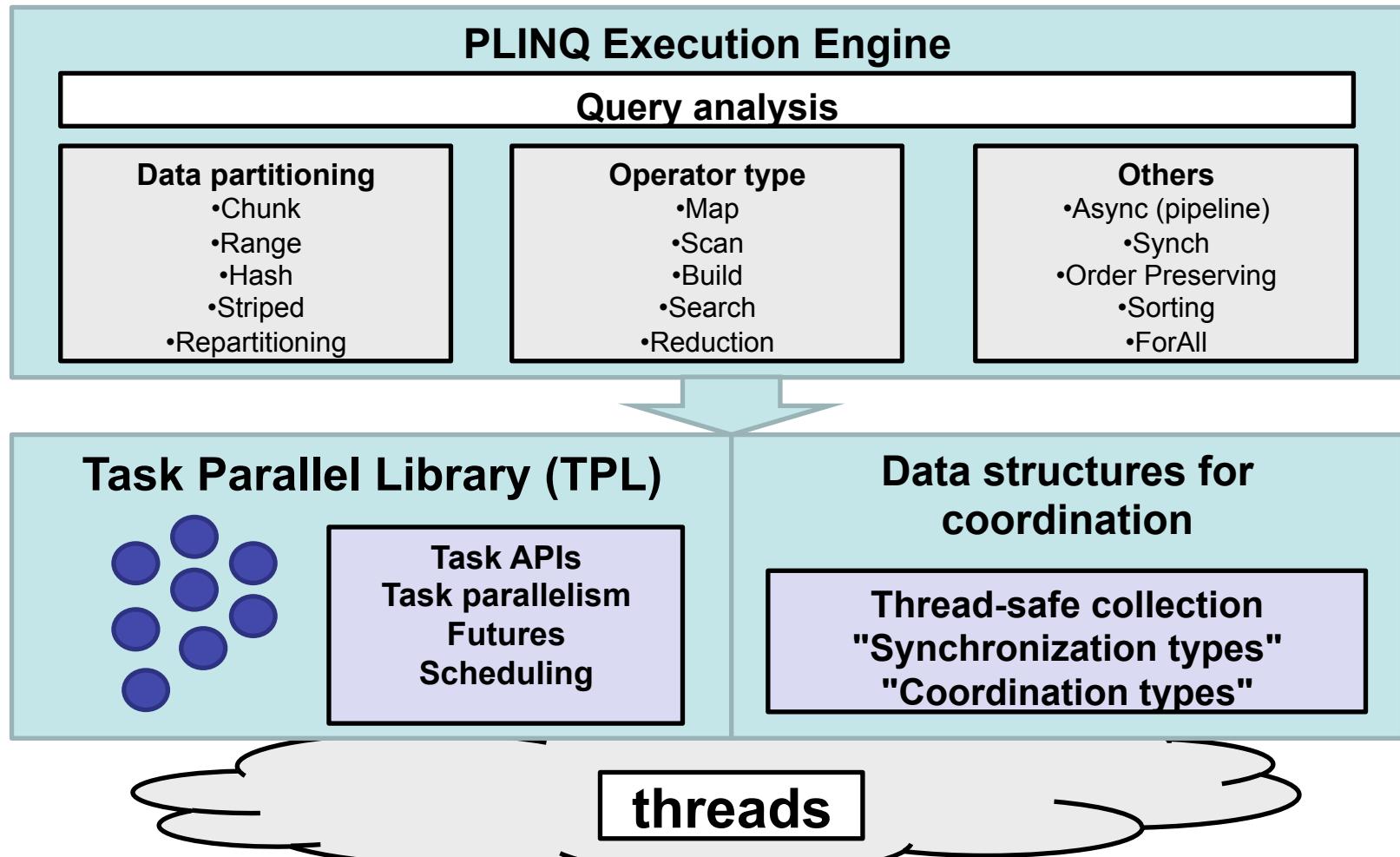
class app {
    static void Main() {
        string[] names = { "Burke", "Connor", "Frank", "Everett",
                           "Albert", "George", "Harris", "David" };

        IEnumerable<string> expr = names.Where(s => s.Length == 5)
            .OrderBy(s => s)
            .Select(s => s.ToUpper());

        foreach (string item in expr)
            Console.WriteLine(item);
    }
}
```

Here Where, OrderBy, and Select are chained using dot (.). Arguments of Where, OrderBy, Select are in C#-notation of **Lambda expression**.

Parallel Language Integrated Query (PLINQ)





Example: PLINQ

```
var q = from p in people.AsParallel()
        where p.Name == queryInfo.Name &&
              p.State == queryInfo.State &&
              p.Year >= yearStart &&
              p.Year <= yearEnd
        orderby p.Year ascending
        select p;
```

- Runtime engine automatically execute code in parallel
- Internally, tasks are created (using TPL)
- Lazy execution (the same as LINQ)

Notes of PLINQ



- By default, order is not guaranteed in parallel execution

```
int[] data = new int[] { 0, 1, 2, 3 };
int[] data2 = (from x in data.AsParallel() select x * 2).ToArray();
// output can be {0,2,4,6}, or {4,6,0,2}, or other order
```

- Preserve order

```
int[] data = new int[] { 0, 1, 2, 3 };
int[] data2 = (from x in data.AsParallel(QueryOptions.PreserveOrdering)
               select x * 2).ToArray();
// usually slower than order is not preserved
```



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

OPENMP

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Introduction**
- Loop-level parallelism
- Alternative work-sharing mechanisms
- Synchronization

Introduction



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- What is OpenMP?
- Components of OpenMP
- Fork-join execution model
- Communication and data environment
- Parallelizing a simple loop



What is OpenMP?

Open specifications for Multi Processing

- Industry standard of a shared-memory programming interface
 - Set of compiler directives to describe parallelism in the source code
 - Library functions
 - Environment variables
- Works with C/C++ and Fortran
- Current specification
 - OpenMP 4.5 (November 2015)
 - <http://www.openmp.org>

Components of the OpenMP API



- **Directives**
 - Instructional notes to any compiler supporting OpenMP
 - Pragmas in C/C++
 - Source-code comments in Fortran
 - **Control structures** for expressing shared-memory parallelism
 - **Data environment constructs** for communicating between threads
 - **Synchronization constructs** for coordinating the execution of multiple threads
- **Runtime library functions**
 - Examine and modify parallel execution parameters
 - Synchronization
- **Environment variables**
 - Preset parallel execution parameters

Directive syntax



- OpenMP pragma (C/C++)

```
#pragma omp ...
```

- Directives ignored by non-OpenMP compiler
- Conditional compilation with preprocessor macro name

```
#ifdef _OPENMP
    iam = omp_get_thread_num();
#endif
```

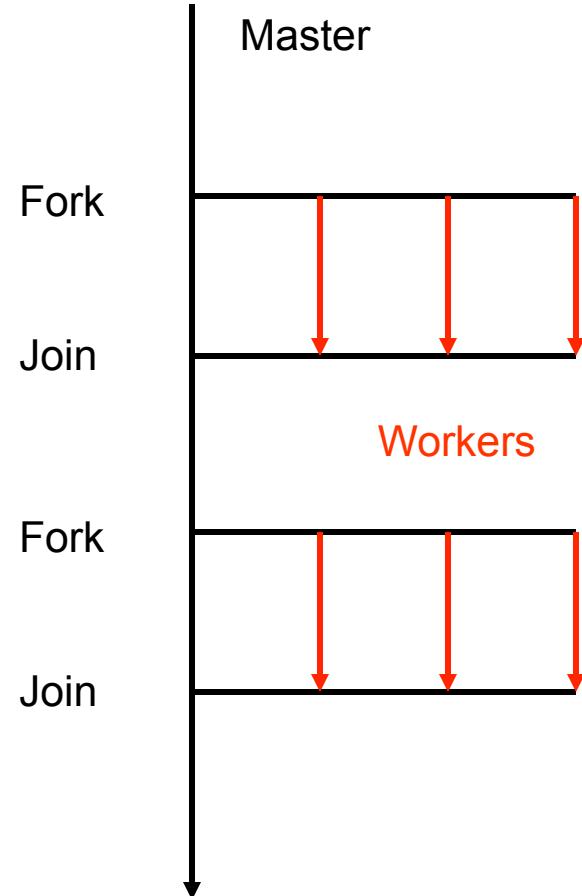


Fork-join execution model

- Programs begins execution as a single process
- This process is called the **master thread** of execution
- The master thread executes until the first parallel construct is encountered (i.e., a parallel directive)
- Then the master thread creates a **team** of threads
- The master becomes the master of the team
- The statements enclosed by the parallel construct are executed in parallel by each thread of the team
- Upon completion of the construct, the threads synchronize and only the master continues execution

Fork-join execution model (2)

- A number of parallel constructs can be specified in a program
- As a result, a program may fork and join many times



Extent of a parallel construct



- Construct = lexical extent of a construct
 - Statements lexically enclosed
- Region = dynamic extent of a construct
 - Also includes statements in routines called from within the construct
- Orphaned directives
 - Directives that do not appear in the construct but lie in the region
 - Allow execution of a program in parallel with only minimal changes to the serial version (incremental parallelization)

Parallel control structures



- Fork new threads or give execution control to one or another set of threads
- OpenMP has only a minimal set of control structures
- Three main functions:
 - Creating a team of threads executing concurrently
 - E.g., parallel directive
 - Dividing work among an existing set of parallel threads
 - Work-sharing constructs
 - E.g., loop-level parallelism
 - Generating a task
 - Task construct

Communication and data environment



- Each thread has data environment or execution context
 - Global variables
 - Automatic variables within subroutines (allocated on the stack)
 - Dynamically allocated variables (allocated on the heap)
- Execution context of the master thread exists for the entire duration of the program
- Execution context of a worker thread
 - Own stack
 - All other variables are either shared or private
 - Can be specified on a per-variable basis using data scope clauses

Communication and data environment (2)



- Shared semantics
 - All threads that access this variable will access the same storage location
 - Communication is expressed by reading from or writing to shared variables
- Private semantics
 - Multiple storage locations
 - One in each thread's execution context
 - Inaccessible to other threads
- Reduction variables
 - In between shared and private
 - Subject to an arithmetic operation at the end of a parallel construct

A simple loop



- Multiply add or saxpy (single-precision a^*x plus y)

```
void saxpy(double z[], double a, double x[], double y,
           int n)
{
    /* for simplicity, we have y as a scalar variable */

    int i;

    for ( i = 0; i < n; i++ )
        z[i] = a * x[i] + y;
}
```

- No dependences
 - Result of one iteration does not depend on results of others

Parallelizing a simple loop



```
void saxpy(double z[], double a, double x[], double y,
           int n)
{
    /* for simplicity, we have y as a scalar variable */
    int i;

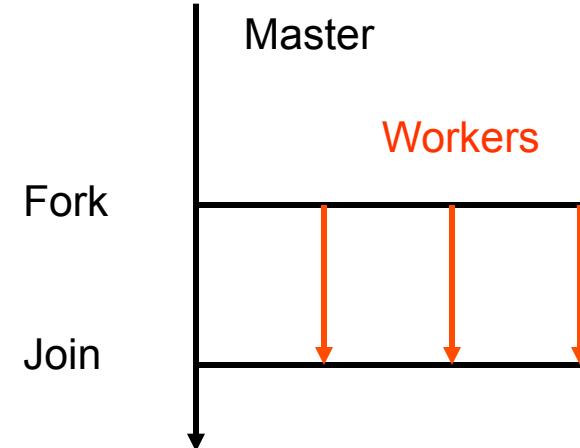
#pragma omp parallel for
    for ( i = 0; i < n; i++ )
        z[i] = a * x[i] + y;
}
```

- Only change is addition of the parallel for directive
- Must be followed by a for loop
- Specifies the concurrent execution of the loop
- Runtime system must create a set of threads and distribute iterations across the threads for parallel execution

Runtime behavior



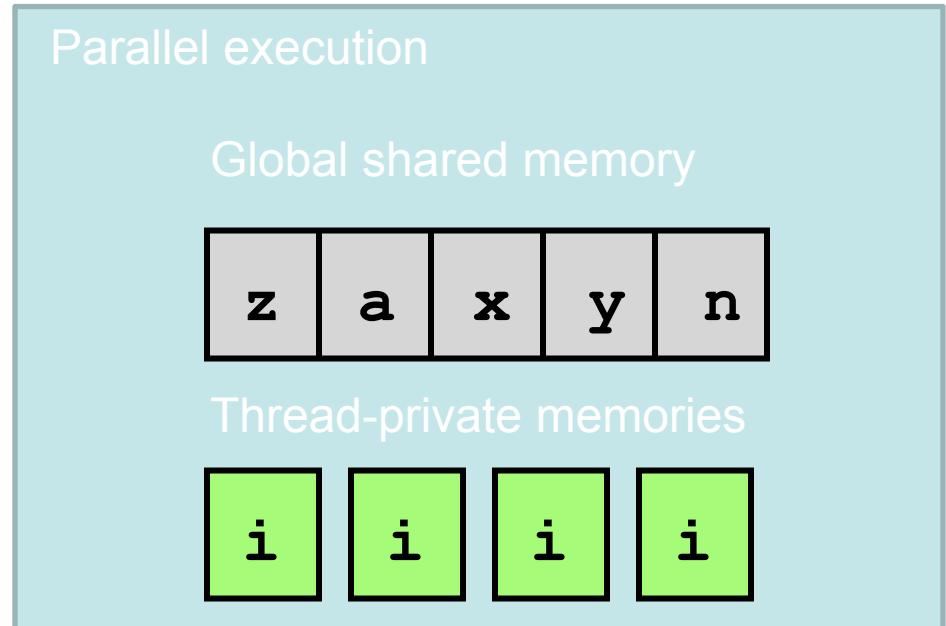
```
void saxpy(...)  
{  
    int i;  
#pragma omp parallel for  
    for ( i = 0; i < n; i++ )  
        z[i] = a * x[i] + y;  
}
```



- Master thread enters **saxpy()** function
- When hitting the parallel for directive, the master creates a team of threads
- Each thread executes a distinct subset of the iterations
- Distribution of iterations is not specified here

Communication and data sharing

- All variables or array elements in the previous example are never updated concurrently except for the loop index
- The loop index of the parallel for construct is private by default
- Each thread has a private copy of the loop index



Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Loop-level parallelism
- Alternative work-sharing mechanisms
- Synchronization
- Performance issues

Loop-level parallelism



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Loop-level constructs in OpenMP
- Data sharing
- Data dependences



Parallel do/for directive

- C/C++

```
#pragma omp parallel for [ clause [,] clause ] ... new-line
    for ( index = first ; test-expr ; incr-expr )
        loop-body
```

- Is actually short cut for

```
#pragma omp parallel new-line
{
    #pragma omp for [ clause [,] clause ] ... new-line
        for ( index = first ; test-expr ; incr-expr )
            loop-body
}
```

- Loop immediately following the directive is parallelized

- Data sharing attribute clauses
 - Control data sharing
- **schedule** clause
 - Controls distribution of work across the team of threads
- **if** clause
 - Specifies conditional parallelization
- **ordered** clause in combination with **ordered** constructs
 - Specifies execution order of ordered constructs to be the same as in serial execution
- **copyin** clause
 - Initializes certain kinds of private variables

Clauses (2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **collapse** clause
 - Specifies how many loops are associated with the loop construct
 - Can be used to collapse nested loop into one larger iteration space to be parallelized

Canonical loop structure



- Restrictions on loops to simplify compiler-based parallelization
- Allows number of iterations to be computed at loop entry
- Program must complete all iterations of the loop
 - C++/C: no **break** statement
 - C++: no exception thrown inside and caught outside the loop
- Exiting current iteration and starting next one possible
 - C/C++: **continue** allowed
- Termination of the entire program inside the loop possible
 - C/C++: **exit** allowed

```
for ( init-expr ; var log-op b ; incr-expr )
    loop-body
```

- Initialization

```
var = lb
integer-type var = lb
random-access-iterator-type var = lb
pointer-type var = lb
```

- Logical operator
- Increment expression

```
<, <=, >, >=
```

```
++var, var++, --var, var--,
var += incr, var -= incr,
var = var + incr,
var = incr + var,
var = var - incr
```

- Loop variable

- (Unsigned) integer or random access iterator (C++)
- Made private by default
- Unless specified lastprivate, its value is indeterminate after the loop

Runtime behavior



- Outside the loop a single master thread executes serially
- When reaching the loop, the master creates a team with zero or more worker threads
- The loop is concurrently executed by all threads
- Each iteration is executed only once
- Each thread may execute more than one iteration
- Variables are either shared or private to each thread
- Implicit barrier at the end of the loop

Parallel for and loop nest



- Parallel for directive refers only to the loop immediately following it unless collapse clause is specified

```
/* sum of a row */
#pragma omp parallel for
for ( i = 0; i < n; i++ ) {
    a[i][0] = 0;
    for ( j = 0; j < m; j++ )
        a[i][0] = a[i][0] + a[i][j];
}

/* smoothing function */
for ( i = 1; i < n; i++ ) {
#pragma omp parallel for
    for ( j = 1; j < m - 1; j++ )
        a[i][j] = ( a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] ) / 3.0;
}
```

Data sharing attribute clauses



- **shared** clause
 - Specifies shared semantics for a variable
- **private** clause
 - Specifies private semantics for a variable
- **firstprivate** clause
 - Initialization of private variables
- **lastprivate** clause
 - Finalization of private variables
- **default** clause
 - Changes default semantics of variables
- **reduction** clause
 - Specifies a reduction operation on a variable



Shared clause

- Variable is shared among all threads
- Single instance in shared memory
- All modifications update this single instance
- Caveat: pointers
 - Only pointer itself is shared but not necessarily the object it points to

Private clause

- Each thread allocates a private copy of the variable from storage within the thread's private execution context
- References within (the lexical extent of) the parallel construct read or write the private copy
- Value undefined upon entry of the parallel construct
 - Exceptions
 - Loop index variable
 - C++ class objects
- Value undefined upon exit of the parallel construct
- Size of the variable must be known to the compiler
 - No incomplete types in C/C++
 - No reference types in C++



Default sharing semantics

- Default shared
- In particular
 - Shared
 - All variables visible upon entry of the construct
 - Static C/C++ variables declared within the dynamic extent
 - Heap allocated memory
 - Private
 - Local variables declared within the dynamic extent
 - Loop index variable of the work-shared loop

Example



```
void caller(int a[], int n)
{
    int i, j, m = 3;

#pragma omp parallel for
    for (i = 0; i < n; i++) {
        int k = m;

        for (j = 1; j <= 5; j++)
            callee(&a[i], &k, j);
    }
}
```

```
extern int c = 4;

void callee(int *x, int *y,
            int z)
{
    int ii;
    static int cnt;

    cnt++;
    for (ii = 0; ii < z; ii++)
        *x = *y + c;
}
```

- Shared: a, n, j, m, *x, c, cnt
- Private: i, k, x, y, z, ii, *y
- Use of j and cnt is not safe

Changing default sharing semantics



- If most variables should have sharing semantics different from default, explicit sharing specification can become tedious
- Default clause
 - Fortran (**private** | **firstprivate** | **shared** | **none**)
 - C/C++ (**shared** | **none**)
- Only a single default clause per parallel directive
- Default shared
 - Is already the default
- Default none
 - All variables must be explicitly specified

Reduction operations



```
sum = 0;

#pragma omp parallel for reduction(+:sum)
for ( i = 0; i < n; i++ )
    sum = sum + b[i];
```

- Syntax
 - **reduction** (*op*: *var-list*)
- Applications
 - Compute sum of array
 - Find the largest element
- Operator should be commutative-associative
- Reduction variables are initialized to the identity element
- Reduction on array element via scalar temporary

Predefined reduction operators (C/C++)



Operator	Initial Value
+	0
*	1
-	0
&	all bits on
	0
^	0
&&	1
	0
max	Least representable value in the reduction list item type
min	Largest representable value in the reduction list item type

Private variable initialization and finalization



- Default avoids copying
 - Undefined initial value of private copy upon loop entry
 - Undefined value of master's copy after loop exit
- **firstprivate** clause
 - Initializes private copy to the value of master's copy prior to entering the construct
- **lastprivate** clause
 - Writes value of sequentially last iteration back to master's copy
- A variable can appear in both clauses
 - Exception to the rule that variable can appear in at most one clause

Data dependences



- Parallelization must preserve the program's correctness
- Data dependences may affect the program's correctness
 - Between output and input data
 - Among intermediate results
 - On the order in which loop iterations are executed
- How to detect data dependences?
- How to remove data dependences?



Data dependences (2)

- Data dependence between two accesses
 - Two accesses of the same memory location
 - At least one of them writes the location
- Location can be anywhere
 - Memory, file
- Example

```
for ( i = 1; i < n; i++ )
    a[i] = a[i] + a[i-1];
```

- Data dependences in parallel programs may cause a **race condition**
 - Result depends on the order in which certain statements are executed

Data dependences (3)



```
#pragma omp parallel for
for ( i = 1; i <= 2; i++ )
    a[i] = a[i] + a[i-1];
```

```
a[0] = 1;
a[1] = 1;
a[2] = 1;
```

Initial values

- $a[2]$ is computed using $a[1]$'s new value

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

- $a[2]$ is computed using $a[1]$'s old value

```
a[0] = 1;
a[1] = 2;
a[2] = 2;
```

Dependence detection



- Different loop iterations executed in parallel
- Same loop iteration executed in sequence
- Important for parallelization are loop-carried dependences
 - Dependence between different iterations of the same loop
- No dependence
 - If location is only read
 - If location is accessed in only one iteration
- Dependence
 - If location is accessed in more than one iteration and written in at least one of them

Dependence detection (2)



- Scalar variables are easy
 - Well-defined name
- Arrays more difficult
 - Array index may be computed at runtime
 - Find two different values i, j of the loop index variable within the index range such that iteration i writes some element and j reads or writes the same element
- Rule of thumb: loop can be parallelized if
 - All assignments are to arrays
 - Each element is assigned in at most one iteration
 - No iteration reads an element assigned by any other iteration

Flow dependences



- Two accesses A1 and A2 (often two statements) of the same storage location
- A1 comes before A2 in a serial execution of the loop
- A1 writes the location
- A2 reads the location
- Result of A1 flows to A2 → **flow dependence** (true dependence or RAW)
- The two accesses cannot be executed in parallel



Anti dependences

- A1 reads the location
- A2 writes the location
- Reuse of the location instead of communication through the location
- Opposite of flow dependence → **anti dependence** (WAR dependence)
- Parallelization
 - Give each iteration a private copy of the location
 - Initialize copy with value A1 would have read during serial execution

Output dependences



- Both A1 and A2 write the location
- Only writing occurs → **output** dependence (WAW dependence)
- Parallelization
 - Make location private
 - Copy sequentially last value back to shared copy at the end of the loop

Example



```
1   for ( i = 1; i < n - 1; i++ ) {  
2       x = d[i] + i;  
3       a[i] = a[i+1] + x;  
4       b[i] = b[i] + b[i-1] + d[i-1];  
5       c[2] = 2 * i;  
6   }
```

Memory location	Earlier access			Later access			Loop carried?	Kind of depend.
	Line	Iteration	r/w	Line	iteration	r/w		
x	2	i	write	3	i	read	no	flow
x	2	i	write	2	i+1	write	yes	output
x	3	i	read	2	i+1	write	yes	anti
a[i+1]	3	i	read	3	i+1	write	yes	anti
b[i]	4	i	write	4	i+1	read	yes	flow
c[2]	5	i	write	5	i+1	write	yes	output

Removing anti dependences



- Anti dependence on $a[i]$ and $a[i+1]$
- Also x read and written in different iterations
- Parallelization
 - Privatize x
 - Create temporary array $a2$
- Overhead
 - Memory
 - Computation

```
for ( i = 0; i < n - 1; i++ ) {  
    x = b[i] - c[i];  
    a[i] = a[i+1] + x;  
}
```

```
#pragma omp parallel for  
for ( i = 0; i < n - 1; i++ ) {  
    a2[i] = a[i+1];  
}  
  
#pragma omp parallel for private(x)  
for ( i = 0; i < n - 1; i++ ) {  
    x = b[i] - c[i];  
    a[i] = a2[i] + x;  
}
```

Removing output dependences



- Output dependence on $d[1]$
- Live-out locations $d[1], x$

```
for ( i = 0; i < n; i++ ) {  
    x = b[i] - c[i];  
    d[1] = 2 * x;  
}  
y = x + d[1] + d[2];
```

```
#pragma omp parallel lastprivate(x, d1)  
for ( i = 0; i < n - 1; i++ ) {  
    x = b[i] - c[i];  
    d1 = 2 * x;  
}  
d[1] = d1;  
y = x + d[1] + d[2];
```

- Parallelization via lastprivate temporary

Removing flow dependences



- A2 depends on result stored during A1
 - Dependence cannot always be removed
- Three techniques
 - Reduction operations
 - Induction variable elimination
 - Loop skewing

```
for ( i = 0; i < n; i++ ) {  
    x = x + a[i];  
}
```

- Reduction operations

