Introduction to Parallel Computing

Introduction, motivation, problems, models

Jesper Larsson Träff
TU Wien
Institute of Computer Engineering
Parallel Computing

©Jesper Larsson Träff

Parallel Computing

Premise:
Parallelism (as in parallel computers) is everywhere!
What to do with all these resources? How can we make use of them efficiently?

Octa-core mobile (2016, Samsung Galaxy 7)

GPU appliances

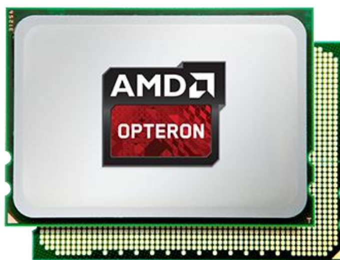
Octa-core laptop, Intel Xeon Processor E5-2690, 2012

Parallel Computing

Premise:
Parallelism (as in parallel computers) is everywhere!
What to do with all these resources? How can we make use of them efficiently?

Intel “Skylake”: 4-28 cores

The processors in our new cluster/server to be used in this lecture



AMD “Naples”: 8-32 cores

Another server, perhaps in later courses

November 2012: Cray Titan, 560,640 cores, 17.5 PFLOPS

Parallel Computing

Premise:
Parallelism (as in parallel computers) is everywhere!
What to do with all these resources? How can we make use of them efficiently?

June 2020: Fujitsu Fugaku, 7,630,848 cores, 442,010 TFLOPS

June 2016: Sunway TiahuLight, 10,649,600 cores, 93 PFLOPS

November 2018: IBM/Nvidia, 2,397,824 cores, 143.5 PFLOPS

June 2011: Fujitsu K, 705,024 cores, 11 PFLOPS

June 2012: IBM BlueGene/Q, 1,572,864 cores, 16 PFLOPS

Parallel Computing

Premise:
Parallelism (as in parallel computers) is everywhere!
What to do with all these resources? How can we make use of them efficiently?

Fact: Since ca. 2010, there are (next to) no sequential computer systems anymore (multi-core, GPU accelerated, …)

Wide range: From a few to >1,000,000 processors

How to use these parallel computers?
a) Parallelize the applications (efficiently)
b) Have enough independent applications that can run at the same time (concurrently) to keep system busy

Fact: Since ca. 2010, there are (next to) no sequential computer systems anymore (multi-core, GPU accelerated, …)

Why?

Premise:
"We" want to solve larger, more complex problems faster (better, cheaper, …)

Challenge a): Parallelizing individual applications to exploit parallel hardware (multiple cores)

Parallel Computing challenge

Lot’s of similar hype/panic around 2005-2010 and still

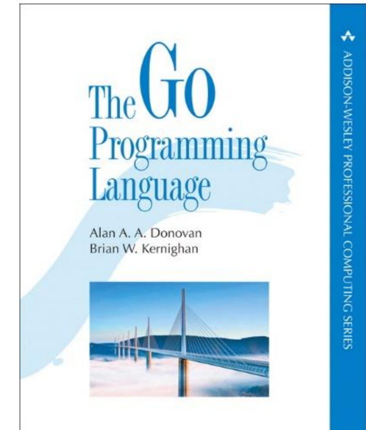
The Free Lunch Is Over
A Fundamental Turn Toward Concurrency in Software
By Herb Sutter
The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.
This article appeared in ***Dr. Dobb's Journal,30(3), March 2005***

“Free lunch” (as in “There is no such thing as a free lunch”)

Exponential increase in single-core performance: No software changes needed to exploit the faster processors, no reason to bother with exploiting parallelism

…and no reason to produce good, efficient code

10 years later (Donovan, Kernighan: The GO Programming language, Addison-Wesley, 2016):



Concurrent programming, the expression of a program as a composition of several autonomous activities, **has never been more important than it is today**. … use concurrency … to exploit a modern computer's many processors, which every year grow in number but not in speed.

… **reasoning about concurrent programs is inherently harder than about sequential ones**, and intuitions acquired from sequential programming may at times lead us astray.

But parallel computing is concerned with efficiency and performance:

~~Concurrent~~ **Parallel programming**, the expression of a program as ~~a composition of~~ several ~~autonomous~~ **parallel activities**, has never been more important than it is today. … use ~~concurrency~~ **parallelism** … to exploit a modern computer's many processors **efficiently**, which every year grow in number but not in speed.

… reasoning about ~~concurrent~~ **the performance of parallel programs** is inherently harder than about sequential ones, and intuitions acquired from sequential programming may at times lead us astray.

Processor performance:

Number of (some kind of) instructions that can be carried out per unit of time

Examples:

- FLOPS: FLoating-Point OPerations per Second (IEEE 64-bit)
- IPS: Instructions per Second

Floating-point? Integer? Branches? Memory accesses? Cache misses? Comparisons?

Measuring/accounting for performance:

- Which kinds of operations matter most in application? Which correspond best to, e.g., running time
- Which kind of operations are most expensive?
- Count number of these per time unit (seconds)

- Measure (empirical)?
- Specification?

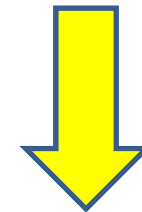
Why FLOPS? Historically, floating-point operations were expensive. And accountedS for most of the work in numerical applications.

Metric prefixes (SI)

	prefix	factor
	Mega	10^6 (Million)
	Giga	10^9 (Billion)
Gr.	Tera	10^{12} (Trillion)
	Peta	10^{15}
	Exa	10^{18}
Ital.	Zetta	10^{21}
Lat.	Yotta	10^{24}
Gr.		

MegaFLOPS, MIPS

GigaFLOPS



Parallel and High-Performance Computing

ExaFLOPS

Current HPC systems: 10-100 PetaFLOPS, see www.top500.org

“Free Lunch” aka Moore’s “law”

Exponential growth in processor performance often referred to as

Moore‘s “law” (popular version):
Sequential processor performance doubles every 18 months

Moore’s “Law” is (only) an empirical observation/extrapolation.