```
#pragma omp parallel for reduction(+: x)  
for ( i = 0; i < n; i++ ) {  
    x = x + a[i];  
}
```



Removing flow dependences (2)

- Induction variable elimination
 - Special case of reduction operations
 - Value of reduction variable is simple function of loop index
 - Uses of the variable can be replaced by simple expression containing the loop index variable
- Loop skewing
 - Convert loop carried dependence into non-loop-carried one
 - Shift (“skew”) access to a variable between iterations

```
for ( i = 1; i < n; i++ ) {
    b[i] = b[i] - a[i-1];
    a[i] = a[i] + c[i];
}
```

- Loop-carried flow dependence from read of $a[i-1]$ to write of $a[i]$

Removing flow dependences (3)



- Parallelization using loop skewing

```
b[1] = b[1] + a[0];
#pragma omp parallel for shared(a, b, c)
for ( i = 1; i < n-1; i++ ) {
    a[i] = a[i] + c[i];
    b[i+1] = b[i+1] + a[i];
}
a[n-1] = a[n-1] + c[n-1];
```

Non-removable dependences



- Recurrences
 - Difficult or impossible
 - Recurrences in loop nests
 - Try to parallelize loop not involving the recurrence
- Fissioning
 - Splitting the loop into a parallelizable and non-parallelizable part
 - Parallelize only the parallelizable part of the loop
- Scalar expansion
 - Computation depends on scalar computed in each iteration
 - Compute array of all scalar values sequentially
 - Parallelize remaining part of the loop

```
for ( i = 1; i < n; i++ ) {  
    a[i] = (a[i-1] + a[i])/2;  
}
```

Scheduling



- The way loop iterations are distributed across the threads of a team is called **schedule**
- A loop is most efficient if all threads finish at about the same time
- Threads should do about the same amount of work (i.e., have the same load)
- If each iteration requires the same amount of work, an even distribution of iterations will be most efficient
- **Equal distribution** is **default** schedule of most implementations

```
#pragma omp parallel for
for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
```

Variable load per iteration



```
#pragma omp parallel for
for ( i = 0; i < n; i++ )
    if ( f(i) )
        do_big_work(i);
    else
        do_small_work(i);
```

- Even distribution of iterations may cause load imbalance
- Load imbalance causes synchronization delay at the end of the loop
- Faster threads have to wait for slower threads
- Total execution time will increase
- Choose alternate scheduling strategy



Static and dynamic scheduling

- Static scheduling
 - Distribution is done (deterministically) at loop-entry time based on
 - Number of threads
 - Total number of iterations
 - Index of an individual iteration
 - Less flexible
 - Low scheduling overhead
- Dynamic scheduling
 - Distribution is done during execution of the loop
 - Each thread is assigned a subset of the iterations at the loop entry
 - After completion each thread asks for more iterations
 - More flexible
 - Can easily adjust to load imbalances
 - More scheduling overhead (synchronization)



Scheduling strategies

- Distribution of iterations occurs in chunks
- Chunks may have different sizes
- Chunks are assigned either statically or dynamically
- There are different assignment algorithms
- Schedule clause
 - `schedule(type[, chunksizes])`
- Types
 - `static`
 - `dynamic`
 - `guided`
 - `auto`
 - `runtime`



Scheduling strategies (2)

- Static without chunk size
 - One chunk of iterations per thread, all chunks (nearly) equal size
- Static with chunk size
 - Chunks with specified size are assigned in round-robin fashion
 - “Interleaved” schedule
- Dynamic
 - Threads request new chunks dynamically
 - Default chunk size is 1
- Guided
 - First chunk has implementation-dependent size
 - Size of each successive chunk decreases exponentially
 - Chunks are assigned dynamically
 - ~~Chunk size specifies minimum size, default is 1~~

Scheduling strategies (3)



- Auto (no chunk size)
 - Gives implementation freedom to choose any possible mapping of iterations to threads
- Runtime
 - Scheduling strategy is determined by environment variable or API call
 - `setenv OMP_SCHEDULE "type[, chunksize]"`
 - `omp_set_schedule(...)`
 - Otherwise scheduling implementation dependent
- Correctness of program must not depend on scheduling strategy
 - Dependences might cause errors only under specific scheduling strategies
 - Dynamic scheduling might produce wrong results only occasionally



Comparison

- Static scheduling
 - Cheap
 - May cause load imbalance
- Dynamic scheduling
 - More expensive than static scheduling
 - Small chunk size increases cost
 - One synchronization per chunk
 - Small chunk size can balance the load better
- Guided scheduling
 - Number of chunks increases only logarithmically with the number of iterations
 - Most costly computation of chunk size
- Runtime
 - Allows testing of different scheduling strategies without recompilation

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Loop-level parallelism
- Alternative work-sharing mechanisms
- Synchronization

Alternative work-sharing mechanisms



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Parallel regions
- Work-sharing in parallel regions
- Tasking
- Nested parallelism
- Controlling parallelism

SPMD-style parallelism and work-sharing

- Loop-level parallelism is a local concept
- Program usually consists of multiple loops and non-iterative constructs
- Need to parallelize larger portions of a program
- Two different ways of parallelism
 - Parallel region construct provides SPMD-style replicated execution
 - SPMD = single-program multiple-data
 - Work-sharing constructs provide distribution of work across multiple threads
 - MPMD = multiple-program multiple-data



Parallel directive

- C/C++

```
#pragma omp parallel [ clause [[,] clause ] ...] new-line
structured-block
```

- Fundamental construct that starts parallel execution

Clauses on the parallel directive



- **private** (*variable-list*)
- **shared** (*variable-list*)
- **default(shared | none)**
- **firstprivate** (*variable-list*)
- **reduction** ({*op* | *intrinsic*} : *variable-list*)
- **if** (*scalar-logical-expression*)
- **copyin** (*variable-list*)
- **num_threads** (*integer-expression*)

Restrictions

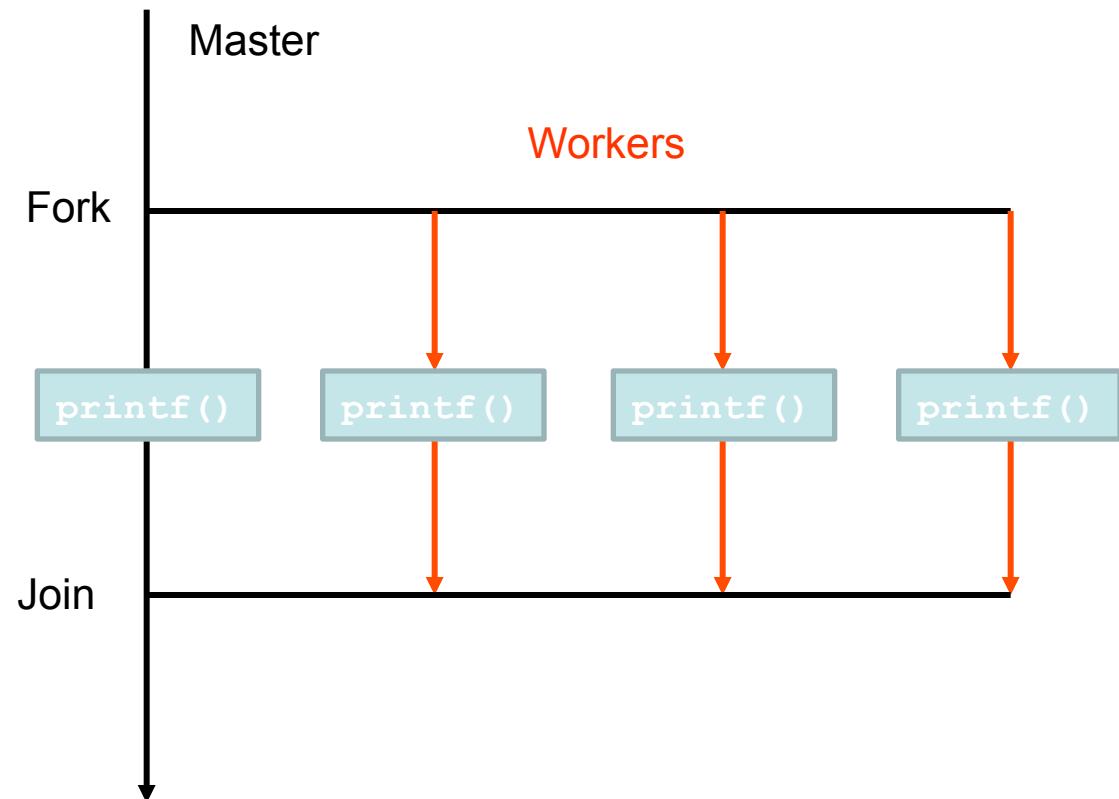
- Code encountered during a specific instance of the execution of a parallel construct is called parallel region
- Parallel region must be a structured block
 - One or more statements
 - Entered at the top, left at the bottom
 - No branches into/out of the parallel region
 - Branches within the parallel region permitted
 - Program termination within parallel region permitted
 - C/C++ `exit()`

Execution model of a parallel region



- Create a team of threads
- Each thread executes the code inside the parallel region
- Implicit barrier at the end
- Only master resumes execution

```
#pragma omp parallel  
printf("Hello world!\n");
```



Comparison between parallel and parallel for directive



- Parallel directive
 - n outputs per thread
 - Work **replication**

```
#pragma omp parallel
{
    int i;
    for ( i=0; i<n; i++)
        printf("Hello world!\n");
}
```

- Parallel for directive
 - n outputs total
 - Work **distribution**

```
int i;
#pragma omp parallel for
for ( i=0; i<n; i++)
    printf("Hello world!\n");
```

Lexical and dynamic extent revisited



- Lexical or static extent
 - Code lexically enclosed
 - Referred to as construct
- Dynamic extent
 - Lexcial extent plus code of subroutines executed as a result of the execution of statements within the lexical extent
 - Referred to as region
- Lexical extent is subset of dynamic extent
- Data-sharing clauses refer to the lexical extent

```
void subroutine();  
{  
    printf("Hello world!\n");  
}  
  
int main(int argc, char* argv[]){  
#pragma omp parallel  
{  
    subroutine();  
}  
}
```

Private clause applies only to construct



```
int my_start, my_end;

void work() {      /* my_start and my_end are undefined */
    printf("My subarray is from %d to %d\n",
           my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel private(my_start, my_end)
{
    /* get subarray indices */
    my_start = get_my_start(omp_get_thread_num(),
                           omp_get_num_threads());
    my_end   = get_my_end(omp_get_thread_num(),
                         omp_get_num_threads());
    work();
}
}
```

Passing private variables as arguments



```
int my_start, my_end;

void work(int my_start, int my_end) {
    printf("My subarray is from %d to %d\n",
           my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel private(my_start, my_end)
{
    my_start = [...]
    my_end   = [...]
    work(my_start, my_end);
}
}
```

- Cumbersome for long call paths

Threadprivate directive



```
int my_start, my_end;  
#pragma omp threadprivate(my_start, my_end)  
  
void work() {  
    printf("My subarray is from %d to %d\n",  
          my_start, my_end);  
}  
  
int main(int argc, char* argv[]) {  
#pragma omp parallel  
{  
    my_start = [...]  
    my_end   = [...]  
    work();  
}  
}
```



Threadprivate directive (2)

- Makes a variable private to a thread across the entire program
- Is initialized once at an unspecified point in the program prior to the first reference to that copy
- Value **persists across multiple parallel regions** if dynamic-threads feature has been disabled and the number of threads remains unchanged
 - Thread with same thread number will have same copy
- Thread-private variables cannot appear in any data-sharing clauses except for **copyin** or **copyprivate**
- Directive must be provided after the declaration of the variable
- C/C++: A directive must be provided for every declaration
 - Multiple translation units in C/C++
- Many special rules
 - See specification

The copyin clause



- Copies value of master copy to every thread-private copy at the entry to a parallel region
- A variable in a **copyin** clause must be a **threadprivate** variable

```
int c;
#pragma omp threadprivate(c)

int main(int argc, char* argv[]) {
    c = 2;
#pragma omp parallel copyin(c)
{
    /* c has value 2 in all thread-private copies */
    [...] = c;
}
}
```



Work sharing in parallel regions

- Four mechanisms:
 1. Task queue (manual)
 2. Domain decomposition
 - Assignment based on thread number and total number of threads
 3. Work-sharing constructs
 - For directive
 - Parallel sections
 - Single directive
 4. Task construct

Parallel task queue



- Shared data structure with a list of tasks
- All tasks can be processed concurrently, e.g.,
 - Rendering part of an image
- Threads in a team repeatedly request tasks

```
int main(int argc, char* argv[]) {
    int my_index;

#pragma omp parallel private(my_index)
{
    my_index = get_next_task();
    while ( my_index != -1 ) {
        process_task(my_index);
        my_index = get_next_task();
    }
}
}
```



Parallel task queue (2)

- Access to the queue needs to be synchronized

```
int index;

int get_next_task() {

#pragma omp critical
    if ( index == MAX_INDEX ) {
        return -1;
    } else {
        index++;
        return index;
    }
}
```

Thread number and number of threads



```
#pragma omp parallel private(nthreads, iam, chunk, start, end)
{
    nthreads = omp_get_num_threads();
    iam      = omp_get_thread_num();
    chunk   = (n + (nthreads - 1))/nthreads;
    start   = iam * chunk;
    end     = n < (iam + 1) * chunk ? n : (iam + 1) * chunk;
    for ( i = start; i < end; i++ )
        printf("");
}
```

- `omp_get_num_threads()` returns number of threads in the team
- `omp_get_thread_num()` returns individual thread identifier in $\{0, \dots, n-1\}$



Work-sharing constructs

- Manual distribution of work can be cumbersome
- Work-sharing constructs provide automatic distribution of work
 - Division of loop iterations
 - For directive
 - Distribution of distinct pieces of code
 - Sections directive
 - Identification of code that needs to be executed by a single thread only
 - Single directive



For directive

- C/C++

```
#pragma omp for [ clause [[,] clause] ...] new-line
    for ( index = first ; test-expr ; incr-expr )
        loop-body
```

- Divides iterations of the following loop among the threads in a team



Clauses of the `for` directive

- Data sharing attribute clauses
 - `private`
 - `firstprivate`
 - `lastprivate`
 - `reduction`
- `schedule` clause
 - Controls distribution of work across the team of threads
- `ordered` clause in combination with `ordered` construct
- Clauses above have same behavior as for parallel for
- `nowait` clause
 - Disables the implicit barrier at the end



Clauses of the for directive (2)

- **collapse** clause
 - Takes positive integer as parameter, which specifies how many nested loops are associated with the construct
 - If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space which is then divided according to the **schedule** clause
 - The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space

```
#pragma omp for collapse(2) schedule(static,1)
    for (i=0; i<2; i++)
        for (j=0; j<10; j++)
            a[i][j] = b[i][j];
}
```



Distributing a set of tasks

- Set of tasks in which none of the tasks depends on the results of others
- Example
 - Transformation of a data set into different output formats
- **Sections directive** provides a means to run the different tasks in parallel
- Coarse-grained parallelism as opposed to fine-grained parallelism
 - Non-iterative work-sharing construct
- Beneficial in particular if tasks are too small to benefit from parallelization themselves
- Execution order of tasks unknown



Sections directive

- C/C++

```
#pragma omp sections [ clause [[,] clause] ...] new-line
{
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line]
        structured-block]
    ...
}
```

- Noniterative work-sharing construct that contains a set of structured blocks that are divided among the threads in a team
- Each structured block is executed once by one of the threads in the team

Clauses of the sections directive



- Data sharing attribute clauses
 - **private**
 - **firstprivate**
 - **lastprivate**
 - **reduction**
- **nowait** clause
 - Disables the implicit barrier at the end
- Sections instead of iterations
 - First-private variables are initialized upon entry of the first section executed by each thread
 - Subsequent sections may see a different value
 - Last-private variables are assigned value computed by lexically last section
 - Reduction variables reflect status of private copies after completion of each thread's last section

Combined parallel/sections directive



- C/C++

```
#pragma omp parallel sections [ clause [[,] clause] ...] new-line
{
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line]
        structured-block
    ...
}
```

- Shortcut for a parallel region with a single sections construct



Suppressing replication

- Some tasks cannot be executed by multiple threads
 - Example: file I/O operations, MPI calls
- Single directive requires that enclosed code is executed by a single thread only
- C/C++

```
#pragma omp single [ clause [[,] clause] ...] new-line
structured-block
```

- Clauses can be
 - `private`, `firstprivate`, `copyprivate`, `nowait`



Write intermediate result to a file

- When reaching the single directive, an arbitrary thread is chosen to perform the operation
- Correctness must not depend on the selection of a particular thread

```
#pragma omp parallel
{
#pragma omp for
    for(i=0; i<n; i++)
        [...]
#pragma omp single
    write_intermediate_result();
#pragma omp for
    for(i=0; i<n; i++)
        [...]
}
```

- Remaining threads wait for the selected thread to finish the I/O operation at the implicit barrier unless there is a nowait

Copyprivate clause



```
double x;
#pragma omp threadprivate(x)

void init() {
    double a;
#pragma omp single copyprivate(a,x)
{
    a = [...];
    x = [...];
}
use_values(a,x);
}
```

- Broadcasts values acquired by a single thread directly to all instances of the private variables in the other threads
- Helpful if using a shared variable is difficult, e.g.
 - In recursion requiring a different variable at each level

Why is the single directive a work-sharing construct?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Enclosed code is executed exactly once, that is, it is not replicated
- Think of it as a sections directive with only one section
- It must be reached by all threads in a team
- All threads must reach all work-sharing constructs in the same order including the single directive
- Has implicit barrier
- Has nowait clause

Block structure and entry/exit of work-sharing constructs



- Contents of work-sharing constructs must have block structure
 - Complete statements
- Block must be entered at the top
 - No branching into the block (e.g. Fortran `goto`)
- Block must be left at the bottom
 - No branching out of the block (e.g., `return`)
- Branches within the block are allowed
- Termination of the program within a block is allowed
 - C/C++: `exit` allowed

Collective execution of work-sharing constructs



```
#pragma omp parallel
{
    if (omp_get_thread_num() != 0) /* illegal */
#pragma omp for
    for (i=0; i<n; i++)
        [...];
}
```

- Sequence of work-sharing constructs and barrier directives must be the same for every thread in a team
 - Every thread must participate in each work-sharing construct
 - If a thread reaches a construct then all the threads must reach it
 - If a thread executes multiple work-sharing constructs in a parallel region, then all the threads must execute them in the same order
 - If a work-sharing construct is skipped by one thread, it must be skipped by all the threads

Nesting of work-sharing constructs



```
#pragma omp for
    for (i=0; i<n; i++)
    {
#pragma omp for          /* illegal */
        for (j=0; j<m; j++)
        a = [...];
    }
```

- Nesting of work-sharing constructs is illegal in OpenMP
 - A thread executing inside a work-sharing construct executes its portion of work alone, therefore further division of work pointless
- Parallelization of nested loops
 - Manually
 - Nested parallel regions
 - Collapse clause
- No explicit barrier in work-sharing constructs

Orphaned work-sharing constructs



- Work-sharing constructs can occur anywhere in the dynamic extent of a parallel region
- Work-sharing constructs not in the lexical extent are called **orphaned** constructs
- User of function should be aware of work-sharing constructs

```
void initialize(double a[], int n) {
    int i;
#pragma omp for
    for (i=0; i<n; i++)
        a[i] = 0.0;
}

int main(int argc, char* argv[]) {

#pragma omp parallel
{
    initialize(a,n);
    [...]
}
}
```

Orphaned work-sharing constructs (2)



- Orphaned constructs reached from within a parallel region
 - Almost same behavior as when inside the lexical extent
 - Sharing clauses of the surrounding parallel region apply only to its lexical extent
 - Sharing semantics of variables in orphaned constructs may be different from those in the surrounding parallel region
 - C/C++ global variables are shared by default
- Orphaned constructs reached from within serial code
 - Almost same behavior as without work-sharing directive
 - Single serial thread behaves like a parallel team composed of only one master thread
 - Can safely be invoked from serial code – directive is essentially ignored

Tasking



- Introduced with OpenMP specification 3.0
- A task is a unit of work

```
#pragma omp task [clauses]
{
    task body
}
```

- Task might be executed by any thread of the parallel region
- Unspecified whether execution starts immediately or is deferred
- Tasks are executed in an undefined order unless the user specifies dependences

Implicit and explicit tasks



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- The body of the parallel region is considered as a task
 - It is called “implicit” task
- Tasks created by task constructs are called “explicit” tasks



Synchronization (barrier)

- A barrier can only complete if all explicit tasks are completed.
 - A parallel region can only complete if all explicit tasks are completed, due to its implicit barrier at the end
 - Calling a barrier inside an explicit task leads to a deadlock!

```
#pragma omp parallel
{
    #pragma omp task
    {
        #pragma omp barrier
    }
}
```

Deadlock



Synchronization (taskwait)

- The taskwait construct waits on all child tasks
 - It does not wait on grandchildren!

```
#pragma omp task          // Task A
{
    #pragma omp task      // Task B
    {
        #pragma omp task // Task C
        {
        }
    }
    #pragma omp taskwait
}
```

- The taskwait (Task A) waits on Task B, but **not on Task C**
 - Task C was not created by Task A



Synchronization (taskwait)

- Recursive taskwaits to ensure waiting for children

```
#pragma omp task          // Task A
{
    #pragma omp task      // Task B
    {
        #pragma omp task // Task C
        {
            }
        #pragma omp taskwait
    }
    #pragma omp taskwait
}
```

- Task A waits only on Task B
 - Task B can only complete if Task C is complete

Tied and untied tasks



Tied tasks

- Default mode
 - `#pragma omp task`
- Can be suspended at scheduling points
- The same thread that suspended it will resume the task
- The implicit task is always tied

Untied tasks

- Untied clause
 - `#pragma omp task untied`
- Can be suspended at scheduling points
- Any thread may resume a suspended task



Task scheduling

- Whenever a thread reaches a task scheduling point, it may perform a task switch
 - Beginning or resuming execution of different task
- Implied task scheduling points
 - Point immediately following task generation
 - After last instruction of task region
 - Implicit and explicit barriers
 - taskwait, taskgroup
- Explicit scheduling points
 - taskyield

taskyield



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Inserts an additional scheduling point
 - Allows the runtime system to interrupt current task at this point
- Does not wait for any task

```
#pragma omp taskyield
```

Data sharing attributes

- By default, variables inherited from creation context are `firstprivate`.
 - Caution: Default of parallel constructs is shared!
- The sharing attribute can be specified by explicit clauses
 - Supports the similar clauses as the parallel construct:
 - `shared` (variable list)
 - `private` (variable list)
 - `firstprivate` (variable list)
- Variables created inside a task are private

Data sharing example



```
#pragma omp parallel
{
    int a, b, c;
    #pragma omp task shared (a) firstprivate (b) // Task A
    {
        int d;
    }
    #pragma omp task shared (a,c)                  // Task B
    {
        int a, d;
    }
}
```

- a is shared among the implicit Task and Task A. In Task B the local a is private and overshadows the shared a
- b is firstprivate in all tasks
- c is firstprivate in Task A, but Task B shares c with the implicit task
- d is private to Task A and Task B



Example: Quicksort

- Sorting algorithm
- Input – array of length n
 - Data type with $<$ relation
- Output – array sorted in ascending order
- Based on the principle of divide & conquer

Quicksort – step 1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Select a pivot element pv
 - Common selection: Middle element or random

pv



Quicksort – step 2

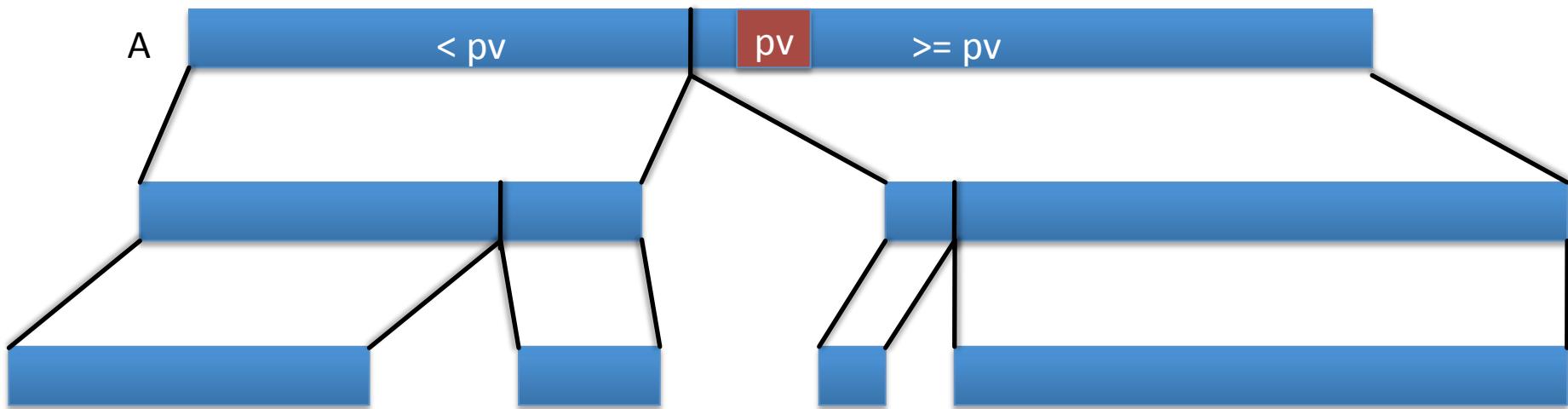
- Split array
 - Move all elements $< \text{pv}$ to the left side
 - Move all elements $\geq \text{pv}$ to the right side



- Find the last element $a < \text{pv}$
- Find the first element $b \geq \text{pv}$
- Swap a and b

Quicksort – recursion

- Sort both parts recursively
 - Recursion stops if only one element is left
 - Arrays with one element are trivially sorted



Quicksort – serial version



```
void quicksort( int* A, int length );
{
    if ( length <= 1 ) return;
    int pv = A[length/2];
    int forw = 0;
    int backw = length-1;
    while ( forw < backw )
    {
        while ( forw < backw && A[forw] < pv ) forw++;
        while ( forw < backw && A[backw] >= pv ) backw--;
        if ( A[forw] > A [backw] ) swap( A, forw, backw );
    }
    quicksort( A, forw );
    quicksort( &A[backw+1], length - backw - 1 );
}
```

Taskified Quicksort



```
void quicksort_task( int* A, int length );
{
    if ( length == 1 ) return;
    int pv = A[length/2];
    int forwa = 0;
    int backw = length-1;
    while ( forw < backw )
    {
        while ( forw < backw && A[forw] < pv ) forw++;
        while ( forw < backw && A[backw] >= pv ) backw--;
        if ( A[forw] > A [backw] ) swap( A, forw, backw );
    }

    #pragma omp task
    quicksort_task( A, forw );

    #pragma omp task
    quicksort_task( &A[backw+1], length - backw - 1 );
}
```

Parallel Quicksort



- Task directives take only effect when called inside a parallel region
- Use single construct to initiate the sorting only once
 - Otherwise, each thread would initiate the sorting.
- The implicit barrier at the end ensures that all tasks are completed when quicksort_task returns

```
// Assume A and length are already initialized
#pragma omp parallel
{
    #pragma omp single
        quicksort_task(A, length);
}
```

Quicksort synchronization



```
#pragma omp parallel
{
    // Assume A and length are already initialized
    #pragma omp single nowait
        quicksort_task(A, length);
    // When using A here: A is not yet sorted!
}
```

- When using tasks to perform work, you must synchronize before using the results!
 - Consider tasks inside function calls

```
#pragma omp parallel
{
    // Assume A and length are already initialized
    #pragma omp single nowait
        quicksort_task(A, length);
    #pragma omp barrier
    // You may use the sorted A now
}
```

Quicksort synchronization (2)



```
#pragma omp parallel
{
    // Assume A and length are already initialized
    quicksort_task(A, length);
    #pragma omp taskwait
    // When using A here: A is not yet sorted!
}
```

- Taskwait waits only for child tasks
- It does not wait for recursively created tasks!
- Need to call taskwait recursively

Quicksort with recursive taskwait



```
void quicksort_task( int* A, int length );
{
    if ( length == 1 ) return;
    int pv = A[length/2];
    int forwa = 0;
    int backw = length-1;
    while ( forw < backw )
    {
        while ( forw < backw && A[forw] < pv ) forw++;
        while ( forw < backw && A[backw] >= pv ) backw--;
        if ( A[forw] > A [backw] ) swap ( A, forw, backw );
    }
    #pragma omp task
    quicksort_task ( A, forw );
    #pragma omp task
    quicksort_task ( &A[backw+1], length - backw - 1 );
    #pragma omp taskwait
}
```

Task overhead



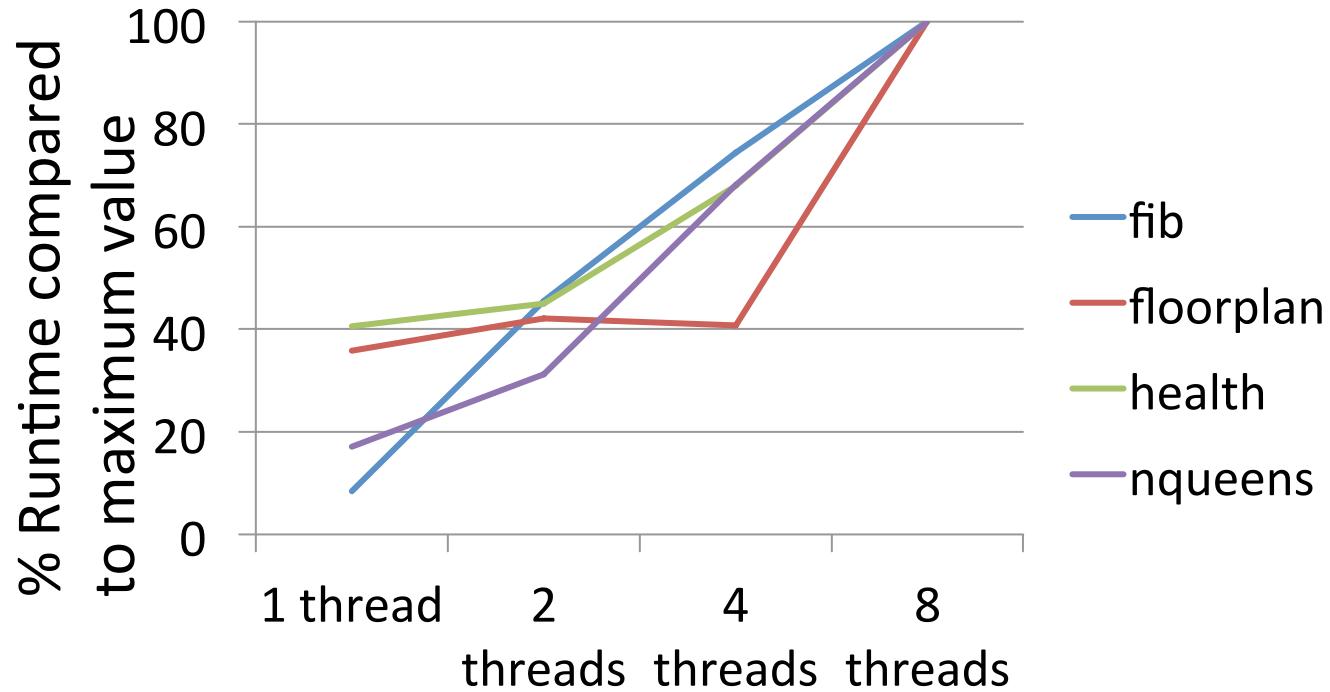
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Task creation and management incurs some overhead
 - Allocate memory for task data structures
 - Insert tasks in work pool
 - Synchronize access to the work pool for insertion and removal
 - Much less than thread creation

Too small tasks



- If tasks become too small, task management may become a performance problem
 - Especially, recursive algorithms tend to create lots of small tasks on deeper recursion levels



Examples with very small tasks



The if clause

- Goal: To avoid extensive overhead, create new task only if work exceeds a certain amount
- Solution: Use if clause:

```
#pragma omp task if ( condition )
{
    task body
}
```

- Task is only created if condition evaluates to true
- If the condition evaluates to false, the task body is executed inside the current task
 - No new task is created
 - Only one if-clause per task allowed

Quicksort with if clause



```
void quicksort_task( int* A, int length );
{
    if ( length == 1 ) return;
    int pv = A[length/2];
    int forwa = 0;
    int backw = length-1;
    while ( forw < backw )
    {
        while ( forw < backw && A[forw] < pv ) forw++;
        while ( forw < backw && A[backw] >= pv ) backw--;
        if ( A[forw] > A [backw] ) swap( A, forw, backw );
    }
    #pragma omp task if ( forw > MIN_VAL )
        quicksort_task( A, forw );
    #pragma omp task if ( length - backw - 1 > MIN_VAL )
        quicksort_task( &A[backw+1], length - backw - 1 );
    #pragma omp taskwait
}
```

final clause



```
#pragma omp task final (condition)
{
    task body
}
```

- If condition evaluates to true, the new task is a leaf
 - All task constructs appearing inside the task are ignored
 - Task body is executed as part of the current task
 - Purpose: optimization of if clause
 - If clause reevaluates the condition on every recursion level
 - Final clause avoids evaluation of condition in any further recursion level
- Usage difference between if clause and final clause
 - Final clause **creates no** further tasks if condition is true
 - **If clause creates new task** if condition is true

Quicksort with final clause



```
void quicksort_task( int* A, int length );
{
    if ( length == 1 ) return;
    int pv = A[length/2];
    int forwa = 0;
    int backw = length-1;
    while ( forw < backw )
    {
        while ( forw < backw && A[forw] < pv ) forw++;
        while ( forw < backw && A[backw] >= pv ) backw--;
        if ( A[forw] > A [backw] ) swap( A, forw, backw );
    }
    #pragma omp task final ( forw < MIN_VAL )
        quicksort_task( A, forw );
    #pragma omp task final ( length - backw - 1 < MIN_VAL )
        quicksort_task( &A[backw+1], length - backw - 1 );
    #pragma omp taskwait
}
```



Task dependences

- In many cases, a complete decomposition of a program into tasks would increase the scalability and adaptivity of the program to the available number of tasks
- However, often intermediate results computed by a task are needed by another task
 - To ensure the availability of intermediate results, it requires certain order of task execution
- Current synchronization mechanisms (barriers and taskwait) provide limited means to influence task execution order
 - Leads to (short) phases of task parallel execution followed by a synchronization phase
 - Limits use of tasks
 - Increases synchronization overhead
 - Reduces the amount of exploitable concurrency
- Solution: Add means to define dependences between tasks

Task dependences



Idea:

- Define which variables are read by the task
- Define which variables are written by the task
- If task A accesses a variable that was accessed by a formerly created sibling task B and one of the accesses is a write, then A depends on B
- Runtime guarantees a data-race-free execution order of tasks if dependences are correctly specified

Task dependences

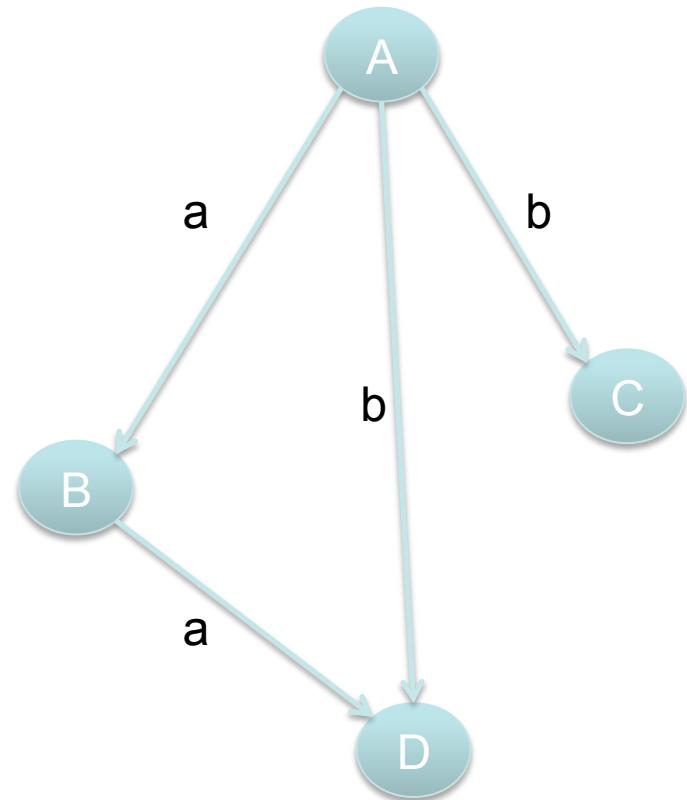


```
#pragma omp task depend(in|out|inout:<variablelist>)
```

- in: Defines the variables that are read by the task
- out: Defines the variables that are written by the task
- inout: Defines the variables that are read and written by the task

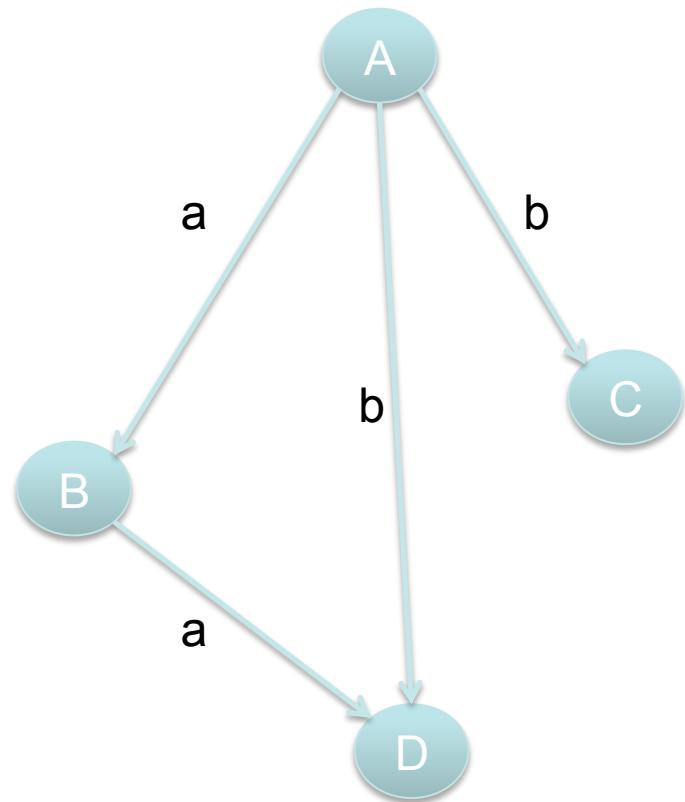
Task dependences example

```
int a, b;  
#pragma omp task depend(out:a,b)  
    shared(a,b) // task A  
{ }  
#pragma omp task depend(inout:a)  
    shared(a) // task B  
{ }  
#pragma omp task depend(in:b)  
    shared(b) // task C  
{ }  
#pragma omp task depend(in:a,b)  
    shared(a,b) // task D  
{ }
```



Execution order

- A must be completed before B, C, and D
- B must be completed before D
- B and C can be computed in parallel
- C and D can be computed in parallel



Dependences



- Dependences exist only between siblings
 - Siblings are tasks that are created by the same parent task

```
#pragma omp task
{
    #pragma omp task depend(out:a) // Task A
    {
    }
}
#pragma omp task depend(in:a) // Task B
{
}
```

- No dependence between A and B
 - A is not a sibling of B

Taskgroup



- Recursive task creation is a common pattern
- Need to wait for the completion of all recursively created tasks
 - taskwait does only wait for direct children
 - barrier waits for all tasks
 - Not usable in explicit tasks
- Solution: taskgroups

```
#pragma omp taskgroup
structured block
```

- At the end of the structured block:
 - Wait for all tasks created inside the structured block **and their descendants**

Nested parallel regions



- So far only work-sharing constructs within parallel regions
- Parallel region nested inside another parallel region
 - Behaves similar to parallel region inside serial code
 - Creates a team of threads for every thread reaching it
 - Parallel region is serialized by default
 - Team contains only one thread with thread number zero (i.e., master)
- All work-sharing constructs and the task construct bind to the closest enclosing parallel directive
- Synchronization constructs barrier and master bind to the closest enclosing parallel directive
- If enclosing parallel region is serialized, enclosed work-sharing constructs behave as if executed in parallel with only a single thread

Nested parallel regions (2)



```
void process_task(int k) {
    int i;
#pragma omp parallel for
    for (i=0; i<n; i++)
        a[k][i] = 3.0;
}

int main(int argc, char* argv[]) {
    int my_index;

#pragma omp parallel private(my_index)
{
    my_index = get_next_task();
    while (my_index != -1) {
        process_task(my_index);
        my_index = get_next_task();
    }
}
}
```



Nested parallel regions (3)

- Control nested parallelism:
 - **omp_set_nested(*value*)**
 - If value is zero, then nested parallel regions are serialized and executed with only a single thread
 - If value is non-zero then implementation may choose to spawn more than one thread inside nested parallel region
 - Number of threads is implementation defined
 - Implementations are still allowed to serialize
 - **omp_get_nested()**
- Environment variable
 - **setenv OMP_NESTED TRUE/FALSE**

Nested parallel regions (4)



- Query functions
 - `omp_get_level()`
 - Returns the number of nested parallel regions enclosing the task that contains the call
 - `omp_ancestor_thread_num(level)`
 - Returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread
 - `omp_get_team_size(level)`
 - Returns the size of the thread team to which the ancestor or the current thread belongs
 - `omp_get_active_level()`
 - Returns the number of nested, active parallel regions enclosing the task that contains the call

Active parallel region - a parallel region that is executed by a team consisting of more than one thread

Nested parallel regions (5)



- Controlling the number of nested active parallel regions
 - `omp_set_max_active_levels (level)`
 - Limits the number of nested active parallel regions
 - `omp_get_max_active_levels ()`
 - Returns the maximum number of nested active parallel regions
 - Environment variable `OMP_MAX_ACTIVE_LEVELS`

Controlling parallelism



- Dynamically disabling parallel directives using if clause
- Check for parallel execution with `omp_in_parallel()`
 - Returns a non-zero value if program is executing inside an active parallel region
- Controlling the number of threads for all parallel regions
 - `setenv OMP_NUM_THREADS n`
 - If set before program start, the application will create teams of the given size
 - Can be specified as a list to define different values for different nesting levels
 - `omp_set_num_threads(n)`
 - Adjustment of the team size at runtime
 - Affects only subsequent parallel regions



Controlling parallelism (2)

- Controlling the number of threads for a particular parallel region

- `numthreads` clause

```
#pragma omp parallel for numthreads(4)
for (i=0; i<16; i++)
[...]
```

- Can be used to request a certain number of threads
- Priorities among the different mechanisms
 - `numthreads`
 - `omp_set_num_threads(n)`
 - `OMP_NUM_THREADS`
- Determining the number of CPUs available to the program
 - `omp_get_num_procs()`
- Adjust number of threads to number of available CPUs
 - `omp_set_num_threads(omp_get_num_procs())`

Controlling parallelism (3)



- Current team size
 - `omp_get_num_threads()`
- Maximum number of threads
 - `omp_get_max_threads()`
 - Returns a number at least as large as the team size would be for a parallel region without `numthreads` clause
- Maximum number of threads in the program
 - `omp_get_thread_limit()`
 - `OMP_THREAD_LIMIT`



Controlling parallelism (4)

- Wait policy
 - **OMP_WAIT_POLICY = ACTIVE | PASSIVE**
 - Specifies whether waiting threads should mostly be active, consuming processor cycles, while waiting or mostly be passive, not consuming processor cycles, while waiting
 - For example, an OpenMP implementation may, for example, make waiting threads spin or go to sleep
- Binding threads to processors
 - Via environment variable **OMP_PROC_BIND** or **proc_bind** clause of parallel construct

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Loop-level parallelism
- Alternative work-sharing mechanisms
- Synchronization

Synchronization



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Introduction

Mutual-exclusion synchronization

Event synchronization

Custom synchronization

Summary

Introduction



- Implicit communication in shared-memory programming
 - Read/write operations on variables in shared address space
 - Requires coordination (i.e., synchronization)
- Three types
 - Mutual exclusion synchronization
 - Critical sections
 - Atomic directive
 - Runtime-library lock routines
 - Event synchronization
 - Barriers
 - Ordered sections
 - Master directive
 - Flush directive
 - Depend clause of task directive / taskwait directive
 - Customized (ad-hoc) synchronization using shared-memory variables

Synchronization mechanisms in OpenMP



- Mutual exclusion
 - Exclusive access to a shared data structure
 - Can be used to ensure
 1. Only one thread has access to the data structure for the duration of the synchronization construct
 2. Accesses by multiple threads are interleaved at the granularity of the synchronization constructs
- Event synchronization
 - Signals the completion of some event from one thread to another
 - Event synchronization can be used to implement ordering between threads
 - Mutual exclusion does not control the order in which a shared data structure is accessed

OpenMP synchronization constructs



- Mutual exclusion
 - Critical directive – implements critical sections
 - Atomic directive – efficient atomic update of a **single memory** location
 - Runtime library lock routines – customized synchronization
- Event synchronization
 - Barrier directive – classical barrier synchronization
 - Ordered directive – imposes sequential order on the execution of the enclosed code section
 - Master directive – code that should be executed only by the master thread
 - Flush directive – enforces memory consistency
 - Taskwait / taskgroup directive – waits on completion of task(s)



Critical section directive

- C/C++

```
#pragma omp critical [ (name) ] new-line
structured-block
```

- Critical section provides mutual exclusion with respect to all critical sections in the program with the same (unspecified) name



Critical section directive (2)

```
    cur_max = MINUS_INFINITY;  
#pragma omp parallel for  
    for ( i = 0; i < n; i++ ) {  
#pragma omp critical  
    {  
        if ( a[i] > cur_max )  
            cur_max = a[i];  
    }  
}
```

- When encountering the directive, a thread waits until no other thread is executing inside a critical section
- No branches in/out of a critical section allowed
- No fairness guaranteed
- Forward progress guaranteed
 - Eligible thread will always get access

Preliminary tests



```
cur_max = MINUS_INFINITY;  
#pragma omp parallel for  
    for ( i = 0; i < n; i++ ) {  
        if ( a[i] > cur_max )  
#pragma omp critical  
        {  
            if ( a[i] > cur_max )  
                cur_max = a[i];  
        }  
    }
```

- Previous example effectively serialized
- We can take advantage of the following observations
 - `cur_max` is monotonically increasing
 - Often most of the iterations only read but never write `cur_max`
- We assume that `cur_max` is a scalar type and written atomically

Atomic directive



- Many systems provide hardware instructions supporting the atomic update of a single memory location
 - Load-linked, store-conditional (LL/SC) on MIPS
 - Compare-and-exchange (CMPXCHG) on Intel x86
- Instructions have exclusive access to the location while performing the update
 - No additional locking required
 - Therefore higher performance
- OpenMP atomic directive can give programmer access to these atomic hardware operations
- Alternative to critical directive, but only in a limited set of cases

Atomic directive



```
#pragma omp atomic [read | write | update | capture ]  
    [seq_cst] new-line  
expression-statement
```

or

```
#pragma omp atomic capture [seq_cst] new-line  
structured-block
```

- Permitted form of expression statement depends on clause
- Structured block is short sequence of statements to update and read a value or vice versa (details in standard)

Expression statement



If clause is **read**

```
v = x;
```

If clause is **write**

```
x = expr;
```

If clause is **update** or not present

```
v = x++;
v = ++x;
v = x--;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

If clause is **capture**

```
x++;
++x;
x--;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

Corresponding structured block forms

```
{v = x; x++;
{v = x; ++x;
{++x; v = x;
{x++; v = x;
{v = x; x--;
{v = x; --x;
{--x; v = x;
{x--; v = x;
```



Atomic directive (2)

- The atomic construct with the read clause forces an atomic read
- The atomic construct with the write clause forces an atomic write
- The atomic construct with the update clause forces an atomic update
 - Note that when no clause is present, the semantics are equivalent to atomic update
- The atomic construct with the capture clause forces an atomic update of the location while also capturing the original or final value with respect to the atomic update
- Any atomic construct with a **seq_cst** clause forces the atomically performed operation to include an **implicit flush** operation without a list



Atomic directive example

- Calculating a histogram

```
#pragma omp parallel for
for ( i = 0; i < n; i++ )
{
#pragma omp atomic
    hist[a[i]] += 1;
}
```

- Advantage of using the atomic directive
 - Multiple updates to **different locations** can occur concurrently
 - However, false sharing is possible if multiple shared variables reside on the same cache line



Atomic vs. critical directive

- Atomic execution of a single update
 - Atomic usually never worse than critical directive
- Some implementations may choose to implement the atomic directive using a critical section
- Atomic execution of multiple updates
 - Would require multiple atomic directives
 - Critical section
 - One synchronization (+)
 - Atomic directive
 - Allows overlap (+)
 - Usually smaller synchronization overhead for a single critical construct
- Guideline
 - Atomic directive to update a single or a few locations
 - Critical directive to update several locations
- Do not mix critical and atomic directives for synchronizing conflicting accesses

Runtime library lock routines



- Functionality
 - Lock initialization
 - Lock testing
 - Lock acquisition
 - Lock release
 - Lock destruction
- More flexible
 - No block structure required
 - Lock variable can be determined dynamically
 - Allows doing computation while waiting
 - Allows nestable locks

Runtime library lock routines example



```
omp_lock_t lock;

omp_init_lock(&lock);

cur_max = MINUS_INFINITY;
#pragma omp parallel for
for ( i = 0; i < n; i++ ) {
    if ( a[i] > cur_max )
    {
        omp_set_lock(&lock);
        if ( a[i] > cur_max )
            cur_max = a[i];
        omp_unset_lock(&lock);
    }
}
omp_destroy_lock(&lock);
```

Working while waiting