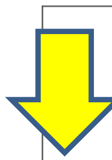
Not to be confused with physical “law of nature” or “mathematical law” (e.g., Amdahl’s Law, see later)

Source: Wikipedia.org, co-founder of Intel

Gordon Moore: Cramming more components onto integrated circuits. Electronics, 38(8), 114-117, 1965

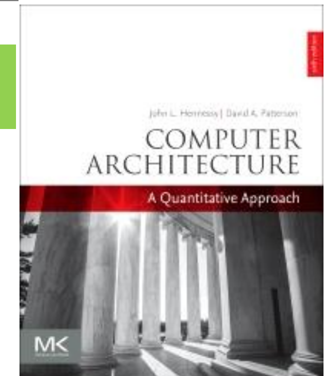
From: Hennessy/Patterson, Computer Architecture, A Quantitative Approach

log-scale



History of Processor Performance

Measured: SPEC



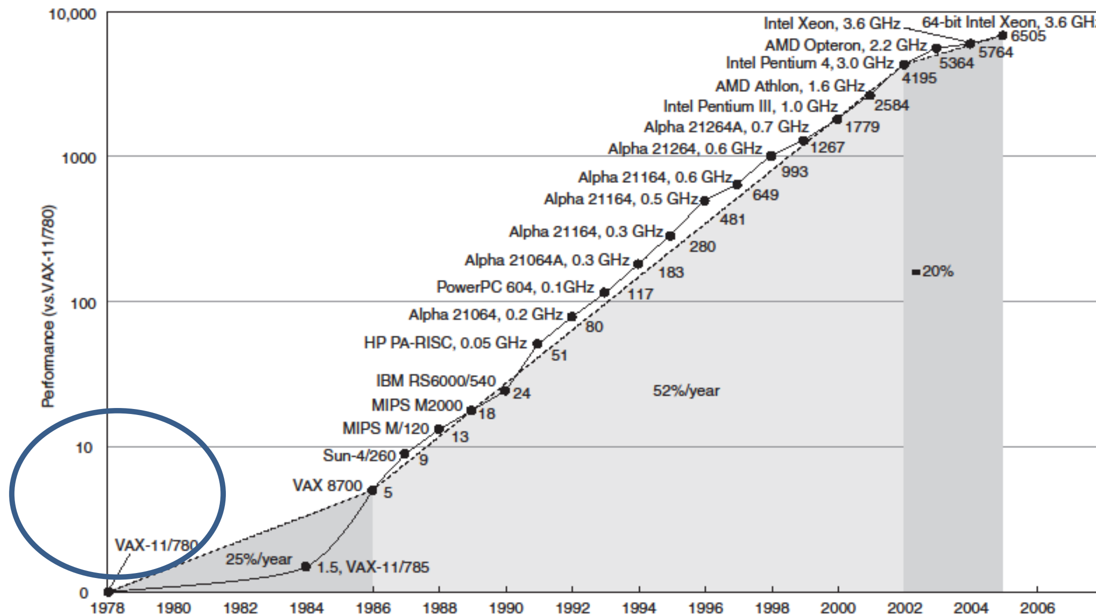


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year. Copyright © 2009 Elsevier, Inc. All rights reserved.

1

Tuesday, April 24, 12

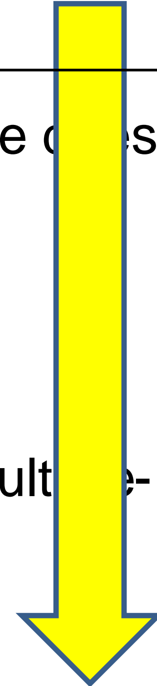
<u>2006</u>: Factor 10^3-10^4 increase in integer performance (SPECint) over 1978 high-performance processor

Some reasons for Moore's "law" phenomenon

1. Increased clock frequency (4MHz ca. 1975 to 4GHz ca. 2005; factor 1000, 3 orders of magnitude)
2. Miniaturization, smaller features sizes, more transistors ("technology")

3. Inventions in computer architecture, increased processor complexity: Deep pipelining requiring branch prediction and speculative execution; processor ILP extraction (need transistors)
4. Multi-level caches (need transistors)

But you can write much more efficient/better programs if you understand these things



Increase in clock-frequency alone (factor 100-1000?) alone does not explain performance increase:

- Deep pipelining
- Branch prediction
- Caches
- Superscalar execution (>1 instruction/clock) through multiple-functional units, ILP (Instruction Level Parallelism), …
- Out-of-order execution, speculative execution
- Simplified/better instruction sets (for compiler)

Mostly fully transparent, at most compiler needs to care: "free lunch"

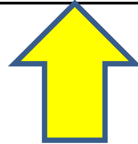
- Data parallelism through SIMD units (same instruction on multiple data/clock)
- SMT/Hyperthreading

Programmer and compiler have to care

Processor performance:

Number of (some kind of) instructions that can be carried out per unit of time

(Number of Instructions per Clock) x (Clock Frequency)



Superscalar processor: ≥ 1
SIMD/vector processor: ≥ 1 (e.g., 512-bit SIMD = 8 64-bit operations/clock)

Caveat:
Processor's point of view. Memory system must be able to deliver data fast enough to keep processor busy

Memory system performance:

Latency: Number of clock cycles to deliver one unit of data (Byte, Word, …)

Bandwidth: Number of units that can be delivered per unit of time (second, clock cycle)

Latency and bandwidth determined by the structure of the memory system (caches, banks, ports, controllers, …), and where the data are in memory

Moore‘s “law” (“Free lunch”) effectively killed parallel computing in the 90ties: To increase performance by an order of magnitude (factor 10 or so), just wait 3 processor generations, that is 4.5 to 6 years

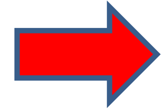
Also: Steep increase in performance/€

Parallel computers, typically relying on processor technologies a few generations old, and taking years to develop and market, could not keep up

Performance/€ ??