```
omp_lock_t lock;

omp_init_lock(&lock);

cur_max = MINUS_INFINITY;
#pragma omp parallel for
for ( i = 0; i < n; i++ ) {
    if ( a[i] > cur_max )
    {
        while (!omp_test_lock(&lock))
            do_work();
        if ( a[i] > cur_max )
            cur_max = a[i];
        omp_unset_lock(&lock);
    }
}
omp_destroy_lock(&lock);
```



Lock functions summary

- **void omp_init_lock(omp_lock_t*)**
 - Initialize lock
- **void omp_destroy_lock(omp_lock_t*)**
 - Destroy lock
- **void omp_set_lock(omp_lock_t*)**
 - Waits until lock becomes available and acquires it
- **int omp_test_lock(omp_lock_t*)**
 - Returns zero if lock unavailable
 - Acquires it if it becomes available
- **void omp_unset_lock(omp_lock_t*)**
 - Releases lock and causes waiting thread to resume

Barrier directive



- C/C++

```
# pragma omp barrier new-line
```

- Synchronizes the execution of all threads in a parallel region
- All the code before the barrier must have been completed by all the threads before any thread can execute any code past the barrier

Barrier directive (2)



```
#pragma omp parallel private(index)
{
    index = generate_next_index();
    while ( index != 0 ) {
        add_index(index);
        index = generate_next_index();
    }
#pragma omp barrier
    index = get_next_index();
    while ( index != 0 ) {
        process_index(index);
        index = get_next_index();
    }
}
```

- All threads executing the parallel region must execute the barrier
- Cannot be placed inside work-sharing constructs
- Implicit barrier at the end of work-sharing constructs if there is no nowait clause

In a serialized parallel region the barrier is trivially complete when the thread arrives at the barrier

Ordered directive



- Imposes sequential order on the iterations of a parallel loop
- Must be in the dynamic extent of a (parallel) for construct
- The (parallel) for to which it binds must have ordered clause
- Each iteration must not execute the ordered directive more than once and must not execute more than one ordered directive
- C/C++

```
#pragma omp ordered new-line
structured-block
```



Ordered directive - Example

- Output can maintain its original order
- Computation can be done in parallel

```
#pragma omp parallel for ordered
    for ( i=0; i<n; i++ ) {
        a[i] = f(i);
#pragma omp ordered
        printf("a[%d] = %d", i, a[i]);
    }
```

Master directive



- The code in a parallel region is executed in a replicated fashion
- Sometimes we want to have a particular code section to be executed by the master only
- C/C++

```
#pragma omp master new-line
structured-block
```

- Not all threads need to reach the master directive
- No implicit barrier at the end



Master directive - Example

- Only the master is allowed to execute the function `print_result()`

```
#pragma omp parallel
{
#pragma omp for
    for ( i=0; i<n; i++ ) {
        /* computation */
    }
#pragma omp master
    print_result();
}
```



Memory consistency

- **Example:** Producer / consumer

Producer

```
data = ...;  
done = true;
```

Consumer

```
while (!done) {}  
... = data;
```

- Code assumes that read / write operations occur strictly in this order
- The problem is that compiler / hardware
 - Might allocate some of the variables in a register and thus delay update of the corresponding memory location
 - Reorder the memory references
 - Note: these transformations are necessary to obtain high performance
- Important for synchronization through memory references
- Usually not a problem in most codes

Memory consistency (2)



- Refers to the ordering of accesses to different memory locations, observable from various threads in the system
 - When must a processor see a value that has been updated by another processor?
 - In what order does a processor observe the data writes of another processor?
- Only way to observe writes are reads
 - Which properties must be enforced among reads and writes to different locations by different processors?

Memory consistency - Example



```
T1:      A = 0;
```

```
...
```

```
    A = 1;
```

```
L1:      if (B == 0) ...
```

```
T2:      B = 0;
```

```
...
```

```
    B = 1;
```

```
L2:      if (A == 0) ...
```

- Assumption
 - The threads are running on different processors
 - A and B are originally cached by both processors with the initial value of 0
 - If all writes take immediate effect and are immediately seen by other processors, it is impossible for both if statements to evaluate as true
 - What happens if write invalidate is delayed?

Sequential consistency



The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

- Simple implementation
 - Delay completion of any memory access until all the invalidations caused by that access are completed
- Advantage
 - Simple programming paradigm
- Disadvantage
 - Potential performance degradation



Relaxed consistency models

- Allow reads and writes to complete out of order
 - Use synchronization primitives to enforce ordering
- Relaxed models can be distinguished by the orderings they relax
 - $X \rightarrow Y$ with X, Y in {R, W}
- **Sequential consistency**
 - Relaxes none
- **Total store or processor consistency**
 - Relaxes $W \rightarrow R$, retains ordering among writes
- **Partial store order**
 - Relaxes $W \rightarrow W$
- **Weak ordering or release consistency**
 - Relaxes $R \rightarrow W$ and $R \rightarrow R$

Relaxed consistency models (2)

- Most multiprocessors support some sort of relaxed consistency model
- Most programs are therefore synchronized programs based on predefined synchronization primitives
 - Built on top of basic hardware primitives
 - e.g., load linked / store conditional



Temporary view of memory

- Shared memory with relaxed consistency:
- All threads have access to **memory**
- Each thread is allowed to have its own **temporary view** of memory
 - Not required to be consistent with memory at all times
 - A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time
 - A read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory

Flush directive



C/C++

```
#pragma omp flush [ (var-list) ] new-line
```

- Enforces consistency between temporary view and memory and restricts reordering of memory operations
 - Cross-thread sequence point (“memory barrier”)
 - Previous evaluations completed
 - Subsequent evaluations haven’t begun yet
- Applied to a set of variables called **flush-set**



Flush directive (2)

- A flush construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program (as defined by the base language) is flushed
- If a thread has captured the value of a write in its temporary view, a flush of that variable does not complete until the variable has been written to memory
 - Temporary view of the variable is discarded
 - No later memory operation by that thread for a variable involved in the flush is allowed to start before completion of flush

Typical usage sequence



1. The value is written to the variable by the first thread
2. The variable is flushed by the first thread
3. The variable is flushed by the second thread
4. The value is read from the variable by the second thread



Flush directive (3)

- Flush without variable list implied for
 - Barrier region
 - Entry/exit parallel, critical, ordered regions
 - Exit from work sharing regions, unless nowait is present
 - Entry/exit combined work-sharing regions
 - Set/unset lock
 - Test lock and set/unset/test nested lock if lock is actually set or unset
 - Immediately before and immediately after every task scheduling point
- Flush with variable list implied for
 - Entry/exit atomic regions where the list contains only the object updated in the atomic construct
- Not implied for
 - Entry to work-sharing regions
 - Entry/exit master region



Flush directive - Example

- Producer-consumer pattern

Thread 1

```
a = foo();  
#pragma omp flush  
flag = 1;  
#pragma omp flush
```

Thread 2

```
#pragma omp flush  
while(!flag){  
#pragma omp flush  
}  
#pragma omp flush  
b = a;
```

- **Thread 1:** First flush ensures `flag` is written after `a`. Second flush ensures `flag` is written to memory
- **Thread 2:** First and second flushes ensure `flag` is read from memory. Third flush ensures correct ordering of flushes
- Using flush correctly is difficult and prone to subtle bugs!



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

THREADING BUILDING BLOCKS

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Generic Parallel Algorithms
- Containers
- Locks, Timing & Memory Allocation
- Flow Graph
- References and Additional Reading

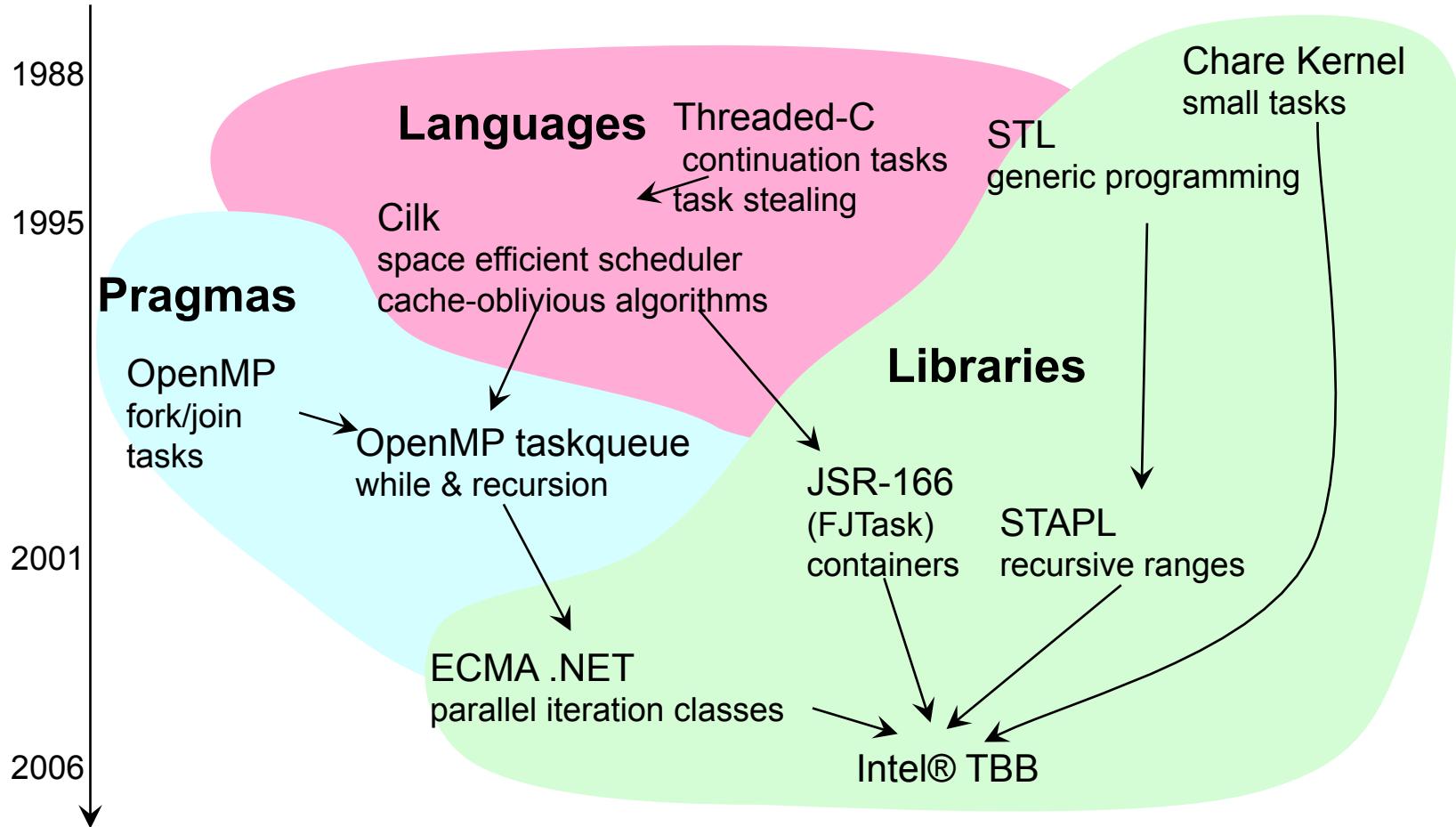
Introduction



- Threading Building Blocks (TBB) is a C++ library that simplifies threading for performance (current version 4.4 update 2)
- Move the level at which you program from threads to logical parallelism.
- Let the run-time library worry about how many threads to use, scheduling, cache etc.
- Committed to:
 - compiler independence
 - processor independence
 - OS independence
- GPL license allows use on many platforms;
commercial license allows use in products



Introduction



IBM PADTad 2009

TBB History



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Version 1.0 in 2006, after first dual core x86 processor from Intel.
- Version 2.0 in 2007.
- Version 3.0 in 2010.
- Version 4.0 in 2011.
- Version 4.1 in 2012.
- Version 4.2 in 2013
- Version 4.3 in 2014
- Version 4.4 in 2015



Components of TBB

Generic Parallel Algorithms

parallel_for, parallel_for_each
parallel_reduce
parallel_scan
parallel_do
pipeline
parallel_sort
.....

Concurrent Containers

concurrent_hash_map
concurrent_queue
concurrent_vector

Task scheduler

task_group
task
task_scheduler_init
task_scheduler_observer

Threads

tbb_thread

Memory Allocation

tbb_allocator
cache_aligned_allocator
scalable_allocator

Synchronization Primitives

atomic, mutex, recursive_mutex
spin_mutex, spin_rw_mutex
queuing_mutex, queuing_rw_mutex
null_mutex*, null_rw_mutex

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- **Generic Parallel Algorithms**
- Containers
- Locks, Timing & Memory Allocation
- Flow Graph
- References and Additional Reading



parallel_for loop

- Sequential Program

```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        Foo(a[i]);
}
```

- No Dependencies in iterations of the loop.
- Index based random access possible.
- Use parallel_for to parallelize it

parallel_for loop



Parallel syntax

```
#include "tbb/tbb.h"

using namespace tbb;

class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
        my_a(a){}
};
```



parallel_for loop

- STL-Style body object for parallelization.
- The objects of ApplyFoo class are used for parallel chunks, one thread per chunk.
- `float *const my_a;` has the local copy of object data.
- `Operator()` declares the process performed on object.
- `blocked_range<size_t>& r` is template class which defines the iterator of the loop.
- `ApplyFoo(float a[]) :my_a(a) {}` is the copy constructor, which is invoked to create a separate copy (or copies) for each worker thread.
- Implicit copy constructor and destructor are used if not defined by user.



parallel_for loop

- The example operator() copies my_a into a. It is not necessary but can be done to:
 - Make the loop body look like original.
 - Put frequently accessed variables into local variables. It may help compilers optimize the loop better.
- Finally call parallel_for in serial code.

```
void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n),
                 ApplyFoo(a));
}
```

Lambda expression



- The conversion from sequential to parallel requires a lot of coding.
- It can be avoided by using C++11 compiler's lambda expressions.

```
void ParallelApplyFoo( float* a, size_t n ) {
    parallel_for( blocked_range<size_t>(0,n),
                  [=] (const blocked_range<size_t>& r) {
                      for(size_t i=r.begin(); i!=r.end(); ++i)
                          Foo(a[i]);
                  }
    );
}
```

- [=] defines the lambda expression. It creates a function object like ApplyFoo.
- Variables defined outside the lambda expression are captured as fields inside it, like variable a.
- [=] captures variables pass by value and [&] captures by reference.

Partitioner



- A partitioner is used to partition the iteration space of a loop.
- Can be passed as a third argument to `parallel_for`.
- Three types of partitioners are available:
 - `simple_partitioner`
 - `auto_partitioner`
 - `affinity_partitioner`
- `auto_partitioner` is the default partitioner in TBB.
- Specifying **`simple_partitioner()`** turns off automatic partitioning.
- Details in TBB User guide.

Grain Size



- Grain size (G) specifies the minimum number of iterations per chunk.
- Default value of G is 1.

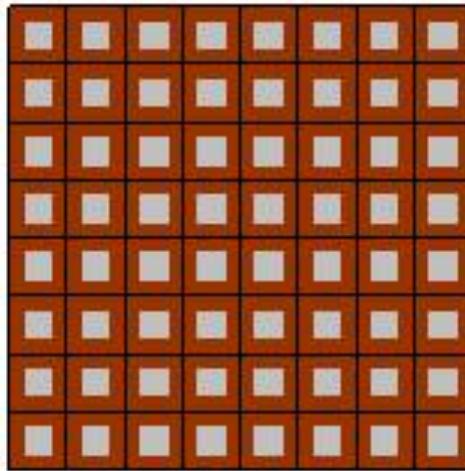
```
void ParallelApplyFoo( float a[], size_t n ) {  
    parallel_for(blocked_range<size_t>(0,n, G),  
                 ApplyFoo(a), simple_partitioner());  
}
```

- Using **simple_partitioner** guarantees that $G/2 \leq \text{chunksize} \leq G$.
- **auto-partitioner** and **affinity_partitioner** use automatic grain size heuristic.
- Given grain size both these partitioners guarantee $\text{chunksize} \geq G/2$.
- This prevents generation of very small chunks.

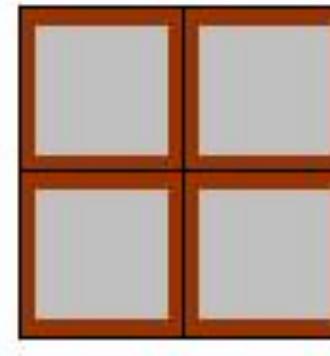
Packaging overhead vs. grain size



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Case A



Case B

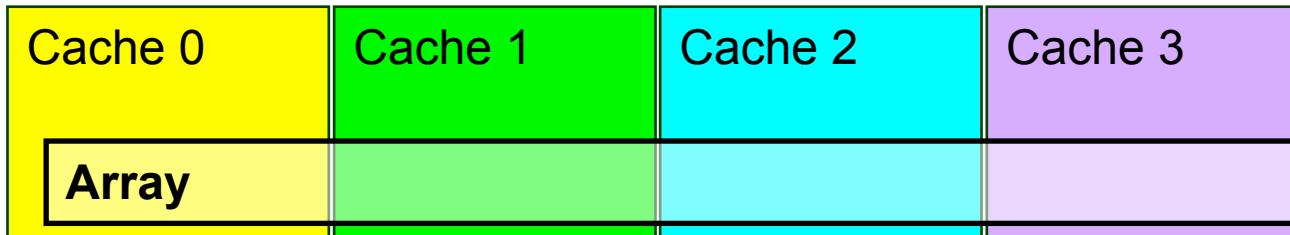
Original Work
Overhead

- Impacts of grain size on work.
- Too small grain size creates more overhead (Case A)
- Big grain size results in large workloads (Case B)

affinity_partitioner



- Big win for serial repetition of a parallel loop.
 - Numerical relaxation methods
 - Time-stepping marches



(Simple model of separate cache per thread)

```
affinity_partitioner ap;  
...  
for( t=0; ...; t++ )  
    parallel_for(range, body, ap);
```

Partitioner summary



Partitioner	Description	When used with explicit grain size (G)
simple_partitioner	Chunk size bounded by G	$G / 2 \leq \text{chunksize} \leq G$
auto_partitioner	Automatic chunk size	$G/2 \leq \text{chunksize}$
affinity_partitioner	Automatic chunk size, cache affinity and uniform distribution of iterations	$G/2 \leq \text{chunksize}$

- **automatic_partitioner** or **affinity_partitioner** should be used, because they take care of the available resources.
- **simple_partitioner** should be used when chunk size is needed to be below a specific limit or application is to be tuned for a specific machine.

parallel_reduce



- Sequential code:

```
float SerialSumFoo( float a[], size_t n ) {  
    float sum = 0;  
    for( size_t i=0; i!=n; ++i )  
        sum += Foo(a[i]);  
    return sum;  
}
```

- Parallel code:

```
float ParallelSumFoo( const float a[], size_t n ) {  
    SumFoo sf(a);  
    parallel_reduce( blocked_range<size_t>(0,n), sf );  
    return sf.my_sum;  
}
```

parallel_reduce



- SumFoo class

```
class SumFoo {  
    float* my_a;  
public:  
    float my_sum;  
    void operator()( const blocked_range<size_t>& r ) {  
        float *a = my_a;  
        float sum = my_sum;  
        size_t end = r.end();  
        for( size_t i=r.begin(); i!=end; ++i )  
            sum += Foo(a[i]);  
        my_sum = sum;  
    }  
};  
contd...
```

parallel_reduce



```
SumFoo::SumFoo( SumFoo& x, split ) : my_a(x.my_a), my_sum(0)
{ }

void SumFoo::join( const SumFoo& y ) {my_sum+=y.my_sum; }

SumFoo::SumFoo(float a[] ) : my_a(a), my_sum(0) {}
```

parallel_reduce



- The constructor initializes the local variable to identity element (`my_sum = 0`).
- The **operator** method applies the actual operation (sum of elements of array `a`).
- The **join** method of parent thread is called when a child thread finishes its work.
- All the partitioners discussed in `parallel_for` section are also valid for `parallel_reduce` too.
- Grain sizes have also the same impact as on `parallel_for`.

parallel_deterministic_reduce



- Some arithmetic problems may have different results if the operations are done in a different order.
- This is due to pitfalls in associative property in floating point arithmetic.

$$\underbrace{\frac{1}{N} + \frac{1}{N} + \cdots + \frac{1}{N}}_{N \text{ fractions}} = 0.999991 \neq$$

$$\underbrace{(\frac{1}{N} + \frac{1}{N}) + (\frac{1}{N} + \frac{1}{N}) + \cdots + (\frac{1}{N} + \frac{1}{N})}_{N/2 \text{ pairs}} = 0.999994, \text{ for } N = 1000$$

<https://software.intel.com/en-us/blogs/2012/05/11/deterministic-reduction-a-new-community-preview-feature-in-intel-threading-building-blocks>

parallel_deterministic_reduce



- parallel_deterministic_reduce provides a solution for such cases.
- The number of splits and joins are always known and do not change over different executions of the same code.
- Hence the result is always the same.
- The result may not be the same as that of a sequential algorithm.

parallel_deterministic_reduce



- parallel_deterministic_reduce has the same syntax as parallel_reduce
- simple_partitioner is used only.
- Different grain sizes may lead to different results as they effect the number of splits.

parallel_scan



- Computes a parallel prefix.
- Useful in scenarios with inherently serial dependences.

```
T DoSequentialScan(T y[], const T z[], int n) {
    T temp = idx;
    for( int i=1; i<=n; ++i ) {
        temp = temp × z[i];
        y[i] = temp;
    }
    return temp;
}
```

- \times is an commutative and associative operator.
- $\text{id}\times$ is the identity element of the operator \times .

parallel_scan



- Such problems can be parallelized by reassociating the application \times and using 2 passes.
- The application of \times maybe upto twice as many times as in serial code.
- Parallel version outperforms given enough resources and selecting right grain size.
- Suitable for systems for more than two cores.
- Interesting strategy to parallelize an inherently sequential algorithm.

parallel_scan



- Implementation of class with parallel_scan.

```
class Body {
    T sum; T* const y; const T* const z;
public:
    Body( T y_[], const T z_[] ) : sum(idx), z(z_),
y(y_) {}
    T get_sum() const {return sum;}
}

contd...
```

parallel_scan



- Implementation of class with parallel_scan.

```
template<typename Tag>
void operator()( const blocked_range<int>& r, Tag )
{
    T temp = sum;
    for( int i=r.begin(); i<r.end(); ++i ) {
        temp = temp × z[i];
        if( Tag::is_final_scan() )
            y[i] = temp;
    }
    sum = temp;
}
Body( Body& b, split ) : z(b.z), y(b.y), sum(id×) {}
void reverse_join( Body& a ) { sum = a.sum × sum; }
void assign( Body& b ) {sum = b.sum; }
};
```

parallel_scan



- parallel_scan components:

Psuedo-signature	Semantics
<code>void Body::operator()(const Range& r, pre_scan_tag)</code>	Accumulate summary for range r .
<code>void Body::operator()(const Range& r, final_scan_tag)</code>	Compute scan result and summary for range r.
<code>Body::Body(Body& b, split)</code>	Split b so that this and b can accumulate summaries separately. Body *this is object a in the table row below.
<code>void Body::reverse_join(Body& a)</code>	Merge summary accumulated by a into summary accumulated by this, where this was created earlier from a by a's splitting constructor. Body *this is object b in the table row above.
<code>void Body::assign(Body& b)</code>	Assign summary of b to this.

parallel_scan



- Parallel call to parallel_scan.

```
float DoParallelScan( T y[], const T z[], int n )
{
    Body body(y,z);
    parallel_scan( blocked_range<int>(0,n),
body );
    return body.get_sum();
}
```



parallel_for_each

- Applies a function object body over each item of a sequence or each item of a container.

```
class ApplyFoo {
public:
    void operator()( Item& item ) const {
        Foo(item);
    }
};

void ParallelApplyFooToList( const std::list<Item>& li ) {
    Parallel_for_each(li.begin(), li.end(), ApplyFoo());
    OR
    Parallel_for_each(li, ApplyFoo());
}
```

parallel_for_each



- For good performance, execution of **operator()** should take more than ~100,000 cycles of CPU. Lesser cycles may outweigh the performance benefits.
- For smaller overheads, use random access iterators for the input stream.

parallel_do



- Applies a function object body over a sequence.
- Iteration space may not be known in advance.
- Loop body may add more iterations to do before the loop exits.