Parallel computers were not commercially viable in the 90ties

…and not only commercially

Still commercially tough 

90ties: Companies went out of business, systems disappeared

Commercial activity

Thinking Machines CM2

MasPar

IBM

INMOS Transputer

Thinking Machines CM-5

Intel iPSC

SB-PRAM

1979 1985 1992 2006

90ties: Companies went out of business, systems disappeared

The lost decade: A lost generation

Commercial activity

Thinking Machines CM2

Ma Par

IBM

Thinking Machines CM-5

INMOS Transputer

Intel iPSC

SB-PRAM

1979 1985 1992 2006

Research activity

TU Wien 2011

PRAM: Parallel RAM

Parallel computing disappeared from mainstream CS

©Jesper Larsson Träff

Informatics

What is an empirical “law”?

- Observation
- Extrapolation
- Forecast
- Prediction
- Self-fulfilling prophecy
- (Political) dictate

but can become a real Law when reasons behind are understood (mechanism)

Observed correlations are not Laws

“Rather than becoming something that chronicled the progress of the industry, Moore's Law became something that drove it.”
Gordon Moore, according to wikipedia.org

„…a self-fulfilling prophecy… nobody can afford to put a processor or machine on the market that does not follow it“, HPPC 2009

Peter Hofstee, IBM Cell co-designer

What is an empirical “law”?

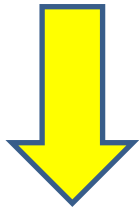
- Observation
- Extrapolation
- Forecast
- Prediction
- Self-fulfilling prophecy
- (Political) dictate

Two types of Laws:

- Mathematical Law (aka Theorem, Proposition, …): Necessary relationship that can be derived analytically from premises/axioms (e.g.: Amdahl’s Law, see later)
- Non-analytic, empirical law: Explained, necessary relationship with predictive power, with supporting, explanatory theory, empirically falsifiable (but not falsified) and supportable

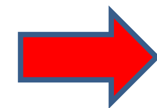
Moore's "law" in the "Free lunch" version ended ca. 2005:

- Clock speed reached limit around 2003 (no 4GHz processors)
- Power consumption limit from 2000
- Instructions/clock limit late 90ties



Single-core performance has not increased (and will most likely not increase) significantly (= exponentially) since ca. 2005

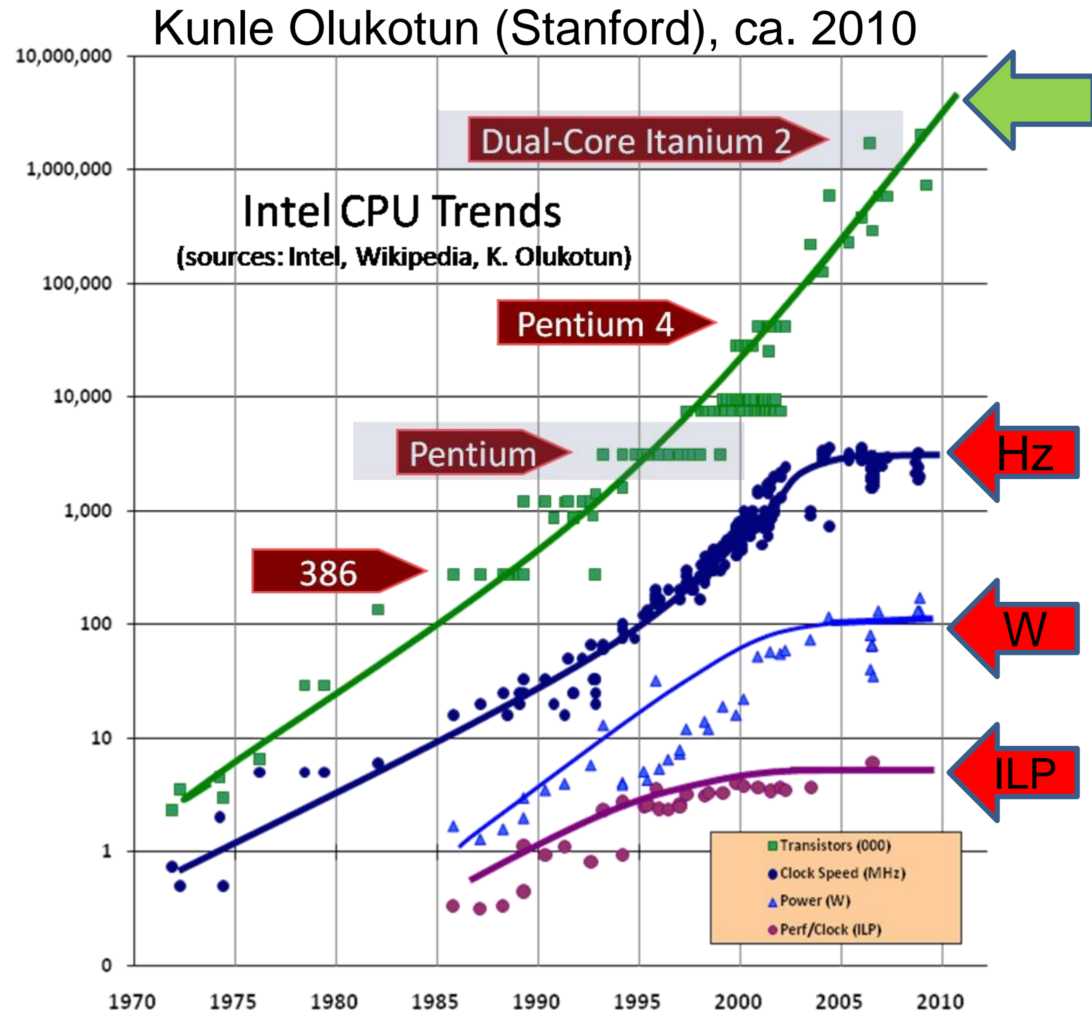
Moore's "law" (popular version) was no Law

"Free lunch" over: End of a certain business model  Panic

"Free lunch" was over…
ca. 2005

- Clock speed limit around 2003
- Power consumption limit from 2000
- Instructions/clock limit late 90ties

But:
Numbers of transistors/chip can continue to increase (>1Billion)



Parallel Computing
TU WIEN Informatics

Moore‘s “law” (what Moore originally observed):
Transistor counts double roughly every 12 months (1965 version); every 24 months (1974 version)

Microprocessor Transistor Counts 1971-2011 & Moore’s Law

curve shows transistor count doubling every two years

Transistor count

Date of introduction

What are all these transistors used for?

Computer Architecture challenge: What is the best way to use transistors? The art of tradeoffs(!)

So few transistors (in the thousands) needed for full microprocessor

Gordon Moore: Cramming more components onto integrated circuits. Electronics, 38(8), 114-117, 1965

Kunle Olukotun, 2010, Karl Rupp (TU Wien), 2015

40 Years of Microprocessor Trend Data

10^7 10^6 10^5 10^4 10^3 10^2 10^1 10^0

Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

1970 1980 1990 2000 2010 2020

Year

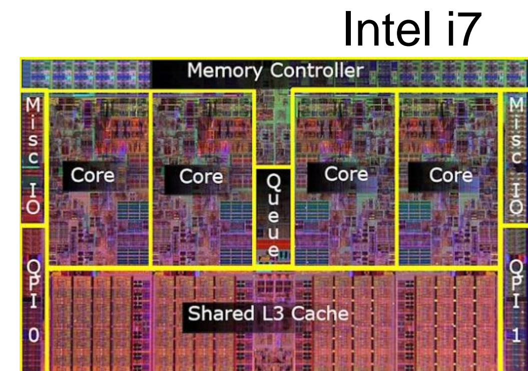
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Computer architecture challenge

Moore's "law" (exponential growth in number of transistors version) can continue for some time

How to use all these transistors efficiently?

Intel i7



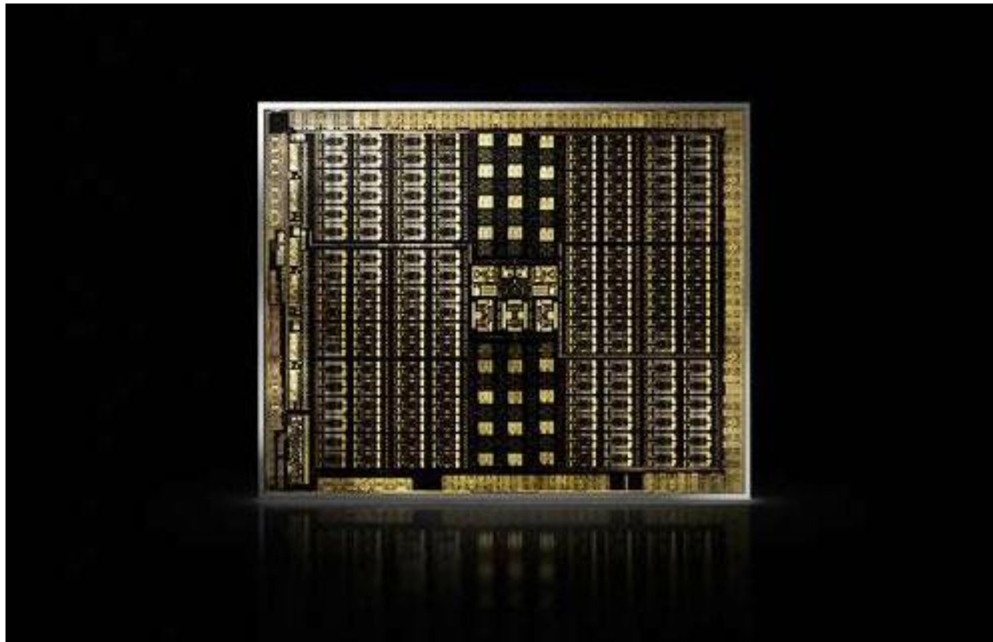
Since early 2000's: Transistors used for

- More cache
- More cores
- More threads (hyperthreading, SIMD)

Performance increase must come through parallelism

How to use all these transistors efficiently?

A different tradeoff/answer: GPU architecture (here: nvidia Turing)



From www.tomshardware.com

Design decisions:

- Many (many) simple cores
- Hardware support for extremely large number of threads
- Lower clock frequency (~1.5GHz)
- Complex, hierarchical memory system

How to use all these transistors efficiently?

Computer architecture is an open area of research (should be)

A different tradeoff/answer: vector co-processor

But (commercially) tough!!

NEC SX-Aurora (ca. 2019) traditional vector processor:

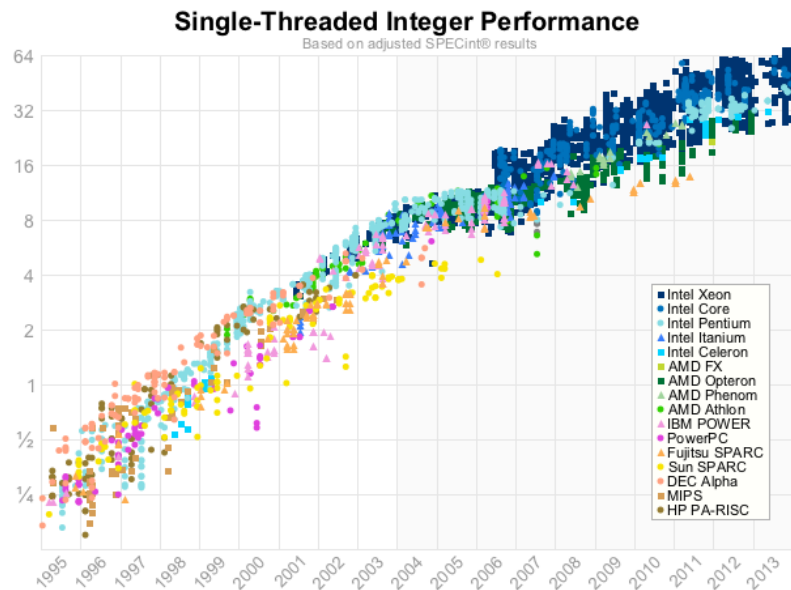
- Powerful SIMD vector processors, many, parallel pipelines
- Banked, on-chip memory system
- Lower clock frequency (1.6GHz)