```
class ApplyFoo {
public:
    void operator()( Item& item[, parallel_do_feeder<Item>& feeder] ) const {

        \\ for each new piece of work implied by item do
        item_t new_item = initializer;
        feeder.add(new_item);
    }
};

void ParallelApplyFooToList( const std::list<Item>& li ) {
parallel_do(li.begin(); li.end(); ApplyFoo());
                    OR
Parallel_do(li, ApplyFoo());
}
```

parallel_do



- For good performance, execution of **operator()** should take more than ~100,000 cycles of CPU. Lesser cycles may outweigh the performance benefits.
- Two threads never run on a single iterator concurrently.
- Typical iterators may not get good scalability because of fetching the work sequentially.
- Two ways to make parallel_do scalable
 - Iterators can be made random
 - **operator()** function can take a second argument **feeder** of same type as item and body of **operator()** add more work to **feeder** by calling **feeder.add(item)**. So that processing on newly added items can be done in parallel.

parallel_sort



- Sorts a sequence or a container.
- Requirements on the iterator or sequence are same as std::sort.
- A comparator comp(x,y) may be provided that returns true if x should come before y.

Arrays a & c are sorted in ascending order by default, While b & d are sorted in descending order using std::greater for comparison

```
const int N = 100000;
float a[N], b[N], c[N], d[N];
void SortExample() {
    for( int i = 0; i < N; i++ ) {
        a[i] = sin((double)i);
        b[i] = cos((double)i);
        c[i] = 1/sin((double)i);
        d[i] = 1/cos((double)i);
    }
    parallel_sort(a, a + N);
    parallel_sort(b, b + N,
std::greater<float>());
    parallel_sort(c);
    parallel_sort(d, std::greater<float>());
}
```

parallel_invoke



- Invokes the passed in functions in parallel.
- There can be from 2 to 10 arguments.
- Each argument must have operator() defined.
- Return values are ignored.
- Arguments are either function objects or pointer to functions.

```
void f();
extern void bar(int);

class MyFunctor {
    int arg;
public:
    MyFunctor(int a) : arg(a) {}
    void operator()() const {bar(arg);}
};

void RunFunctionsInParallel() {
    MyFunctor g(2);
    MyFunctor h(3);
    tbb::parallel_invoke(f, g, h );
}
```

Pipeline

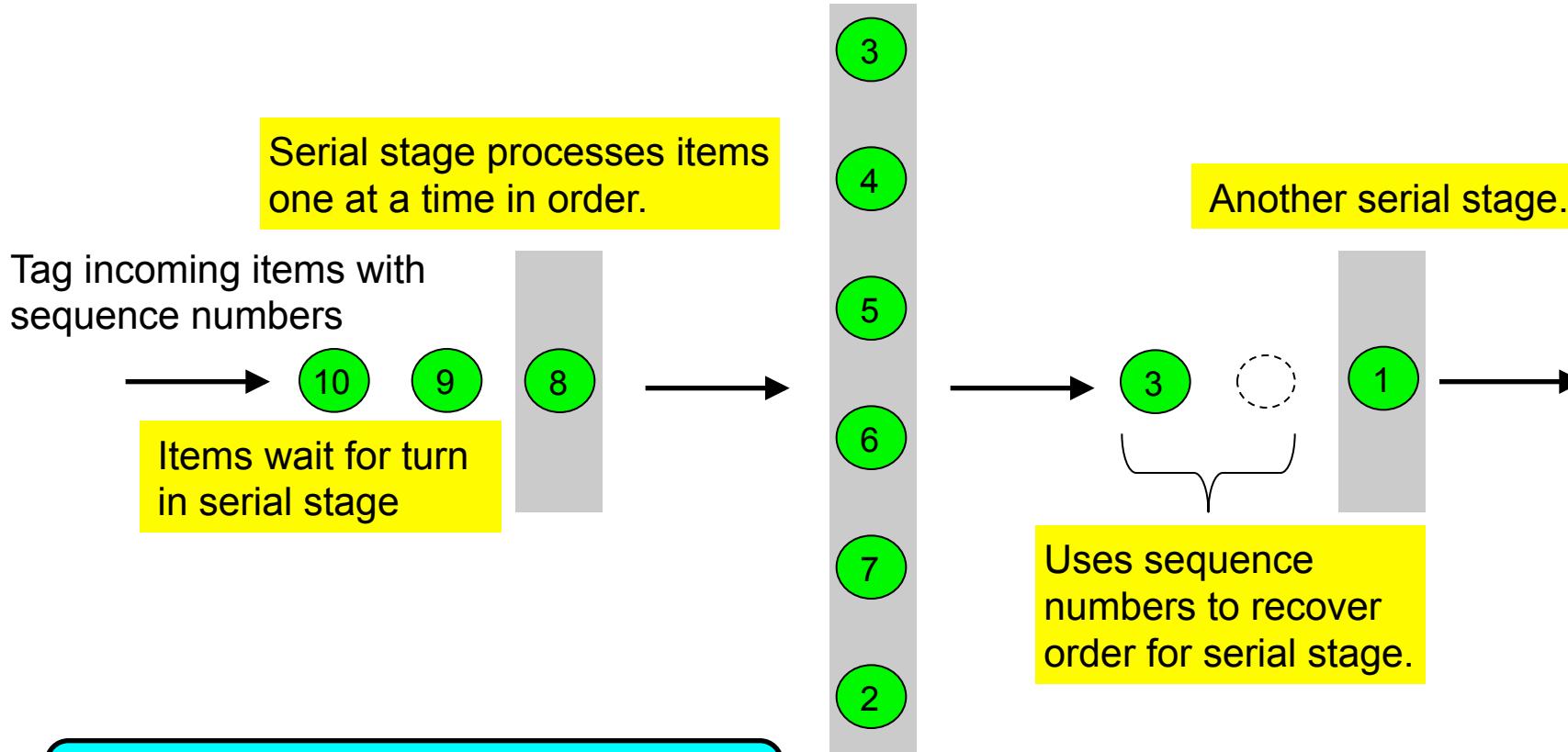


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Common parallel pattern, containing stages or filters
- Three kind of filters
 - **serial_in_order**: Should be performed serially (only one instance of this filter) and order of data should be kept
 - **serial_out_of_order**: Should be performed serially but data order is not necessary.
 - **parallel**: can be performed in parallel to other instances of same filter

Pipeline

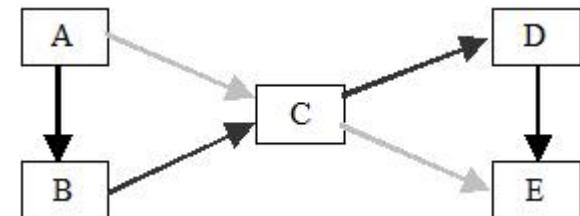
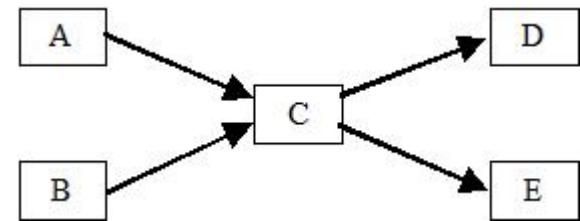
Parallel stage scales because it can process items in parallel or out of order.



Pipeline



- Throughput of a pipeline is rate at which a token flows through it and is limited by:
 - the number of tokens.
 - throughput of slowest sequential filter.
- TBB pipeline supports only linear pipelines. It do not allow non-linear pipelines like:
- The topology of non-linear pipelines needs to be changed to make them work with TBB.



parallel_pipeline function



- A strongly typed lambda-friendly function to build and run pipelines.
- Stages are specified via functors.
- **g_0, ..., g_n** are functors.
- **mode_0 ... mode_n** are type of stages (e.g. Parallel).
- **number_tokens** is the maximum number of live tokens allowed in the pipeline.

```
parallel_pipeline(number_tokens,  
                  make_filter<void,I1>(mode_0,g_0) &  
                  make_filter<I1,I2>(mode_1,g_1)  &  
                  make_filter<I2,I3>(mode_2,g_2)  &  
                  ...  
                  make_filter<In,void>(mode_n,g_n) );
```

parallel_pipeline function



```
float RootMeanSquare( float* first, float* last ) {  
    float sum=0;  
    parallel_pipeline( /*number_token=*/16,  
        make_filter<void,float*>(  
            filter::serial,  
            [&] (flow_control& fc) -> float*{  
                if( first<last ) {  
                    return first++;  
                } else {  
                    fc.stop();  
                    return NULL;  
                }  
            }  
        ) &  
    Contd...  
}
```

parallel_pipeline function



```
make_filter<float*,float>(
    filter::parallel,
    [](float* p){return (*p)*( *p); }
) &
make_filter<float,void>(
    filter::serial,
    [&](float x)
    {sum+=x;}
)
;
return sqrt(sum);
}
```

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Generic Parallel Algorithms
- **Containers**
- Locks, Timing & Memory Allocation
- Flow Graph
- References and Additional Reading

Containers



- TBB has highly concurrent data container classes.
- These containers allow multiple threads to access and update data concurrently.
- The concurrency is achieved by locks or lock free techniques.
- Concurrent containers are slower than standard ones due to synchronization overheads.

concurrent_unordered_set & concurrent_unordered_multiset



- Support concurrent insertion and traversal.
- Do not support concurrent erasure.
- Semantics similar to C++11 std::unordered_set and std::unordered_multiset except:
 - methods not safe for concurrency are prefixed with **unsafe_**.
 - for concurrent_unordered_set, insert may create a temporary pair that is destroyed if another thread inserts the same key concurrently.
 - insertion and traversal maybe concurrent. New insertions do not invalidate iterators nor change their order.
 - Iterators or only of forward type iterators

concurrent_unordered_set & concurrent_unordered_multiset



Class	Key Difference
concurrent_unordered_set	<ul style="list-style-type: none">An item may be inserted only once
concurrent_unordered_multiset	<ul style="list-style-type: none">An item may be inserted more than oncefind will return the first item in set with the matching search key, though concurrent accesses to the set may insert other occurrences of the same item before the one returned

concurrent_unordered_map & concurrent_unordered_multimap



- Support concurrent insertion and traversal.
- Do not support concurrent erasure.
- Semantics similar to C++11 std::unordered_map and std::unordered_multimap except:
 - methods not safe for concurrency are prefixed with **unsafe_**.
 - for concurrent_unordered_map, insert may create a temporary pair that is destroyed if another thread inserts the same key concurrently.
 - insertion and traversal maybe concurrent. New insertions do not invalidate iterators nor change their order.
 - Iterators or only of forward type iterators

concurrent_hash_map



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Maps keys to values in a way to support concurrent access to values.
- Keys are unsorted.
- The interface resembles typical STL containers.

concurrent_unordered_set & concurrent_unordered_multiset & concurrent_hash_map



Class	Key Difference
concurrent_hash_map	<ul style="list-style-type: none">Permits concurrent erasure, and has built-in locking.
concurrent_unordered_map	<ul style="list-style-type: none">Permits concurrent traversal and insertion, no visible locking, closely resembles C++11 unordered_mapHas the [] operator and at accessors.
concurrent_unordered_multimap	<ul style="list-style-type: none">More than one <key,value> pair with the same key value is allowedfind will return the first <key,value> pair with the matching key at the point of search, though concurrent accesses to the multimap may other insert other pairs with the same key before the one returned

concurrent_queue



- First-in first-out data structure permitting multiple threads to concurrently push and pop items.
- Unbounded capacity subject to memory and hardware limitations.
- The interface similar to std::queue, except:
 - Access to front and back is not permitted as it is unsafe for concurrency.
 - Size reported maybe incorrect if concurrent push or try_pop operations are in flight.
 - try_pop function both copies and pop items from the queue unless it is empty.



concurrent_bounded_queue

- Similar to concurrent_queue but with following differences:
 - Adds the ability to specify a capacity
 - Push operation waits until it can complete without exceeding the capacity of the queue.
 - Waiting pop functions waits until it can pop an item.
 - size function returns number of push operations minus number of pop operations. For example, if there are three pop operations waiting then size will return -3.
 - abort function causes any waiting push or pop operations to abort and throw an exception.



concurrent_priority_queue

- Concurrent push and pop operations permitted.
- Items popped according to the priority order as determined by parameter.
- Similar to std::priority_queue, except:
 - Access to highest priority item is not available.
 - try_pop copies and pops item from the queue if present.
 - size and empty functions may report inaccurately due to pending push or pop operations.

concurrent_vector



- Random access to elements by index. Index starts from zero.
- Multiple threads may append to vector concurrently.
- Inserting new elements do not invalidate the current iterators or indices.
- Meets all requirements of a container and reversable container as specified in ISO C++ standard.
- Does not meet the sequence requirements due to the absence of insert and erase methods.
- May allocate memory in fragments, which may increase the access time due to fragmentation across cache lines.
- `shrink_to_fit` method merges several fragments into single contagious array that may improve access time.

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Generic Parallel Algorithms
- Containers
- Locks, Timing & Memory Allocation
- Flow Graph
- References and Additional Reading

Locks



- Different type of locks are available in TBB.
- They control how many number of threads at a time can run a code region.
- **Speculative locking:**
 - On hardware that support hardware transactional memory (HTM), this lock lets multiple threads to lock the same region as long as no conflict occurs. A conflict depends on hardware in use.
 - On hardware without HTM support, this type of lock behaves like its other counterparts but with worst performance.



- **ReadWrite Mutex Lock:**

- A parameter (`write=true`) tells if the lock is being requested for write permission or (`write=false`) for read permission only.
- Multiple reader locks can be held simultaneously if it does not have a write lock.
- A write lock excludes all other threads from holding a lock on the mutex in same time.



- **mutex:**
 - models the mutex concept.
 - Wrapper around OS provided mutual exclusion.
 - Portable across different OSs
 - Lock released if exception occurs in the locked region
- **recursive_mutex:**
 - Similar to mutex, except that a thread may acquire multiple locks on it.
 - Thread must release all locks before other threads may acquire it.



- **spin_mutex:**
 - Not scalable, fair, or recursive.
 - Ideal when lock is lightly contended and is held for only few instructions.
 - If a thread has to wait for the lock, it busy waits that degrades system's performance.
 - If wait time is short, it significantly improves performance than other mutexes
- **speculative_spin_mutex:**
 - Better performance than spin_mutex if running on a processor with HTM support due to multiple threads running same lock.



- **spin_rw_mutex:**
 - Same as spin mutex but models ReadWrite mutex model.
- **speculative_spin_rw_mutex:**
 - Same as speculative_spin_mutex but models ReadWrite mutex model.



- **queuing_mutex:**
 - scalable, fair, and not recursive.
 - Waiting thread spins on its own local cache line.
 - Threads acquire lock in the order they requested it.
 - queuing_mutex does busy waiting, so may degrade performance for long waiting times.
- **queuing_rw_mutex:**
 - Same as queuing_mutex but models ReadWrite mutex concept.



- **null_mutex:**
 - Models mutex concept syntactically but does nothing.
 - Useful for instantiating a template that accepts mutex but mutual exclusion is not needed for that instance.
- **null_rw_mutex:**
 - Same as null_mutex but models ReadWrite mutex concept.
- **atomic:**
 - Supports atomic read, write, fetch-and-add, fetch-and-store, and compare-and-swap operations.

Locks



```
Node* FreeList;  
typedef spin_mutex FreeListMutexType;  
FreeListMutexType FreeListMutex;  
  
Node* AllocateNode() {  
    Node* n;  
    {  
        FreeListMutexType::scoped_lock lock(FreeListMutex);  
        n = FreeList;  
        if( n )  
            FreeList = n->next;  
    }  
    Contd....
```

Locks



```
if( !n )
    n = new Node();
return n;
}

void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```



- The speed up is measured by wall clock time instead of CPU time.
- `tick_count` class provides interface to get the wall clock time.
- Routines are wrappers around OS services and provide timing across threads. They have been verified as safe to be used across threads.

```
tick_count t0 = tick_count::now();
... do some work ...
tick_count t1 = tick_count::now();
printf("work took %g seconds\n", (t1-t0).seconds());
```

Memory Allocation



- Allocation of memory in parallel programs has two critical issues
 - **Scalability:** If standard serial memory allocators are used, the parallel threads must compete for shared memory, as these allocators allocate memory to one thread at a time
 - **False Sharing:** False sharing may happen if memory is allocated on more than one cache lines.
- TBB has several types of allocators. Important ones are:
 - **tbb_allocator** allocates and frees memory using scalable TBB malloc if available, otherwise reverts back to using standard malloc and free.
 - **scalable_allocator** allocates memory in a way that scales with the number of processors.
 - **cache_aligned_allocator** allocates memory along cache line boundaries. This avoids false sharing problem.
- **cache_aligned_allocator** has some space constraints, as it allocates whole cache line wide memory even for small object. So use it only if false sharing is the problem to be addressed.

Outline

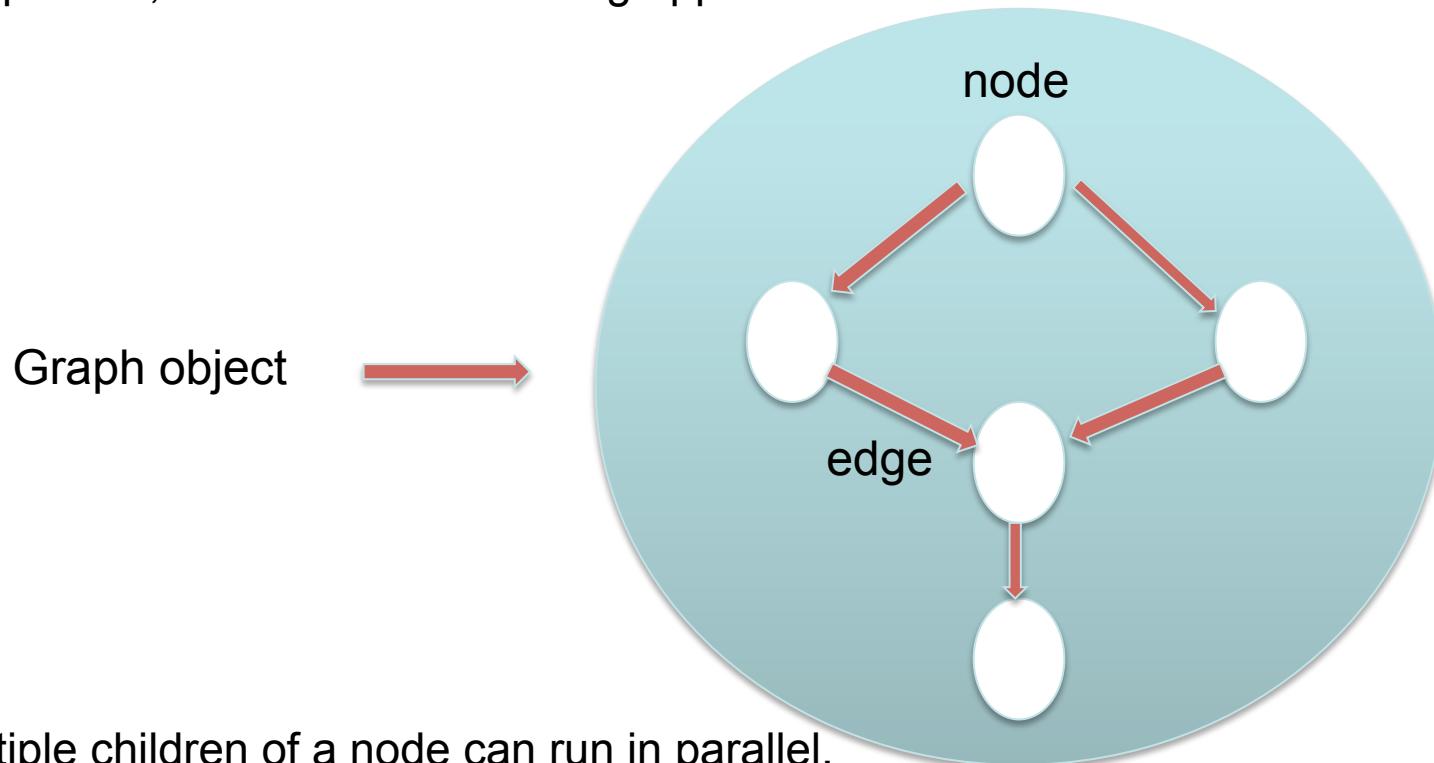


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Introduction
- Generic Parallel Algorithms
- Containers
- Locks, Timing & Memory Allocation
- **Flow Graph**
- References and Additional Reading

Flow Graph

- Available since version 4.0 (Sep 2011)
- For parallel, reactive and streaming applications



- Multiple children of a node can run in parallel.

Threadingbuildingblocks.org

Graph



- Three kinds of components available
 - A *graph* object
 - Nodes
 - Edges
- The flow of execution starts from root to children nodes.
- After completion of a node's task, a message of completion is sent to all children nodes.
- The message can be simple continue message or can have input data for children nodes.
- Different special purpose node types are available. Like function nodes, buffering nodes and split / join nodes.

Graph (Example)



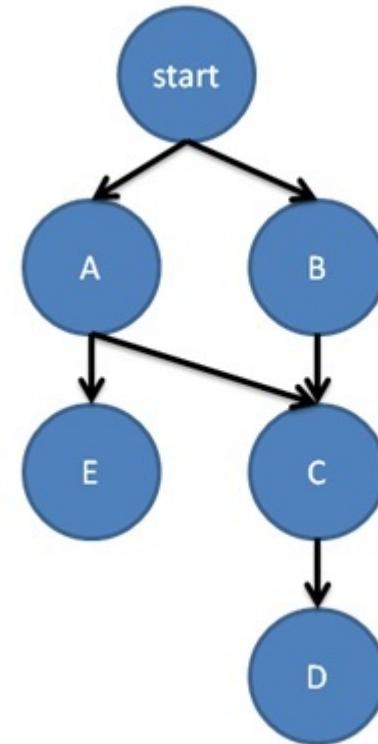
```
#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct body {
    std::string my_name;

    body( const char *name ) : my_name(name) {}

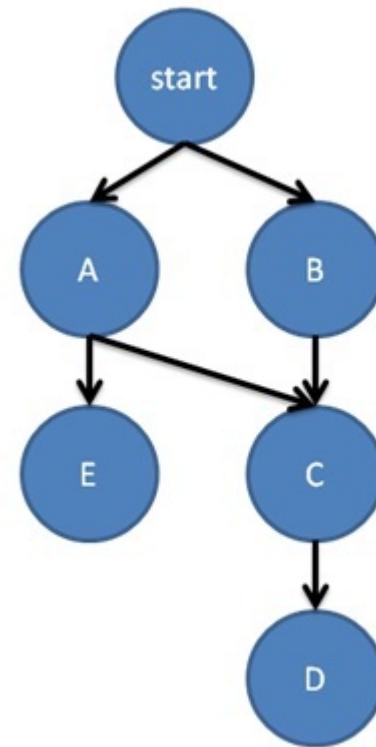
    void operator()( continue_msg ) const {
        printf("%s\n", my_name.c_str());
    }
};
```



Graph (Example)

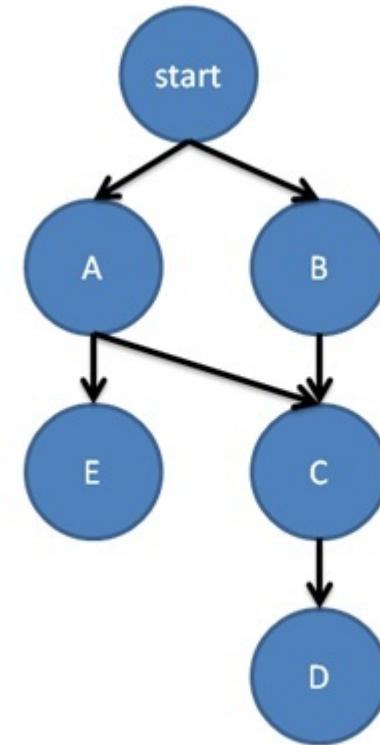


```
int main() {  
  
    graph g;  
  
    broadcast_node< continue_msg > start;  
  
    continue_node<continue_msg> a( g, body("A") );  
    continue_node<continue_msg> b( g, body("B") );  
    continue_node<continue_msg> c( g, body("C") );  
    continue_node<continue_msg> d( g, body("D") );  
    continue_node<continue_msg> e( g, body("E") );  
  
    make_edge( start, a );  make_edge( start, b );  
    make_edge( a, c );  make_edge( b, c );  
    make_edge( c, d );  make_edge( a, e );  
  
contd....
```



Graph (Example)

```
for (int i = 0; i < 3; ++i) {  
  
    start.try_put( continue_msg() );  
  
    g.wait_for_all();  
  
}  
  
return 0;  
}
```



References and Additional Reading

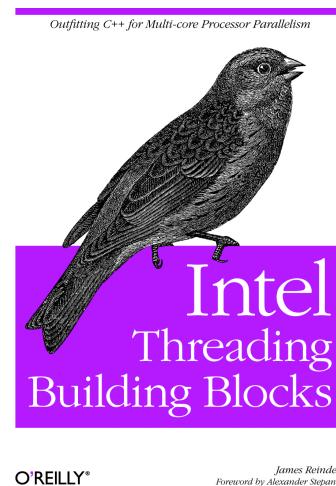


TECHNISCHE
UNIVERSITÄT
DARMSTADT

<http://threadingbuildingblocks.org/>

<https://software.intel.com/en-us/tbb-reference-manual>

Intel Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism, O'Reilly Media, 2007





Software Engineering for Multicore Systems

Dr. Ali Jannesari

C++11 CONCURRENCY

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Thread Management**
- Synchronization
- Atomics

Thread creation



- `std::thread();`
- `std::thread(std::thread&& other);`
- `template<class Function, class... Args>`
`std::thread(Function&& F, Args&&... args);`
- `std::thread(const std::thread&) = delete;`

Example: Thread creation



```
void do_some_work();
std::thread my_thread(do_some_work);
```

passing a function

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
background_task f;
std::thread my_thread(f);
```

passing a callable
object

f is copied
into storage
of thread

Example 2: Thread creation



```
struct A {
    int a = 1;
    int b = 2;
};

void foo(A& refA) {
    refA.a = 2;
    ++refA.b;
    cout << "In thread: a=" << refA.a << ", b=" << refA.b << endl;
}

int main(int argc, char* argv[]) {
    A aa;
    thread t(foo, aa);           // aa is firstly copied into thread, then the temporary
    copy is bound to foo()'s parameter.
    t.join();
    cout << "In main: a=" << aa.a << ", b=" << aa.b << endl;
    cout << t.joinable() << endl;
    return 0;
}

// Fix: thread t(foo, std::ref(aa));
// Any (implicit) conversion you want should be performed outside std::thread()
```

Thread termination



- Completion of the function supplied to its constructor
- Termination of the program

```
void oops() {  
    int some_local_state=0;  
    func my_func(some_local_state);  
    std::thread my_thread(my_func);  
    my_thread.join();  
}
```

- Calling join() ensures that thread is finished before the function exits
- Also cleans up any storage associated with the thread
- Can be called only once for a given thread – thread no longer joinable afterwards



Exceptions

Decision to join or to detach must be made before thread is destroyed

- `detach()` can be called immediately
- Finding the best place to `join()` requires more consideration

To avoid termination in the presence of exceptions

- Don't forget to call `join()` also in the exceptional case

Exceptions (2)



```
std::thread t(my_func);  
try  
{  
    do_something_in_current_thread();  
}  
catch(...)  
{  
    t.join();  
    throw;  
}  
t.join();
```

Disadvantage – verbose and error prone

Resource Acquisition Is Initialization (RAII)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Programming idiom designed for exception-safe resource management
- Motivation
 - Guarantee release of a resource at the end of a scope
 - Basic exception safety
- Implementation
 - Release resource in destructor
 - Automatic variables are destroyed automatically when leaving the scope where they have been allocated



```
class scoped_thread {
    std::thread t;
public:
    explicit scoped_thread(std::thread t_) :
        t(std::move(t_))
    {
        if(!t.joinable())
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};
```

Background threads



- `detach()` leaves the thread to run in the background without any means of communicating with it
 - No longer joinable
 - Ownership and control passed to the C++ runtime system
- Detached threads often called **daemon** threads after the UNIX concept of a daemon process that runs in the background
- Typically long running
- Perform background tasks
 - Monitoring the file system
 - Clearing unused entries out of object caches

Ownership semantics of std::thread



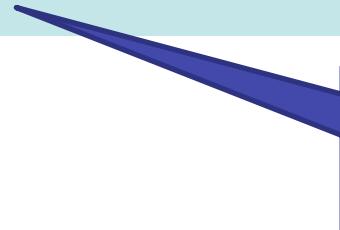
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Similar to std::unique_ptr
 - Difference – doesn't own a dynamic object but rather a **resource**
- std::thread can be moved but not copied
- Ownership can be transferred between instances – but not shared

Transferring ownership of a thread



```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2 = std::move(t1);
t1 = std::thread(some_other_function);
std::thread t3;
t3 = std::move(t2);
t1 = std::move(t3);
```



Will terminate the program

Identifying threads



- Thread identifiers are of type `std::thread::id`
- Can be obtained in two ways
 - `get_id()` member function
 - `std::this_thread::get_id()`
- If thread object does not have an associated thread of execution, `get_id()` returns default constructed `std::thread::id` object
 - Indicates “not any thread”
- Can be freely copied and compared
 - If they are equal they represent the same thread or “not any thread”
 - If they are different, they represent different threads or one is a thread and the other is holding the “not any thread” value

Identifying the master thread



- Store result of `std::this_thread::get_id()` before launching other threads
- Core part of algorithm can check its own thread ID against stored value

```
std::thread::id master_thread;

void some_core_part_of_algorithm()
{
    if (std::this_thread::get_id() == master_thread)
        do_master_thread_work();
    else
        do_common_work();
}
```

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Thread Management
- **Synchronization**
 - Mutexes and locks
 - Condition variables
 - Asynchronized tasks and futures
- Atomics

Mutexes in C++



```
#include <mutex>

std::list<int> some_list;
std::mutex some_mutex;

void add_to_list(int new_value) {
    std::lock_guard<std::mutex> guard(some_mutex);
    some_list.push_back(new_value);
}

bool list_contains(int value_to_find) {
    std::lock_guard<std::mutex> guard(some_mutex);
    return std::find(some_list.begin(), some_list.end(),
                     value_to_find) != some_list.end();
}
```

Unique locks



TECHNISCHE
UNIVERSITÄT
DARMSTADT

`std::lock_guard<>`

- Always needs to own the mutex
- Not copy-assignable

`std::unique_lock<>`

- Does not have to own the mutex
 - Locking can be deferred
 - Mutex can be unlocked before object is destroyed (via `unlock()`)
- Movable
- More flexible – but incurs some overhead

std::unique_lock



```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);

class X {
private:
    some_big_object some_detail;
    mutable std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd) {}

    friend void swap(X& lhs, X& rhs) {
        if(&lhs==&rhs) return;
        std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock);
        std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock);
        std::lock(lhs.m,rhs.m);
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

std::unique_lock (2)



Ownership of a mutex can be transferred by **moving** instances around

```
std::unique_lock<std::mutex> get_lock() {
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;
}

void process_data() {
    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

- Pattern useful where mutex is dependent on the state of the program

Usage pattern example



- Unique lock not returned directly but as data member of a gateway class
 - Can be used to ensure correctly locked access to protected data
- All access to the data through gateway class
 - If you want to access the data, you obtain instance of the gateway class by calling function such as `get_lock()`
 - When you are finished, you destroy the gateway object. Lock is then released

Condition variables



- Can be used to wait for the event itself
- Associated with an event or condition
- One or more threads can wait for the condition to be satisfied
- Thread that establishes the condition can notify waiting thread(s) and wake them up
- Two flavors in C++
 - `std::condition_variable` – needs mutex
 - `std::condition_variable_any` – needs mutex-like object (more flexible, but potentially less efficient)

Preparing data to process



```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```

Waiting for data to process



```
void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,
                       []{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

wait()



```
void wait(std::unique_lock<std::mutex>& lock);
```

```
void wait(std::unique_lock<std::mutex>& lock, Predicate pred);
```

- Causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied
- Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on the condition variable
- The thread will be unblocked when notify_all() or notify_one() is called. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait exits
- Equivalent to

```
while (!pred()) {  
    wait(lock);  
}
```

notify_x()



TECHNISCHE
UNIVERSITÄT
DARMSTADT

void notify_one()

- If any threads are waiting on the condition variable, calling `notify_one` unblocks one of the waiting threads

void notify_all()

- Unblocks all threads currently waiting for the condition variable
- Useful to let all threads wait for the (re-)initialization of shared data

notify_x() (2)

- The effects of notify_one()/notify_all() and wait() take place in a single total order
- It is impossible for notify_one() to be delayed and unblock a thread that started waiting just after the call to notify_one() was made

Future and std::async()



- Use `std::async()` to start an asynchronous task for which you don't need the result right away
- `std::async()` returns future object, which will eventually hold the value of the return function
- Calling `get()` on the future blocks the thread until future is ready and returns the value
- Allows additional arguments to be passed to the function – in the same way as `std::thread`

Example: std::async()



```
struct X
{
    void foo(int, std::string const&);
    std::string bar(std::string const&);
};

X x;
auto f1=std::async(&X::foo, &x, 42, "hello");
auto f2=std::async(&X::bar, x, "goodbye");
```

Calls p->foo(42, "hello")
where p is &x

Calls
tmpx.bar("goodbye")
where tmpx is copy of x



Unique future

- `std::future<>`
- Modeled after `std::unique_ptr`
- Only one instance may refer to the same event
- Movable
- Access to single instance must be protected (e.g., with mutex)

Template parameter is type of associated data. Use `void` if there is no associated data.

Shared future

- `std::shared_future<>`
- Multiple instances may refer to the same event
- Copyable
- All instances become ready at the same time
- Multiple threads may access their own copy of a shared future without synchronization

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Thread Management
- Synchronization
- **Atomics**

Atomics in C++



- Atomic types accessible through specializations of `std::atomic<T>` class template
 - standard atomic operations may use mutexes internally
 - expected that atomic variants of all built-in types are lock-free
 - can be found out using `is_lock_free()` member function
 - operations on `std::atomic_flag` are required to be lock-free
 - can be used to create atomic variant of user-defined type (no performance guarantee since mutexes may be used internally)

Standard atomic types



Atomic type	Corresponding specialization
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uchar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<unsigned short>
atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>
atomic_char16_t	std::atomic<char16_t>
atomic_char32_t	std::atomic<char32_t>
atomic_wchar_t	std::atomic<wchar_t>

Also typedefs for Standard Library typedefs such as std::size_t available

Operations on atomic types



- Standard atomic types are not copyable or assignable in the conventional sense – no copy constructor or copy assignment operator
 - Rationale – assignment and copy construction involve two objects. A single operation on two distinct objects can't be atomic
- They support
 - Assignment from and implicit conversion to the corresponding built-in types
 - Direct load() and store() member functions, exchange(), compare_exchange_weak(), and compare_exchange_strong()
 - Compound assignment operators where appropriate: +=, -=, *=, |=, etc. and corresponding named member functions such as fetch_add()



Example: A Spinlock

```
class TASLock {
    std::atomic_bool flag = false;
public:
    void lock() {
        while(flag.exchange(true, std::memory_order_acquire));
    }
    void unlock() {
        flag.store(false, std::memory_order_release);
    }
};
```



Example: A Spinlock (2)

```
class TTASLock {
    std::atomic_bool flag = false;
public:
    void lock() {
        while(true) {
            while(flag.load()) {}
            if(!flag.exchange(true, std::memory_order_acquire));
                return;
        }
    }
    void unlock() {
        flag.store(false, std::memory_order_release);
    }
};
```

Whose performance is better, TASLock or TTASLock?

An array-based lock



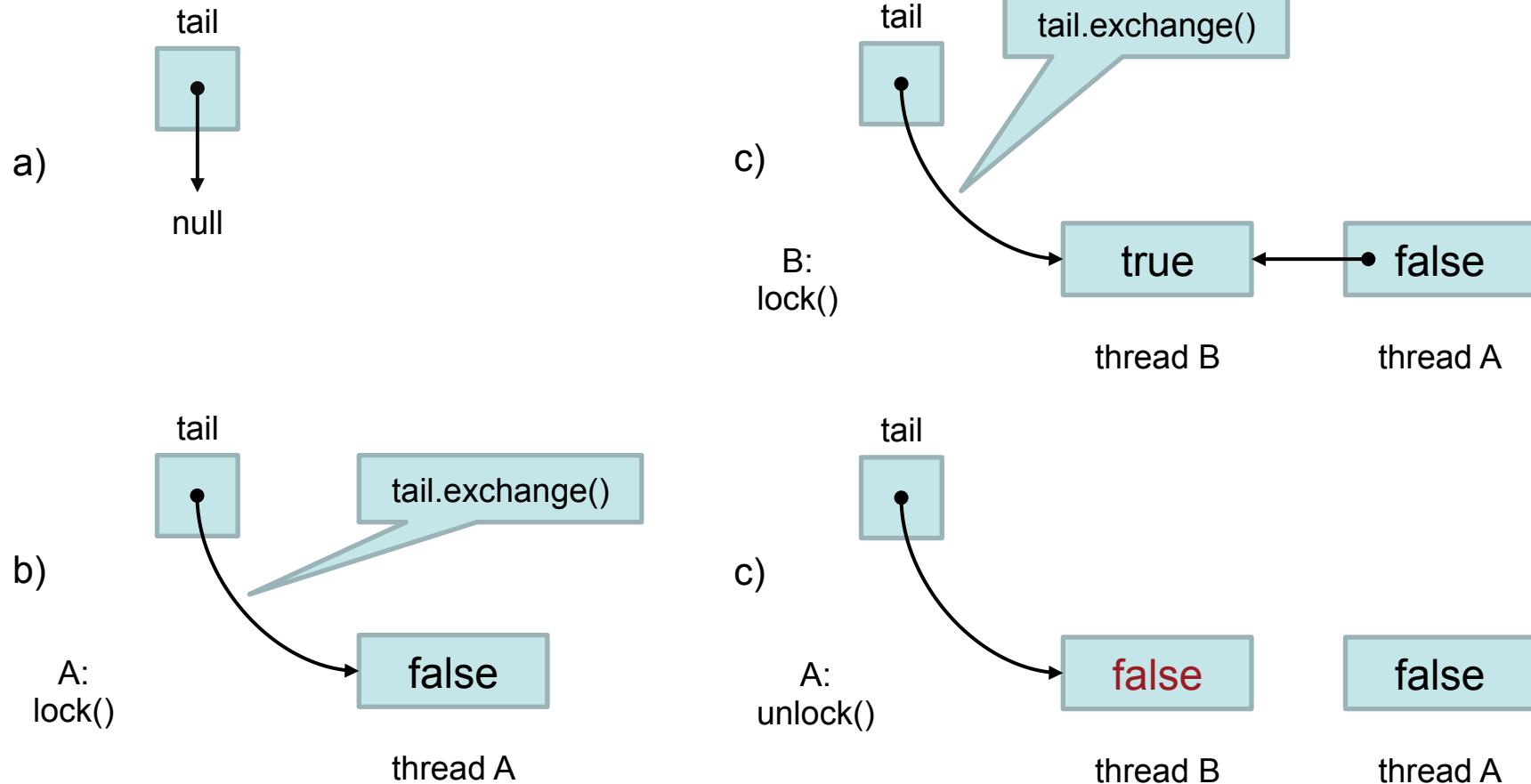
```
class ALock {
    static thread_local int myIndex;
    std::atomic_int tail = 0;
    std::atomic_bool *flag;
    int size = 0;

public:
    Alock(int capacity) { flag = new std::atomic_bool[capacity];
                          flag[0].store(true); // all other flags are false
                          size = capacity;
                          myIndex = 0; }

    void lock() {
        myIndex = tail.fetch_add(1) % size;
        while (!flag[myIndex].load(std::memory_order_acquire)) {};
    }

    void unlock() {
        flag[myIndex].store(false, std::memory_order_release);
        flag[(myIndex + 1) % size].store(true, std::memory_order_release);
    }
};
```

A queue lock





Software Engineering for Multicore Systems

Dr. Ali Jannesari

PARALLEL PROGRAMMING MODELS – WRAP-UP

Case Study

- Case study^[1]:
 - 135 open-source parallel Java, C# and C++ projects (46 Java projects, 45 C# projects and 44 C++ projects)
 - From major repositories (SourceForge, GitHub, apache.org, ...)
- Much less parallel C++ programs compared to Java and .Net
- Java and .NET are on a par parallel feature-wise, slightly different focus, C++ distant third
- C++ mostly uses libraries, Java and C# hardly any

[1] Marc Kiefer, Daniel Warzel, and Walter F. Tichy. An empirical study on parallelism in modern open-source projects. In Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems (SEPS 2015). ACM, New York, NY, USA, 35-44.

Parallel Programming Models

Availability of different concurrent features

	High-level Constructs	Concurrent data structures	Synchronization support
Java	30	26	15
.NET	9	10	18
C++ • C++11 • OpenMP • pthread • etc.	5	17	8

Parallel Programming Models

High-level constructs



Java

Feature	# projects	% of total
Executor	31	67.39%
ExecutorService	28	60.87%
Callable	24	52.17%
Future	23	50.00%
BlockingQueue	20	43.48%
ScheduledExecutorService	16	34.78%
ThreadPoolExecutor	15	32.61%
ArrayBlockingQueue	13	28.26%
LinkedBlockingQueue	13	28.26%
ScheduledFuture	11	23.91%
FutureTask	8	17.39%
ScheduledThreadPoolExecutor	8	17.39%
PriorityBlockingQueue	3	6.52%
CompletionService	2	4.35%
AbstractExecutorService	2	4.35%
Delayed	2	4.35%
DelayQueue	2	4.35%
ExecutorCompletionService	2	4.35%
ForkJoinPool	2	4.35%
RecursiveAction	2	4.35%
RunnableFuture	2	4.35%
SynchronousQueue	2	4.35%
BlockingDeque	1	2.17%
LinkedBlockingDeque	1	2.17%
LinkedTransferQueue	1	2.17%
RecursiveTask	1	2.17%
TransferQueue	1	2.17%
ForkJoinTask	0	0.00%
ForkJoinWorkerThread	0	0.00%
RunnableScheduledFuture	0	0.00%

C#

Feature	# projects	% of total
Task	33	73.33%
ThreadPool	19	42.22%
TaskScheduler	10	22.22%
Parallel.For	8	17.78%
BlockingCollection	6	13.33%
Parallel.ForEach	6	13.33%
TaskFactory	5	11.11%
Parallel.Invoke	1	2.22%
Partitioner	0	0.00%

C++

Feature	# projects	% of total
#pragma omp parallel for	6	13.64%
future/promise	3	6.82%
#pragma omp parallel packaged task	2	4.55%
shared future	0	0.00%
	0	0.00%

Parallel Programming Models

Concurrent data structures



Java

Feature	# projects	% of total
ConcurrentHashMap	26	56.52%
synchronizedMap	15	32.61%
CopyOnWriteArrayList	14	30.43%
ConcurrentLinkedQueue	11	23.91%
synchronizedList	10	21.74%
SynchronizedSet	9	19.57%
CopyOnWriteArraySet	6	13.04%
ConcurrentSkipListSet	3	6.52%
synchronizedCollection	3	6.52%
ConcurrentLinkedDeque	1	2.17%
ConcurrentSkipListMap	1	2.17%
synchronizedSortedMap	1	2.17%
synchronizedSortedSet	1	2.17%
ConcurrentNavigableMap	0	0.00%

C#

Feature	# projects	% of total
ConcurrentDictionary	22	48.89%
ConcurrentQueue	11	24.44%
ConcurrentBag	5	11.11%
ConcurrentStack	0	0.00%

C++

Feature	# projects	% of total
#pragma omp parallel for	6	13.64%
future/promise	3	6.82%
#pragma omp parallel	2	4.55%
packaged_task	0	0.00%
shared_future	0	0.00%

Parallel Programming Models

Synchronization support



Java

Feature	# projects	% of total
synchronized	40	86.96%
Lock	19	41.30%
ReadWriteLock	15	32.61%
ReentrantReadWriteLock	14	30.43%
CountDownLatch	12	26.09%
ReentrantLock	11	23.91%
Semaphore	9	19.57%
Condition	5	10.87%
CyclicBarrier	4	8.70%
AbstractQueuedSynchronizer	2	4.35%
LockSupport	2	4.35%
AbstractOwnableSynchronizer	0	0.00%
AbstractQueuedLongSynchronizer	0	0.00%
Exchanger	0	0.00%
Phaser	0	0.00%

C++

Feature	# projects	% of total
mutex	39	88.64%
condition variable	28	63.63%
Semaphore	18	40.91%
CriticalSection	17	38.64%
unique lock	16	36.36%
lock guard	12	27.27%
barrier	5	11.36%
#pragma omp critical	3	6.82%

C#

Feature	# projects	% of total
lock()	42	93.33%
ManualResetEvent	22	48.89%
Monitor	17	37.78%
AutoResetEvent	16	35.56%
ReaderWriterLockSlim	15	33.33%
WaitHandle	13	28.89%
EventWaitHandle	10	22.22%
Mutex	10	22.22%
ManualResetEventSlim	8	17.78%
Barrierr	7	15.56%
Semaphore	6	13.33%
SpinWait	4	8.89%
CountdownEvent	3	6.67%
ReaderWriterLock	3	6.67%
SemaphoreSlim	3	6.67%
MethodImplOptions.Synchronized	2	4.44%
Interlocked.MemoryBarrier	1	2.22%
SpinLock	0	0.00%

Parallel Programming Models

Libraries / programming models C++



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Library	# projects	% of total
pthreads	22	50.00%
Windows-threads	20	45.45%
Boost-thread	7	15.91%
OpenMP	6	13.64%
Intel TBB	3	6.82%
MPI	2	4.55%
Microsoft PPL	1	4.55%
Apple GCD	1	4.55%

Parallel Programming Models

Patterns



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Pattern	Java	.NET	C++
Master-Worker	29	23	23
Producer-Consumer	10	5	5
Pipeline	8	9	8
Parallel Loop	1	8	5
Fork-Join	3	1	1

Wrap-up

Parallelization Strategy



Java	<ul style="list-style-type: none">• Support low to high level parallelization
OpenMP	<ul style="list-style-type: none">• Incremental• Low Parallelization effort (LOC)• Hard to obtain optimal performance
.NET	<ul style="list-style-type: none">• PLINQ/Functional way of writing parallel code
Intel TBB	<ul style="list-style-type: none">• Pattern oriented• Easy to implement complex algorithms• High level parallelization• High parallelization effort (LOC)• Low flexibility
C++11	<ul style="list-style-type: none">• Low level parallelization• Complex programming• High learning curve

Wrap-up

Comparison



	Concurrent Algorithms and patterns	Concurrent data structures	Synchronization support
Java	★★	★★★★★	★★★★★
OpenMP	★★★	★	★★★
.NET	★★★	★★★★★	★★★★★
Intel TBB	★★★★★	★★★★★	★★★
C++11	★	★	★★★★★

Wrap-up

Comparison (2)



	Thread Manipulation	Automatic Scheduling	Portability
Java	★★★★★	✓	✓
OpenMP	★★	✓	✓
.NET	★★★★★	✓	✗
Intel TBB	★	✓	✓
C++11	★★★★★	✗	✓



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering for Multicore Systems

Dr. Ali Jannesari

TESTING AND DEBUGGING

Difficulties of testing and debugging in parallel programs – Why?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Parallel programs are non-deterministic
 - Behavior not always reproducible
 - Even with identical inputs results may vary in different executions
 - Due to the timing behavior of threads or synchronization error
 - Undesired behavior may occur very rarely
- Measuring effect ("sample Effect")
 - When troubleshooting any program modification (even simple "printf" statements) can change the timing behavior of an application and reduce the probability of occurrence of the bug investigated
 - Problem similar to the "uncertainty principle" → "Heisenbugs"
- Typical concurrency bugs: deadlock and data races

Data race



- Find largest element in a list of numbers

```
cur_max = MINUS_INFINITY;  
#pragma omp parallel for  
for ( i = 0; i < n; i++ )  
    if ( a[i] > cur_max )  
        cur_max = a[i];
```

- This parallel version can yield a wrong result

Thread 0

```
read a[i] (12)  
read cur_max (10)  
if (a[i] > cur_max) (12 > 10)  
    cur_max = a[i] (12)
```

Thread 1

```
read a[j] (11)  
read cur_max (10)  
  
if (a[j] > cur_max) (11 > 10)  
    cur_max = a[j] (11)
```

Values
read or
written

Data race (2)



- Data race: if two concurrent threads access a shared variable and at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous
- Example

```
#pragma omp parallel for shared(a,b)
    for ( i = 0; i < n; i++ ) {
        a[i] = a[i] + b;
    }
```
- b is declared shared
 - b is not modified - only read
 - No conflict on b → no need for synchronization
- a[] is declared shared
 - Each thread modifies a distinct array location!
 - No conflict on a[] → no need for **explicit** synchronization

Removing data races using privatization



```
#pragma omp parallel for shared(a) private(b)
    for ( i = 0; i < n; i++ ) {
        b = f(i, a[i]);
        a[i] = a[i] + b;
    }
```

- Some variables are accessed by all the threads but not used to communicate between them
- Used as scratch storage within a thread
- Declare private to avoid data races

Removing data races using reduction



- Sum of all elements in an array

```
    sum = 0;
#pragma omp parallel for reduction(+:sum)
for ( i = 0; i < n; i++ ) {
    sum = sum + a[i];
}
```

- Variation of the first example
- Compute the sum within each thread
- Sum up the local sums and yield global sum
 - Requires synchronization
 - Hidden inside the OpenMP implementation

Removing data races – synchronization primitives



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Explicit synchronization between threads:
 - Locks
 - Critical Sections
 - Barriers
 - Mutexes
 - Semaphores
 - Monitors
 - Events (signal/wait)
 - Etc.

Intentional (acceptable) data races



- Find an element in a list

```
found = 0;  
#pragma omp parallel for  
for ( i = 0; i < n; i++ ) {  
    if ( a[i] == item )  
        found = 1;  
}
```

- Not all data races require synchronization
- In the example, all updates write the same value
- If item can be found, final value will be 1 regardless of concurrent updates
- If item cannot be found, final value will be 0 because then there will be no update



Intentional(acceptable) data races (2)

- Five-point successive over relaxation (SOR)