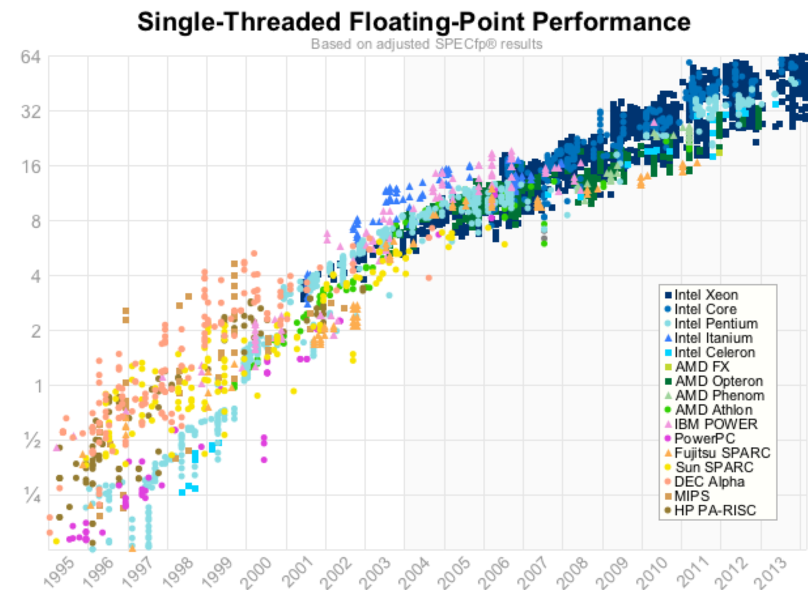
Will this take off?

Much is possible with FPGA, ASIC

Hot Chips 2018, www.hotchips.org

Single-core performance does still increase, somewhat, but much slower… Henk Poley, 2014, www.preshing.com





Average, normalized, SPEC benchmark numbers, www.spec.org

Moore's law (exponential growth in number of transistors version) can continue for some time (but exponential growth cannot continue forever!?)

Computer Architecture challenge: How to use these transistors efficiently?

Tradeoff:
Simpler (less complexity, fewer transistors, less power) and slower (less power) cores in order to get more cores on chip

⇒ Even better parallel computing solutions required

Terminology…

Core: The smallest unit capable of executing instructions according to a program (used to be called “processor”, “processing unit”)

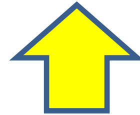
Processor: Collection of one or more cores on a single chip (as in “multi-core processor”)

Sometimes used:
Multi-core: handful of (standard CPU) cores (2-32)
Many-core: a lot of cores, often in special-purpose processors (GPU)

Parallel (multi-core) processor performance:

Number of (some kind of) instructions that can be carried out per unit of time

(Number of cores) x (Number of Instructions per Clock) x (Clock Frequency)

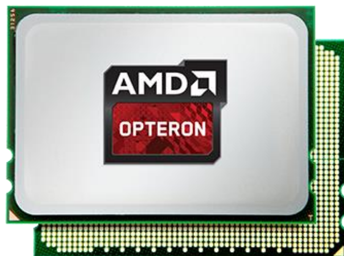


Superscalar processor: ≥ 1
SIMD/Vector processor: ≥ 1 (e.g., 512-bit SIMD = 8 64-bit operations/clock
Multi-core: > 1 core/processor, possibly many processors

Caveat:
Processor's point of view. Memory and communication system must be able to deliver data fast enough to keep processor busy

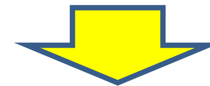
Parallel Computing challenge, concrete, practical

Solve some problems faster on p cores than on just one…



Issues:

- Parallel computers are diverse, very different architectures
- No commonly agreed upon "bridging" model for designing, analyzing and implementing algorithms



Many different paradigms (models), many different programming languages & interfaces

©Jesper Larsson Träff

Parallel Computing challenge, theoretical, model-driven

Informal model: p processor-cores, some means for exchanging information (memory, communication network, …) and coordination

Problem P on input I:

- How much faster can P(I) be solved with p processor-cores instead of 1 processor-core? Can P(I) be solved better with p processor-cores?
- How? New algorithms? New techniques?
- Can all problems be solved faster? How much faster?
- Are there problems that cannot be solved faster with more processors?
- Which assumptions are reasonable?
- Does parallelism give new insights into nature of computation?

“Bridging models”, from theory to practice

Computational/Architecture/Machine model (formally) describes components of processor (ALU, memory, interconnect), and defines how programs are executed and at what costs

- Design and analysis algorithms, account for costs (Example: Worst-case asymptotic complexity)
- Objectifies costs (good algorithm has low model costs, best algorithm has lowest possible model costs)

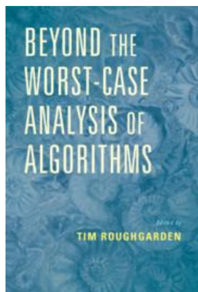
Model costs translate reasonably into/predicts performance on wide range of actual hardware

Leslie G. Valiant: A Bridging Model for Parallel Computation. Commun. ACM 33(8): 103-111 (1990)

Model costs translate reasonably into/predicts performance on wide range of actual hardware

Minimum requirement for a good bridging model:

If Algorithm A performs better than Algorithm B in model, then the implementation of Algorithm A performs better than the implementation of Algorithm B on the applicable hardware



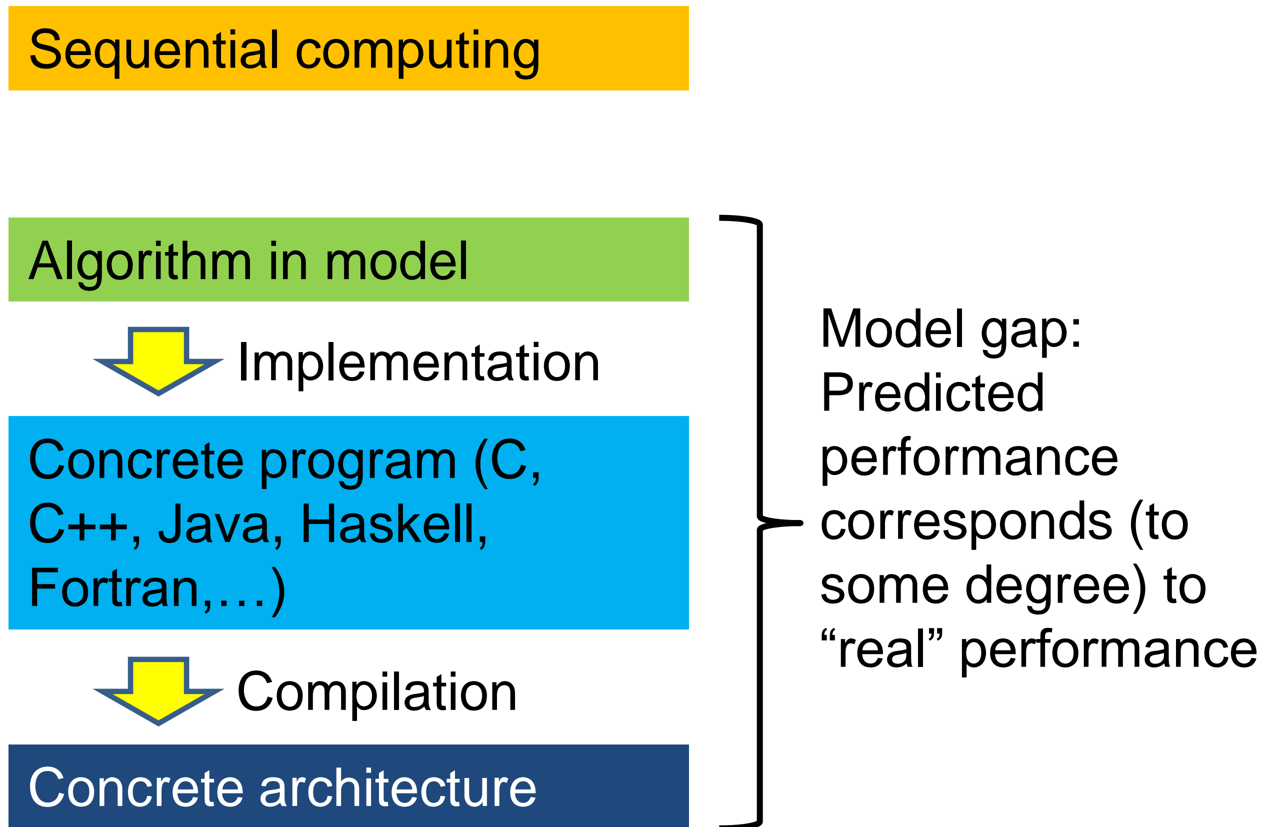
Tim Roughgarden (ed): Beyond worst-case analysis of algorithms. Cambridge University Press, 2021.

Computational/Architecture/Machine model (formally) describes components of processor (ALU, memory, interconnect), and defines how programs are executed and at what costs