```
#pragma omp parallel for
for ( i = 1; i < n-1; i++ ) {
    for ( j = 1; j < n-1; j++ ) {
        m[i][j] = (m[i][j] + m[i][j-1] + m[i][j+1] +
                    m[i+1][j] + m[i-1][j]) / 5.0;
    }
}
```

- Two dimensional matrix
- Each element is updated with average of itself and the four nearest neighbors in the grid
- Code contains loop-carried data dependence

Intentional (acceptable) data races (3)



- Parallelized SOR code may behave different from serial version
 - Iteration may use older value from a preceding or succeeding column
- After acceptable number of steps the result will still converge to desired solution
 - Tolerance of data races
- Algorithm belongs to the class of **relaxed algorithms**
- Caveat
 - Algorithm assumes that individual matrix elements can be updated atomically
 - Different situation if matrix elements are complex or structured

Data race detection approaches



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Static:** perform a compile-time analysis of the code, reporting potential data races
 - RacerX
- **Dynamic:** use tracing mechanism to detect whether a particular execution of a program actually exhibited data races
 - The program may be “instrumented” with additional instructions to monitor access to shared variables and synchronization operations

Dynamic data race detection



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Lockset analysis**
 - Eraser
- **Happens-before analysis**
- Hybrid (combined Lockset and Happens-before)

Lockset analysis



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Observe all instances where a shared variable is accessed by a thread
- If there is a chance that a data race can occur, be sure the shared variable is protected by a lock
- If variable isn't protected, issue a warning!



Lockset algorithm

- Idea: Try to infer the locks protecting X
- Algorithm:

C_X : locks protecting location X (“lockset”)

L_T : locks being held by thread T

When X is accessed by T,

$$C_X = C_X \cap L_T$$

If C_X is empty, then issue a warning

Lockset analysis - Example

Thread 1	Thread 2	C_x
<pre>Lock(m1); x = x + 1; Unlock(m1);</pre>		{m1}
	<pre>Lock(m1); x = x + 1; Unlock(m1);</pre>	{m1}
<pre>x = x + 1;</pre>		{ } ← warning

- The locking discipline for x is violated since no lock protects it consistently



Happens-Before analysis

- Based on Lamport's Clock (vector clocks)
- Let event **a** be in thread A and event **b** be in thread B.
 - If event **a** and event **b** are synchronization operations, construct a happens-before edge between them:
 - E.g. If $a = \text{unlock}(\mu)$ and $b = \text{lock}(\mu)$ then
$$a \xrightarrow{\text{hb}} b \text{ (a } \textit{happens-before} \text{ b)}$$
- Shared accesses i and j are concurrent
 - if neither $i \xrightarrow{\text{hb}} j$ nor $j \xrightarrow{\text{hb}} i$ holds.
- Data races between threads are possible if accesses to shared variables are concurrent

Happens-Before - Example (1)



The happens-before analysis will **not** produce a false warning in the following simple case:

Thread 1
 $X=0$

Thread 2

Signal(CV)



$T=X$



Happens-Before - Example (2)

Thread 1

```
lock(mu);
```



```
v = v + 1;
```



```
unlock(mu);
```

Thread 2

```
lock(mu);
```



```
v = v + 1;
```



```
unlock(mu);
```

The arrows represent happens-before.
The events represent an actual execution of
the two threads.

Happens-Before - Example (3) – False negative



Thread 1

$y = y + 1;$



$lock(mu);$



$v = v + 1;$



$unlock(mu);$

Thread 2

$lock(mu);$



$v = v + 1;$



$unlock(mu);$



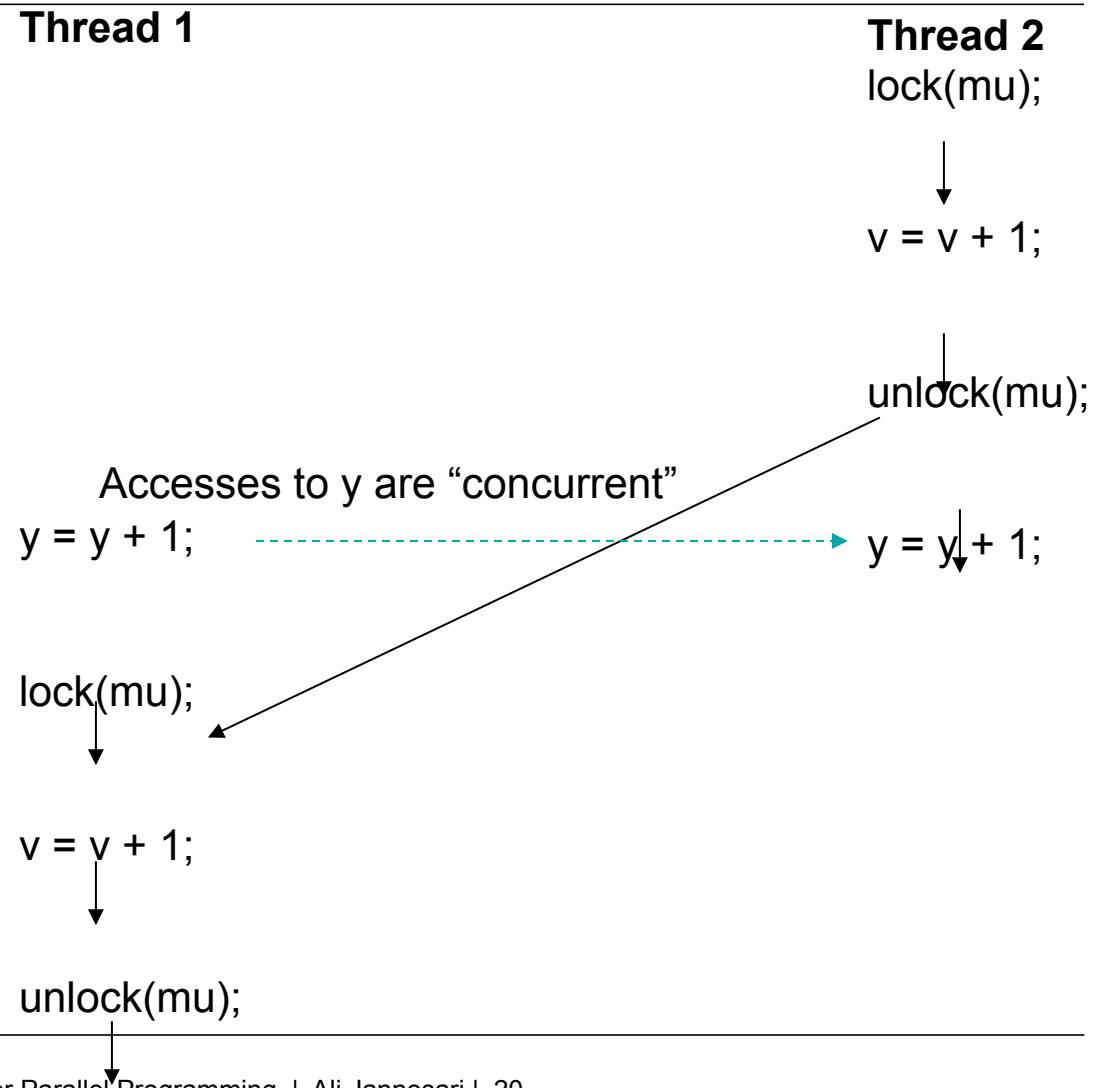
$y = y + 1;$

Accesses to both y and v are ordered by happens-before, so no data race occurred. But a false negative on y which is not protected.

Happens-Before - Example (3) – False negative



If Thread 2 executes before Thread 1, then happens-before no longer holds between the two accesses to y , so the possibility of a data race occurs and should be notified to the programmer. (It is not guaranteed, however)





Lockset vs. Happens-Before

	Scalability & performance	Low false positive rate	Insensitivity to interleaving
Lockset [Eraser-SOSP'97]	Yes	No	Yes
Happens-Before [Schonberg-PLDI'89]	No	Yes	No

Happens-Before & Lockset combination

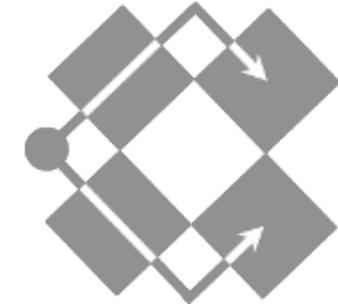


- Lockset
 - Scalable and insensitive to interleaving
 - But gives many false positives
- Until a variable is not accessed by at least 2 concurrent threads, there's no danger of a data race so no need to report a race!
- Use Happens-Before to find out if 2 threads are concurrent
- Deferring happens-before analysis until lockset analysis proves insufficient appears (two-phase analysis)

Testing parallel programs

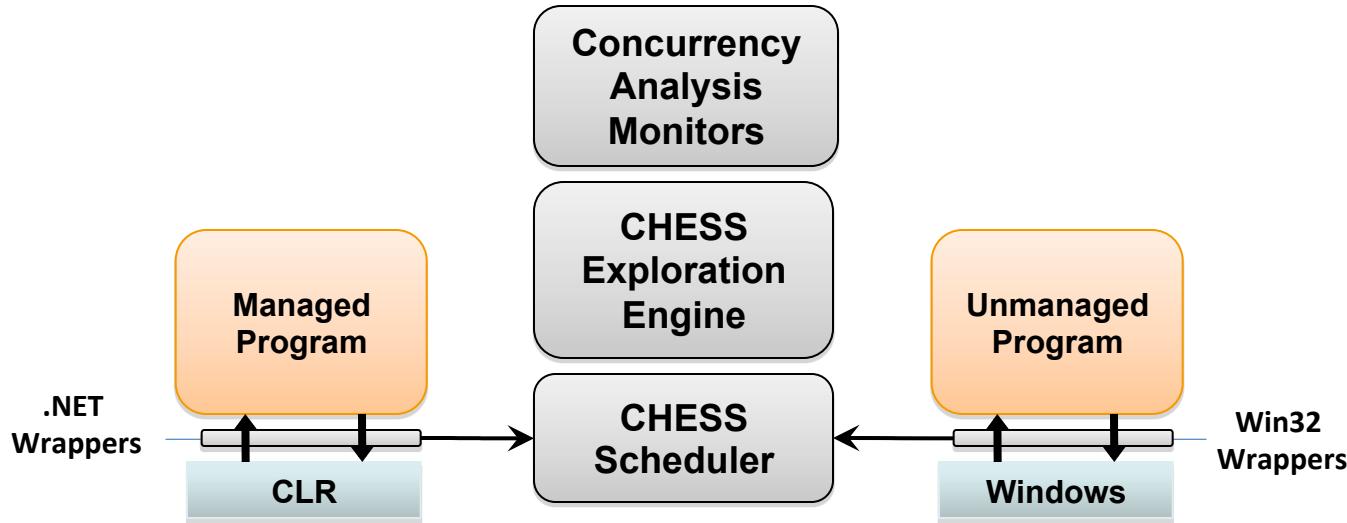


- CHESS^[1] – Dynamic approach for multi-threaded programs
- Creates systematically different threads interleaving
- Each program is executed with different interleavings (potential interleavings)
- "Replay" possible: specific interleaving which leads to data race, can be stored and replayed



[1] S. Burghardt et al, Chess: ANALYSIS and Testing of Concurrent Programs, Tutorial PLID09. link: <http://research.microsoft.com/en-us/projects/chess/>

CHESS - Architecture



- Extendable: SDK for its own monitors, happens-before queries, etc.
- Integration in Visual Studio

CHESS - Scheduler



- Exhaustive search of all non-deterministic executions

Thread 1

$x = 1;$
 $y = 1;$

$x = 1;$

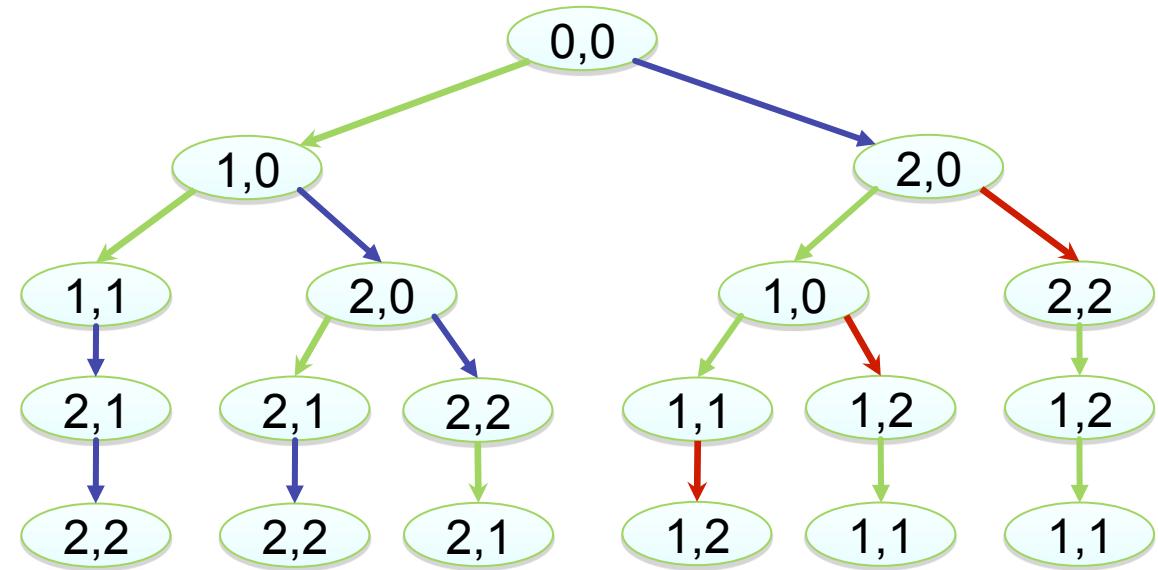
$y = 1;$

Thread 2

$x = 2;$
 $y = 2;$

$x = 2;$

$y = 2;$



For each interleaving:

- Executes the code
- Annotates all access with vector clocks and checks if there are conflicting accesses

CHESS - State Space Explosion



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Number of possible interleavings growing exponentially^[1]
- Limited scalability for large programs
- Solution:
 - Insert systematically break points ("preemption points")
 - Preemption points represent context switches that are enforced by the CHESS-Scheduler
 - Assumption: Most bugs are detectable with few preemption points (<2)

[1] M. Musuvathi & Shaz Qadeer, Iterative Context Bounding for Systematic Testing of Multithreaded Programs, PLDI07.

Software Engineering for Multicore Systems

Dr. Ali Jannesari

OPTIMIZATION

Agenda

- Introduction to optimization
- Common basic optimization techniques
- Common compiler optimizations
- Performance factors
 - Load balancing & scheduling
 - Locality
 - ccNUMA
 - False sharing
 - Critical Section

Introduction

- Goal of parallel programming is performance
- Ease of parallelization depends on the problem at hand
 - Some are embarrassingly parallel
 - Others can easily lead to speedup < 1
- Complexity of the underlying machine makes speedup sometimes hard to achieve

Important questions...



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Questions related to performance
 - Which fraction of my program runs in parallel?
 - How much work is in each parallel region?
 - How evenly is the work distributed across different threads?
 - What are the costs of accessing the memory?
 - What are the costs of communication between different threads?

Coverage



- The higher the number of processors used in a parallel program, the larger the influence of serial code regions on the performance
- Amdahl's Law

$$S = \frac{1}{(1 - F) + \frac{F}{S_p}}$$

S = overall speedup

S_p = speedup in parallel regions

F = fraction of parallelized code
(if executed serially)

- Performance is limited by F for large numbers of processors
- If $F = 50\%$ then total speedup can never be better than 2

Common Basic Optimizations

Do less work!



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Rearranging the code such that less work than before is being done will improve performance

```
bool FLAG
FLAG = false
for(i=0, i<N, i++)
    if(complex_func(A[i]) < THRESHOLD)
        FLAG = true
```

Common Basic Optimizations

Do less work!



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Rearranging the code such that less work than before is being done will improve performance

```
bool FLAG
FLAG = false
for(i=0, i<N, i++)
    if(complex_func(A[i]) < THRESHOLD)
        FLAG = true
    exit
```

Common Basic Optimizations

Avoid expensive operations



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- “one-to-one” implementation results in poor performance.
- All “strong” operations should be substituted by “cheaper” alternatives.
 - e.g. trigonometric functions or exponentiation
- This process is called “**strength reduction**”

```
int iL,iR,iU,iO,iS,iN          // {-1, +1}
double precision edelz,tt
...
edelz = iL+iR+iU+iO+iS+iN    //! loop kernel
BF = 0.5d0*(1.d0+tanh(edelz/tt))
```

Common Basic Optimizations

Avoid expensive operations



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- “one-to-one” implementation results in poor performance.
- All “strong” operations should be substituted by “cheaper” alternatives.
 - e.g. trigonometric functions or exponentiation
- This process is called “**strength reduction**”

```
double tanh_table(-6:6)
int iL,iR,iU,iO,iS,iN
double tt
...
do i=-6,6                                ! Do this once
tanh_table[i] = 0.5d0*(1.d0*tanh(db1e[i]/tt))
enddo
...
BF = tanh_table(iL+iR+iU+iO+iS+iN) ! loop kernel
```

Common Basic Optimizations

Shrinking the working set



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- “Working set” of a code is the amount of memory it uses (i.e. actually touches) in the course of a calculation.
- Shrinking the working set raises the probability for **cache hits**.
 - e.g. choosing right data types:

```
integer iL,iR,iU,iO,iS,iN → bool iL,iR,iU,iO,iS,iN
double edelz,tt
...
edelz = iL+iR+iU+iO+iS+iN //! loop kernel
BF = 0.5d0*(1.d0+tanh(edelz/tt))
```

Common Compiler Optimizations

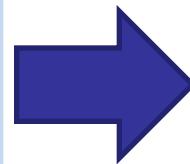
Common subexpression elimination



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- One of the most common optimization tasks done by compilers
- Trying to save time by pre-calculating parts of complex expressions and assigning them to temporary variables before a loop starts

```
// inefficient
for(i=1, i<N, i++)
    A[i]=A[i]+s+r*sin(x)
```



```
tmp=s+r*sin(x)
for(i=1, i<N, i++)
    A[i]=A[i]+tmp
```

Common Compiler Optimizations

Avoiding branches



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- “Tight” loops (few operations), are typical candidates for:
 - Software pipelining, Loop unrolling, etc.
 - If the loop body contains conditional branches, the compiler will fail to apply these optimization.

```
for(j=1, j<N, j++) {
    for(i=1, i<N, i++) {
        if(i > j)
            sign = 1
        else if(i < j)
            sign = -1
        else
            sign = 0
        C[i,j] = C[i,j] + sign * A[i,j] * B[i,j]
    }
}
```

Common Compiler Optimizations

Avoiding branches



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- “Tight” loops (few operations), are typical candidates for:
 - Software pipelining, Loop unrolling, etc.
 - If the loop body contains conditional branches, the compiler will fail to apply these optimization.

```
for (j=1, j<N, j++)
    for (i=j+1, i<N, i++)
        C[i,j] = C[i,j] + A[i,j] * B[i,j]

for (j=1, j<N, j++)
    for (i=1, i<j-1, i++)
        C[i,j] = C[i,j] - A[i,j] * B[i,j]
```

Common Compiler Optimizations

Loop fusion / Fission



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Loop Fusion:
 - Replaces multiple loops with a single one, when two loops iterate over the same range and do not reference each other's data.
- Loop Fission:
 - Transform a single loop into two loops.

Data Locality
Matters

Fusion

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
    a[i] = 1;
for (i = 0; i < 100; i++)
    b[i] = 2;
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

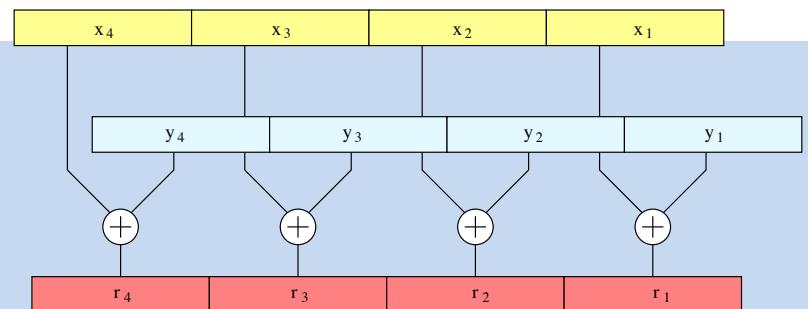
Fission

Common Compiler Optimizations

Using SIMD instruction sets (vectorization)

- Recent microprocessors have instruction set extensions for both integer and floating-point SIMD operations
- Allow concurrent execution of arithmetic operations on “wide” registers
- a “vectorizable” loop will run faster if more operations can be performed with a single instruction
 - i.e. the size of the data type should be as small as possible.
 - Switching from DP to SP data could result in up to a twofold speedup, with the additional benefit that more items fit into the cache.

```
float r[N], x[N], y[N]  
  
for(i=0, i<n, i++)  
    r[i] = x[i] + y[i]
```



Common Compiler Optimizations

Using SIMD instruction sets (vectorization)

- Using SIMD may require some rearrangement of a loop kernel
 - A number of iterations equal to the SIMD register size has to be executed in parallel without any branches in between.
 - As the overall number of iterations is generally not a multiple of the register size, some remainder loop is left to execute in scalar mode.

```
rest = mod(N, 4)
for (i=0, i<N-rest, i+=4) {
    R1 = [x[i],x[i+1],x[i+2],x[i+3]]
    R2 = [y[i],y[i+1],y[i+2],y[i+3]]
    R3 = ADD(R1,R2)
    r[i],r[i+1],r[i+2],r[i+3] = R3
}
for (i=N-rest, i<N, i++)
    r[i] = x[i] + y[i]
```

Common Compiler Optimizations

Using compiler logs



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Compiler is a crucial component in writing efficient code.
- Many compilers offer options to generate *annotated source code* listings or at least *logs* that describe in some detail what optimizations were performed

```
#<swps> 16383 estimated iterations before pipelining
#<swps>      4 unrollings before pipelining
#<swps>    20 cycles per 4 iterations
#<swps>      8 flops          ( 20% of peak) (madds count as 2)
#<swps>      4 flops          ( 10% of peak) (madds count as 1)
#<swps>      4 madds          ( 20% of peak)
#<swps>     16 mem refs       ( 80% of peak)
#<swps>      5 integer ops    ( 12% of peak)
#<swps>    25 instructions   ( 31% of peak)
#<swps>      2 short trip threshold
#<swps>    13 integer registers used.
#<swps>    17 float registers used.
```

Granularity



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Switching to parallel execution incurs overhead
 - Spawning threads
 - Distributing work
 - May involve synchronization
 - Dynamic scheduling
 - Task queue
 - Implicit barrier at the end of parallel regions / loops (unless there is a no wait clause)
- Benefit of parallelization should justify overhead
- Granularity of parallelization shouldn't be too small

Data Access



- The most important performance limiting factor is data access!
- Microprocessors tend to be inherently “unbalanced” with respect to the relation of theoretical peak performance versus memory bandwidth.
- Many applications in science and engineering consist of loop-based code that moves large amounts of data in and out of the CPU
 - On-chip resources tend to be underutilized and performance is limited only by the relatively slow data paths to memory or even disks.
- Any optimization attempt should therefore aim at reducing traffic over slow data paths

Balance & lightspeed estimates



- An estimate to assess the theoretical performance of a loop-based code is required. (single loop scope)
- The central concept is *balance*.

- Machine balance:

$$B_m = \frac{\text{memory bandwidth (GWords/sec)}}{\text{peak performance (GFlops/sec)}}$$

- Code Balance:

$$B_c = \frac{\text{data traffic (Words)}}{\text{floating point ops (Flops)}}$$

Balance & lightspeed estimates



- The expected maximum fraction of peak performance one can expect from a code with balance B_c on a machine with balance B_m is:
- **Lightspeed estimate**
$$l = \min\left(1, \frac{B_m}{B_c}\right)$$
- If $l \approx 1$, loop performance is not limited by bandwidth but other factors, either inside the CPU or elsewhere.

data path	balance
cache	0.5–1.0
machine (memory)	0.05–0.5
interconnect (high speed)	0.01–0.04
interconnect (GBit ethernet)	0.001–0.003
disk	0.001–0.02

Balance & lightspeed estimates



- Critical assumptions regarding this performance model:
 - The loop code makes use of all arithmetic units (multiplication and addition). If this is not the case, e.g., when only additions are used, one must introduce a correction term that reflects the ratio of MULT to ADD operations.
 - Code is based on double precision floating-point arithmetic. In cases where this is not true, one can easily derive similar, more appropriate metrics (e.g., words per instruction).
 - Data transfer and arithmetic overlap perfectly.
 - The system is in “throughput mode”, i.e. latency effects are negligible.
- More advanced performance model for real applications: **Roofline Model**

Balance & lightspeed estimates



- Example:

- Processor: 3.2GHz, 2 flops/cycle, 64-bit
- Memory: 6.4 GBytes/sec