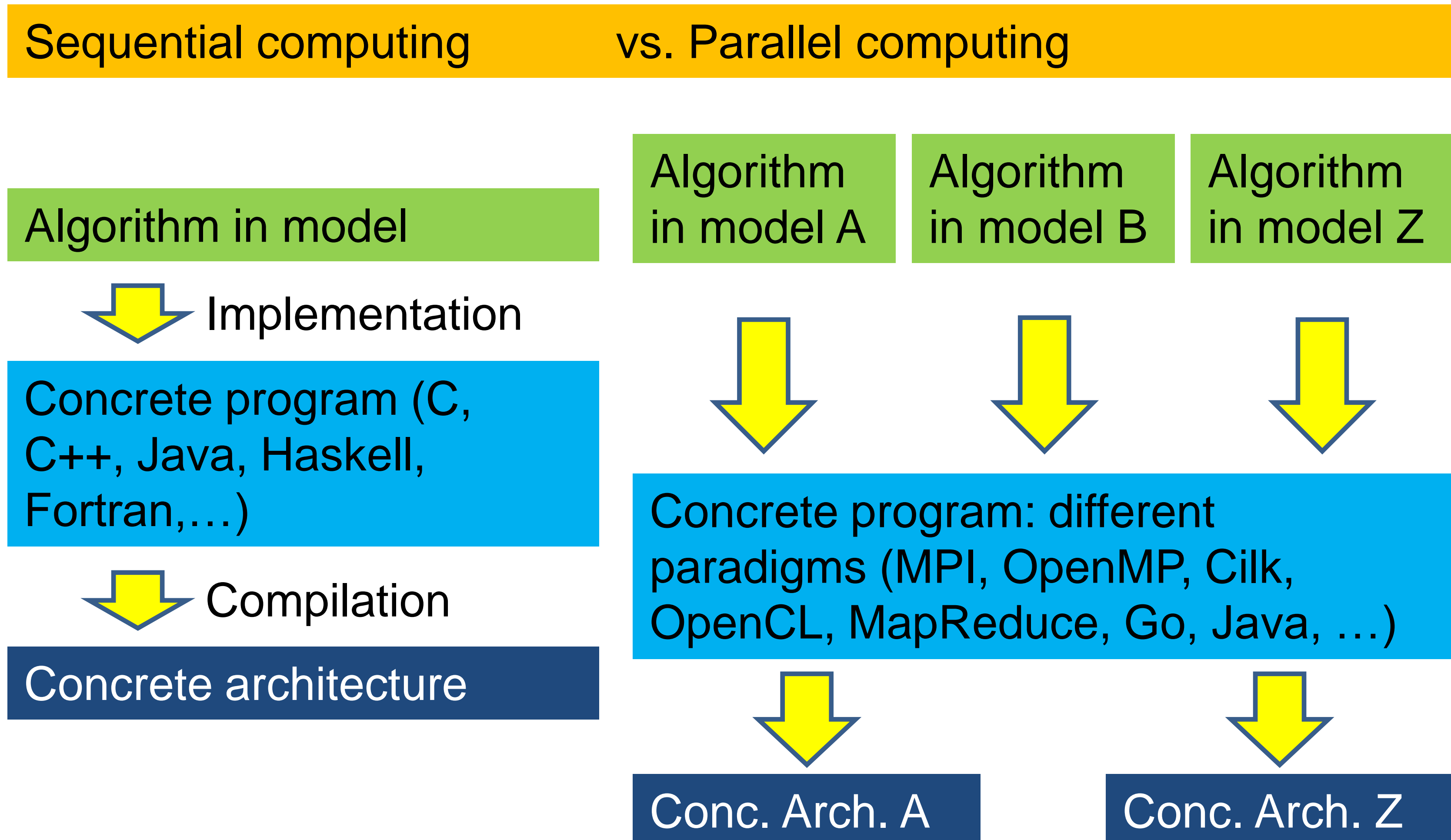
Good model

- abstracts unnecessary detail,
- accounts for the essentials,
- leads to interesting results, algorithms and lower bounds,
- and “bridging” works

Sometimes “just” informal mental model (“intuition”) that guides design choices (“this loop is better than that, because…”)



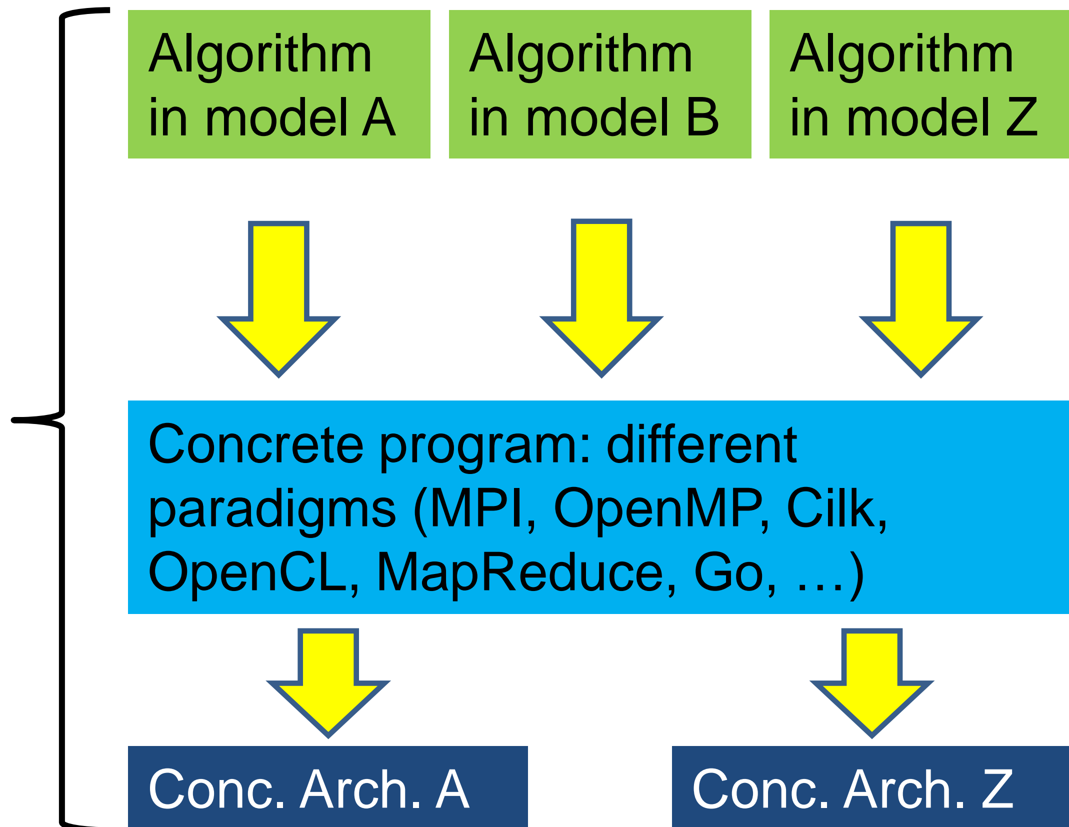
Model gap: Predicted performance corresponds (to some degree) to "real" performance


Sequential computing
vs. Parallel computing
Algorithm in model
Implementation
Concrete program (C, C++, Java, Haskell, Fortran,…)
Compilation
Concrete architecture
Algorithm in model A
Algorithm in model B
Algorithm in model Z
Concrete program: different paradigms (MPI, OpenMP, Cilk, OpenCL, MapReduce, Go, Java, …)
Conc. Arch. A
Conc. Arch. Z

Parallel computing challenges

Model(s) gap:

- Many different models, no single, agreed upon model
- Many different paradigms, languages and frameworks
- Many different architectures, highly diverse
- No good “bridging” properties