```
for(i=0, i<N, i++)
    A[i] = B[i] + C[i] * D[i]
```

- Solution:

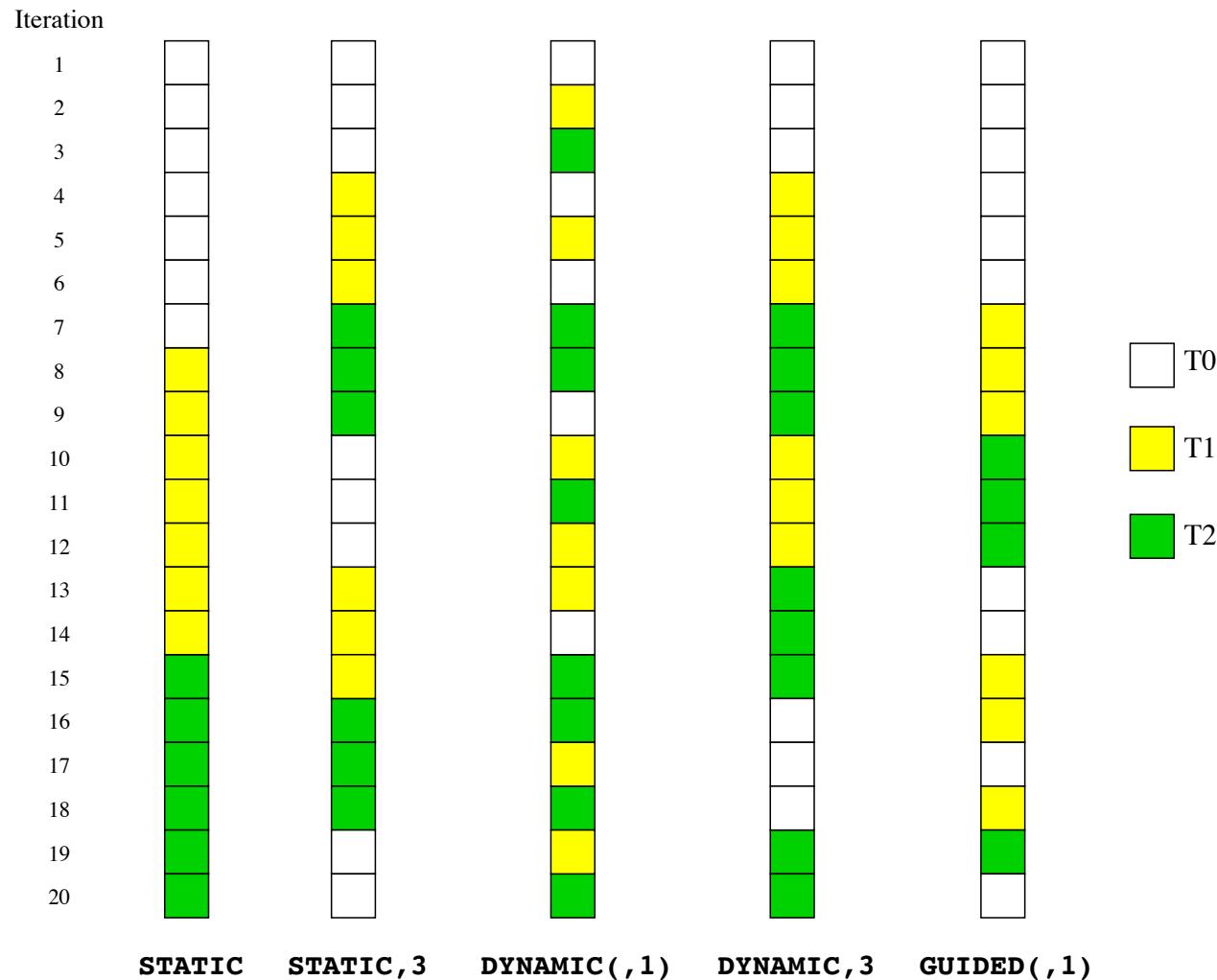
- 2 flops per iteration / 3 loads (B, C, D) / 1 store to A

$$B_m = \frac{\text{memory bandwidth (GWords/sec)}}{\text{peak performance (GFlops/sec)}} = \frac{6.4/8}{3.2 * 2} = 0.125 \text{ Words/Flops}$$

$$B_c = \frac{\text{data traffic (Words)}}{\text{floating point ops (Flops)}} = \frac{(3+1)}{2} = 2 \text{ Words/Flops}$$

$$l = \left(1, \frac{B_m}{B_c}\right) = \frac{0.125}{2} \approx 0.06$$

OpenMP Scheduling -- Recap



Load balance

- A chain is only as strong as its weakest link
- Duration of parallel execution is determined by thread with highest load
- Different loop scheduling strategies
 - Static scheduling
 - Division of iterations depends on
 - Number of iterations
 - Number of threads
 - Duration of each iteration is ignored
 - Works fine if each iteration contains the same amount of work
 - Often with simple domain decomposition plus non-sparse algorithms
 - Dynamic / guided scheduling
 - Division of iterations can adapt to irregular work loads
 - Incurs synchronization overhead

Scaling a sparse matrix



```
typedef struct {
    double *element; /* non-zero elements */
    int    *index;   /* index of non-zero elements */
    int    num_el;   /* number of non-zero elements */
} SPARSE_ROW;

SPARSE_ROW *rows;
[...]

for (i=0; i<num_rows; i++) {
    SPARSE_ROW r = rows[i];
    for (j=0; j < r.num_el; j++)
        r.element[j] = c * r.element[j];
}
```

- Program multiplies each matrix element with a constant

Scaling a sparse matrix (2)

- Naïve parallelization would result in a static schedule with most implementations

```
#pragma omp parallel for
    for (i=0; i<num_rows; i++) {
        SPARSE_ROW r = rows[i];
        for ( j=0; j<r.num_el; j++)
            r.element[j] = c * r.element[j];
    }
```

- Might perform well if each row has about the same number of non-zero elements
- If not then some threads might get larger amounts of work
- The ones with smaller amount would have to wait at the implicit barrier

Scaling a sparse matrix (3)

- With a schedule (*dynamic, 1*), each thread will be assigned one row (= 1 chunk) at a time
- Each thread will get the next row (i.e., chunk) only after finishing the previous one
- At the end slowest thread will have at most one row left to process
- Why not always dynamic scheduling?
 - Each work assignment incurs synchronization overhead
 - Changing chunk size is tradeoff decision
 - Chunk size 1 delivers best balance but highest overhead
 - Chunk size `num_rows / p` imitates static scheduling
 - Programmer has to find optimal medium
 - Dynamic scheduling can influence locality of data accesses

Scaling a triangular dense matrix

```
for (i=0; i<n; i++)
    for (j=i+1; j < n; j++)
        a[i][j] = c * a[i][j];
```

- Each iteration has a different amount of work
- But amount varies regularly
- Static schedule without chunk size will create load imbalance
- Static schedule with small chunk size will distribute chunks of decreasing size in a round-robin fashion
 - Almost even distribution of work
- Dynamic schedule would not improve load balance

Static vs. dynamic schedule



- Static schedule achieves good load balance
 - Same amount of work per iteration
 - Different amount of work – but difference follows regular pattern
 - Assumption: all threads enter the loop at the same time
- Dynamic or guided schedule
 - Irregular distribution of work across iterations
 - Threads enter the loop at a different time

```
#pragma omp sections nowait
[...]
#pragma omp for schedule(?)
for (i=0; i<n; i++)
[...]
```

- Programs tend to reuse data and instructions they have used recently
- **Temporal locality**
 - Recently accessed items are likely to be accessed in the near future
- **Spatial locality**
 - Items with addresses near to each other tend to be referenced close in time
 - Stride-1 memory references have perfect spatial locality
- **Challenge in a multiprocessor environment**
 - Restrict locality to a single processor
 - Accessing data that is in someone else's cache can be more expensive than accessing main memory
 - User must ensure that processors do not touch the same cache line

Zeroing a matrix

- Two options

- Column-wise

```
for (j=0; j<n; j++)
    for (i=0; i<n; i++)
        a[i][j] = 0.0;
```

- Doesn't exploit spatial locality in C

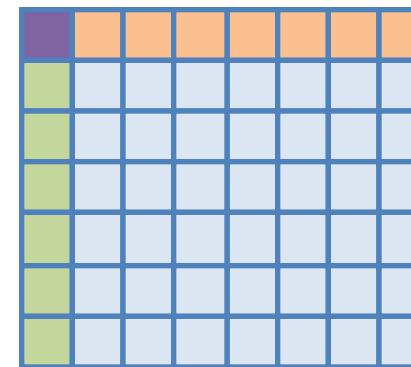
- Row-wise

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = 0.0;
```

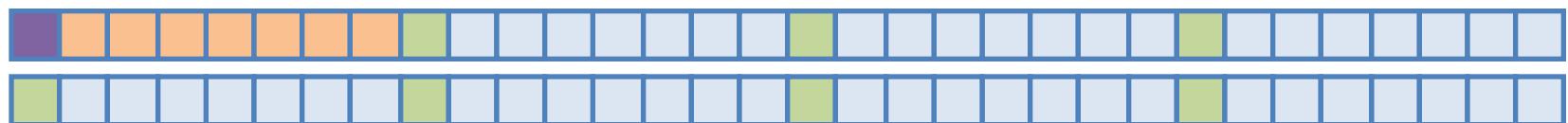
- Can be more efficient in C
- Some compilers might automatically interchange loops

Row-major and column-major order

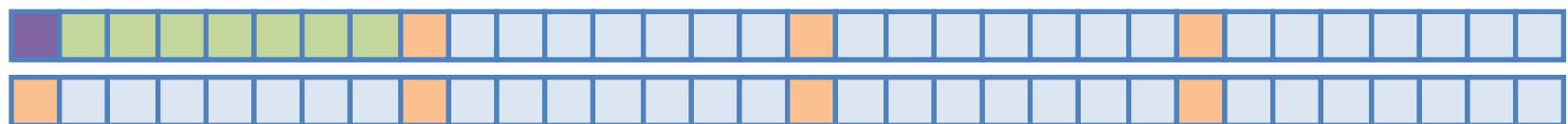
Two-dimensional matrix



Memory layout in C (row major)



Memory layout in Fortran (column major)



Locality and parallel loop schedules



- Scaling a sparse matrix

```
#pragma omp parallel for schedule(?)  
for (i=0; i<num_rows; i++) {  
    SPARSE_ROW r = rows[i];  
    for ( j=0; j<r.num_el; j++)  
        r.element[j] = c * r.element[j];  
}
```

- Assume that matrix is small enough to fit in the aggregate caches of all processors and that each portion is small enough to fit in a single cache
- After one scaling, each processor's caches will contain the matrix portion scaled by this processor
- What happens during a second scaling?

Scaling the matrix a second time



- Static scheduling
 - Every invocation of the scaling would divide the iterations of the loop in the same way
 - Each processor would get the same portion of the matrix to scale
 - The corresponding data would already be present in the cache
 - Memory accesses would be very fast
 - We might observe load imbalance
- Dynamic scheduling
 - No guarantee that a particular portion of the matrix would be assigned to the same processor
 - Each processor might get a different portion of the matrix, which resides in a different cache
 - Memory accesses would be much slower

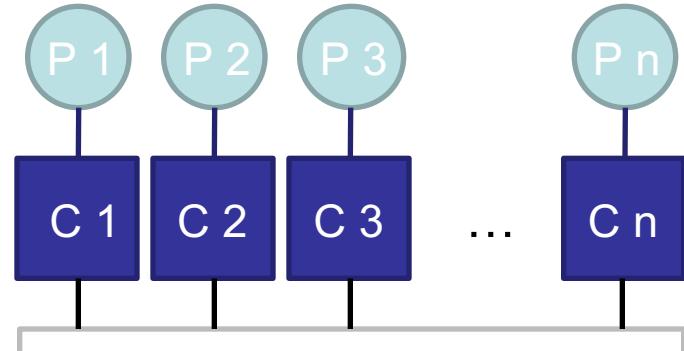
Locality effect of a dynamic schedule



- Depends on the size of each processor's portion of the data
 - If it fits in the cache, impact might be big
 - If it doesn't, impact might be small
 - The difference can be significant, so an experiment might be useful
 - Superlinear speedup possible
 - If the overall size of the data does not fit in a single cache but individual portions do
- Depends on degree and type of locality
 - Scaling a matrix
 - Temporal locality only across different invocations of scaling routine
 - Spatial locality (not affected by scheduling)
 - If there were a lot of temporal locality within a single invocation, scheduling would become less important

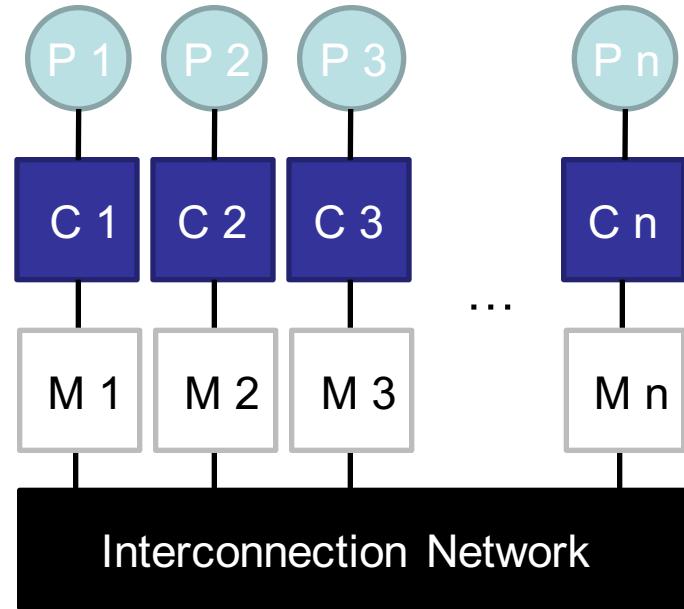
Memory architecture impact on locality

UMA architecture



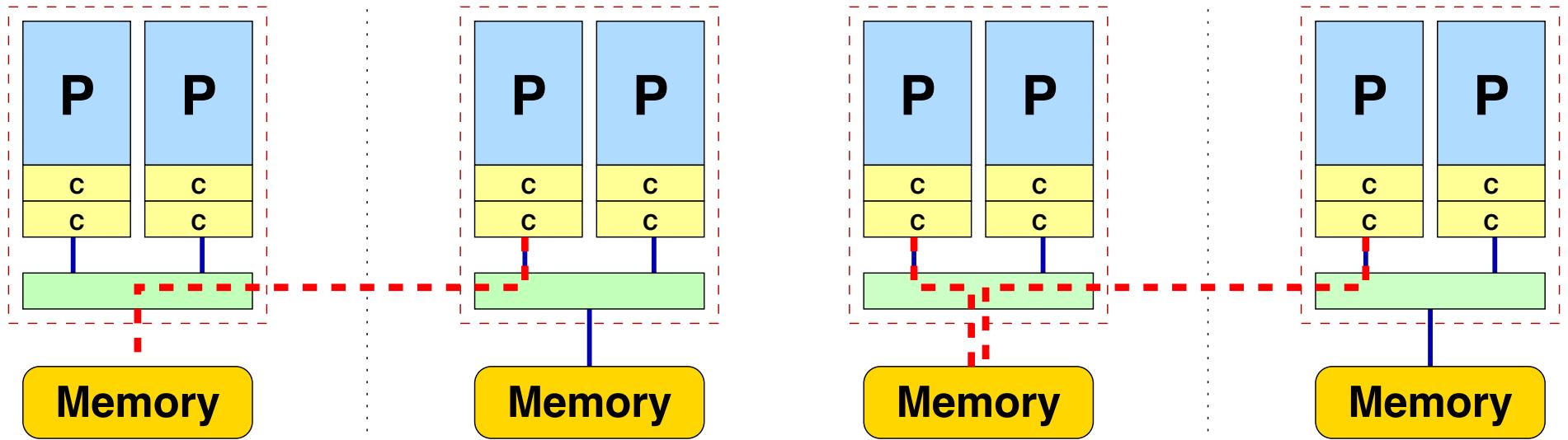
(cc)NUMA architecture

- Most multisocket systems today



Memory architecture impact on locality

Locality vs. Congestion



- Locality problem on a ccNUMA system.
- Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic.
- Congestion problem on a ccNUMA system.
- Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and non-local accesses.

Memory architecture impact on locality

Ensuring locality of memory access



- The data placement problem has two dimensions:
 - Make sure that memory gets *mapped* into the locality domains of processors that actually access them.
 - Threads or processes must be *pinned* to those CPUs which had originally mapped their memory regions in order not to lose locality of access.

Memory architecture impact on locality

Ensuring locality of memory access



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Initial mapping can be enforced in a portable manner on all current ccNUMA architectures.
- They support a *first touch policy* for memory pages: *A page gets mapped into the locality domain of the processor that first reads or writes to it.*
 - *Allocating* memory is not sufficient

Data initialization code that deserves attention on ccNUMA

Memory architecture impact on locality

Ensuring locality of memory access



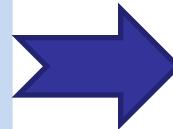
TECHNISCHE
UNIVERSITÄT
DARMSTADT

Improper data locality
Congestion

```
int N=1000000
Double A[N], B[N]

// executed on single LD
for (i=0; i<n; i++)
    A[i] = 0

// congestion problem
#pragma omp for
for (i=0; i<n; i++)
    B[i] = func(A[i])
```



Proper data locality
Without congestion

```
int N=1000000
Double A[N], B[N]

// distribute A on LDs
#pragma omp for
for (i=0; i<n; i++)
    A[i] = 0

#pragma omp for
for (i=0; i<n; i++)
    B[i] = func(A[i])
```

False sharing



Counting the even elements of an array

```
int my_id;
int local_sum[MAX_NUM_THREADS];
int global_sum = 0;

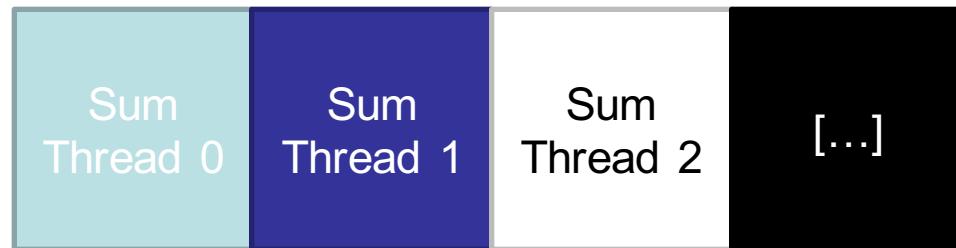
#pragma omp parallel private(my_id)
{
    my_id = omp_get_thread_num();

    #pragma omp for schedule(static)
    for (i=0; i<n; i++)
        if ( a[i] % 2 == 0 )
            local_sum[my_id] += 1;

    #pragma omp atomic
    global_sum += local_sum[my_id];
}
```

False sharing (2)

- Memory layout of array local_sum



- Different elements are used by different threads
- Likely that adjacent elements would be part of the same cache line
- Each time a processor updates its portion, it must first invalidate the line in all other processors' caches
- Subsequent references by other processors to different parts of the invalidated line will cause a miss

False sharing (3)

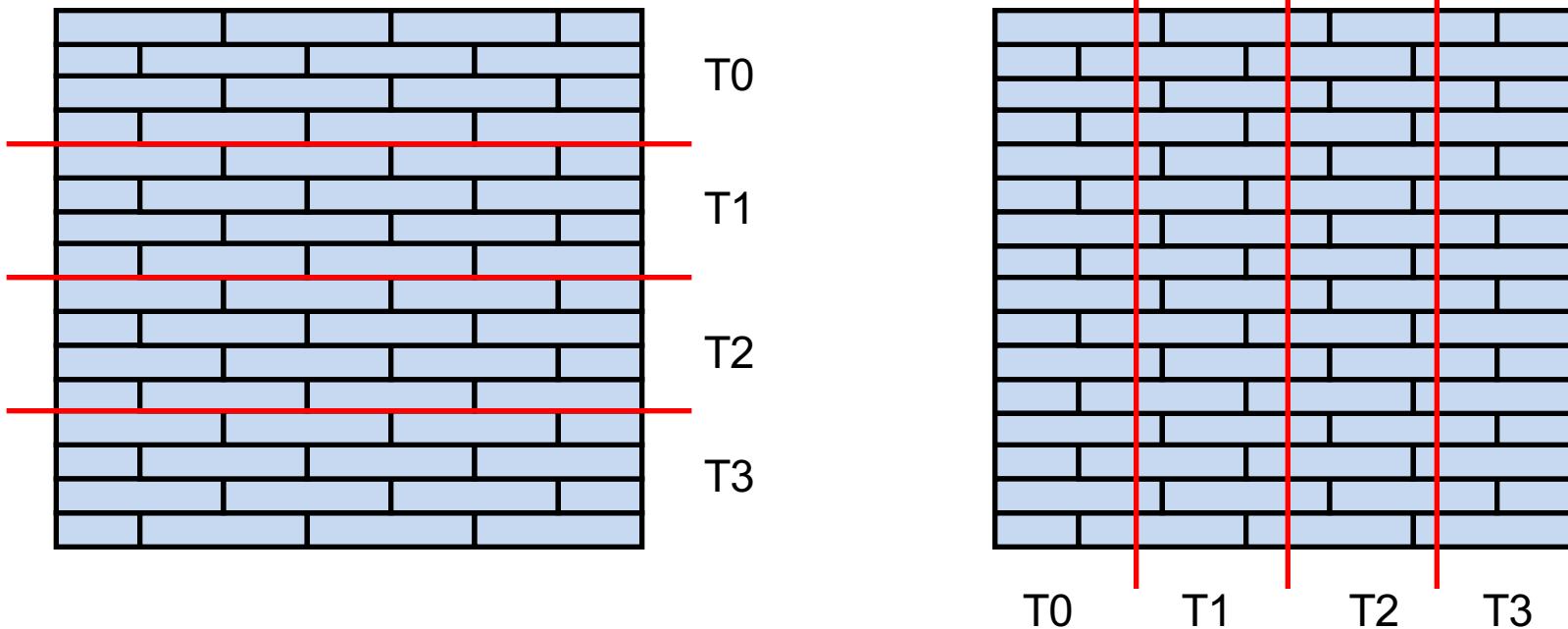


- Privatization / reduction can remove false sharing

```
#pragma omp parallel for schedule(static) reduction(+,sum)
for (i=0; i<n; i++)
    if ( a[i] % 2 == 0 )
        sum += 1;
```

- False sharing as a result of wrong domain decomposition
- C/C++
 - Row-major memory layout
 - Division across rows can cause false sharing
- Fortran
 - Column-major memory layout
 - Division across columns can cause false sharing

False sharing (4)



- False sharing potential wherever a cache line is divided
- Reduce false sharing by
 - Divide along row borders in C
 - Divide along column borders in Fortran

Inconsistent parallelization



```
for (i=0; i<n; i++)
    a[i] = b[i];
for (i=1; i<n; i++)
    a[i] = a[i] + a[i-1];
```

- Parallelization of the first loop is trivial
- Parallelization of the second loop is not possible due to a data dependence
- However, parallelization of the first loop only can have a negative effect on performance
 - If the arrays are small enough, parallelizing the first loop might distribute the arrays across the different processor caches
 - A serial second loop would have to bring the data back to the master's cache



- Implicit barriers at the end of parallel work-sharing constructs ensure that the enclosed work is finished at the time when the master resumes execution
- Barriers without special hardware support can be expensive and often include a significant amount of waiting time
- Avoiding barriers
 - Adding a nowait clause if it doesn't affect the program's correctness
 - Coalescing multiple loops into one if not prohibited by dependences

```
#pragma omp for
for (i=0; i<n; i++)
    a[i] = ...;
#pragma omp for
for (i=0; i<n; i++)
    b[i] = a[i] + ...;
```

```
#pragma omp for
for (i=0; i<n; i++)
    a[i] = ...;
    b[i] = a[i] + ...;
```

Critical sections

- Critical sections require expensive communication between threads
- Possible implementation of a critical section
 - Combination of hardware instructions LLSC (load-linked, store-conditional) to atomically update a memory location
 - Store fails if location has been modified since load
 - One thread holds the critical section and all others try to obtain it
 - Release is done using normal store to a specific memory location
 - Before the store, all other processors' cache entries are invalidated
 - Then all other processors read the new value using the linked load
 - Eventually, one of the others updates the location using the conditional store
 - Again, all other cache entries must be invalidated

Critical sections (2)

- Previous scenario is expensive because we assumed that all threads tried to enter the critical section simultaneously
 - If only one thread tries to gain access there is no communication
 - If there is no concurrent attempt to gain access, there is only communication between a thread and its successor

Limiting the cost of mutual exclusion

- Granularity of data items subject to mutual exclusion
 - Example: entering data into a binary tree
 - Use critical sections or locks
 - Locking the entire tree while updating
 - No two threads update any part of the tree simultaneously
 - All updates occur in a serial fashion even if changes are only local and affect distinct parts of the tree
 - Might reduce parallelism
 - Locking only distinct sections of the tree
 - Exploits parallelism while still ensuring correctness
 - Different locks for different purposes
- Number of exclusive accesses
 - Example: parallel sum reduction
 - Adding local sums at the end instead of updating a global sum all the time saves synchronization overhead

Further Reading



[1] Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)
Barbara Chapman, Gabriele Jost, Ruud Van Der Pas, 2008,
MIT Press