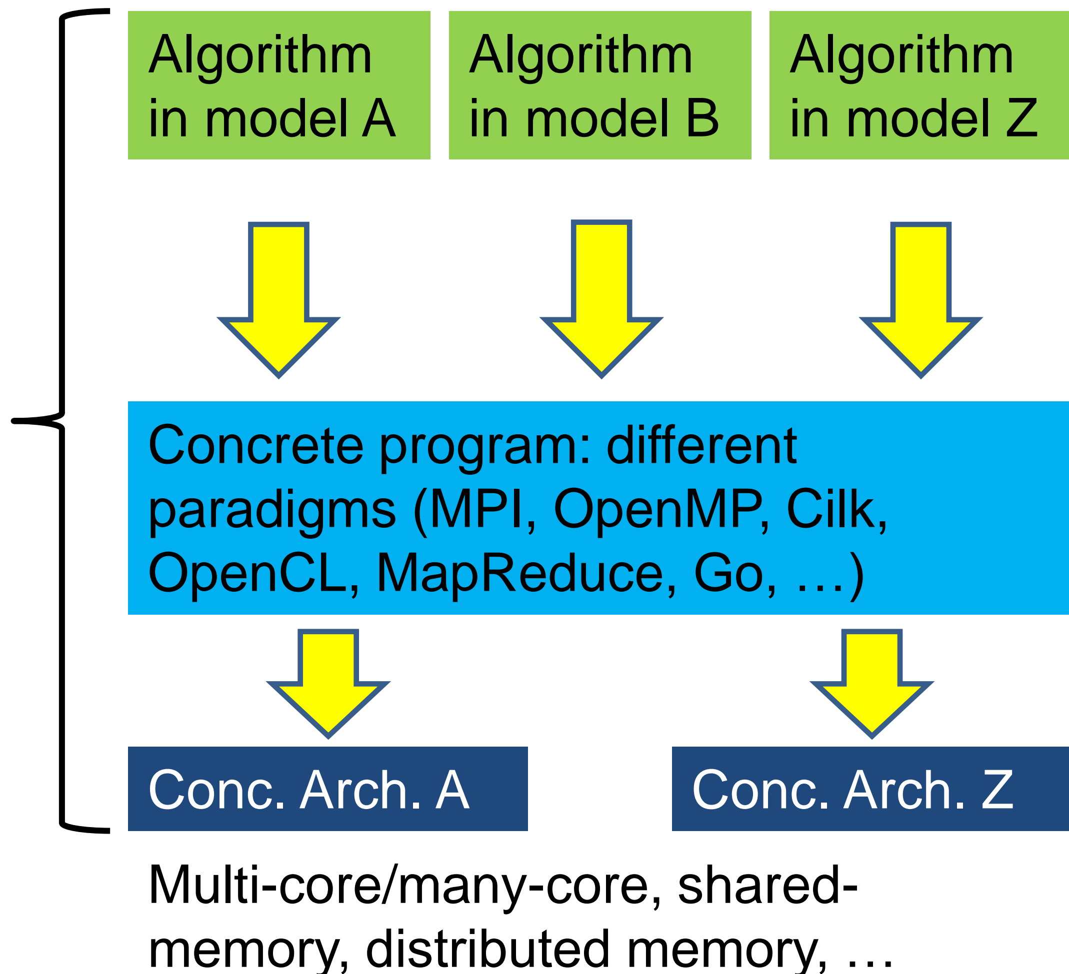


Multi-core/many-core, shared-memory, distributed memory, GPU, …

Parallel computing challenges

Predicted parallel performance often does not correspond to measured parallel performance

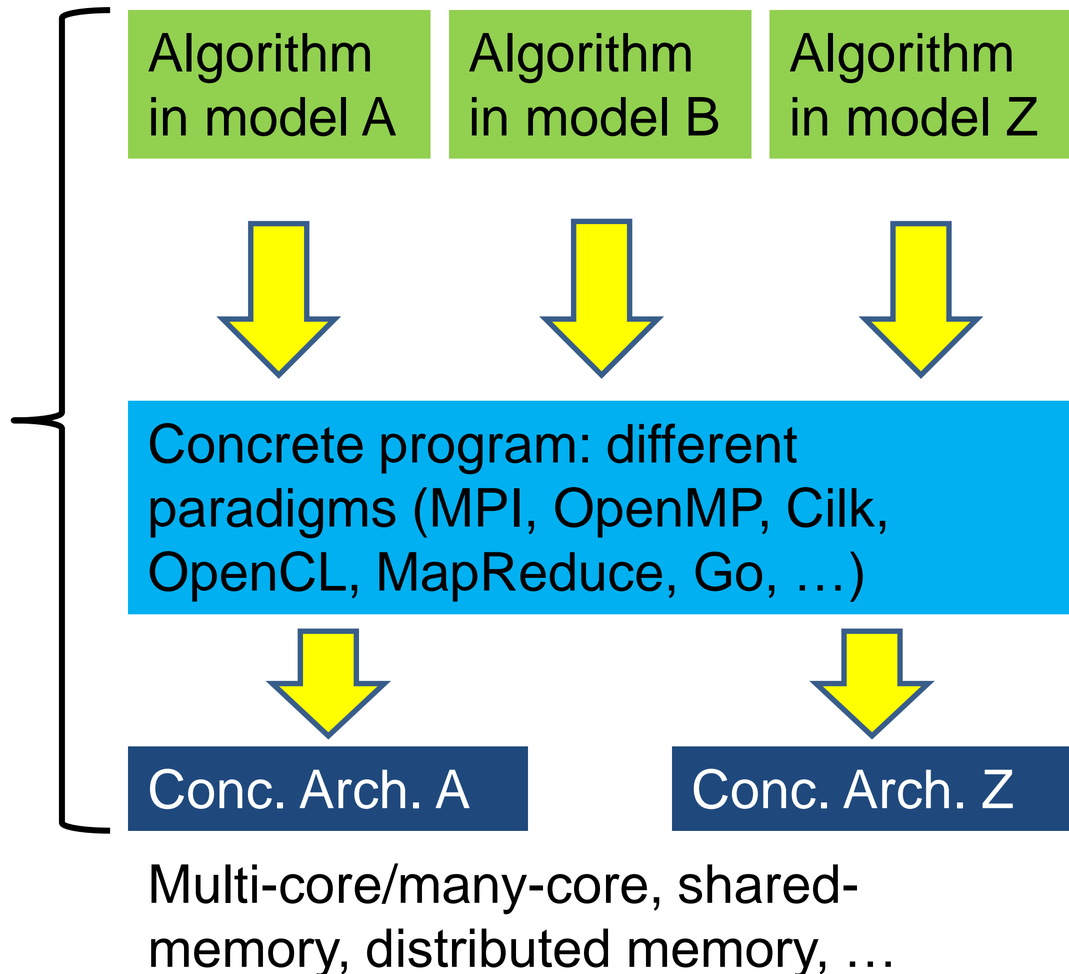
Designing, analyzing and benchmarking parallel algorithms is a skill (“The Art of…”) that requires experience



Parallel computing challenges

Portability problem:
Program written in language A for architecture A may not work for architecture Z

Performance portability problem:
Well performing algorithm/program for architecture A designed in model A may not perform well on architecture Z



Parallel computing: Definitions

Challenge a): Parallelizing single applications to exploit parallel hardware (multiple cores)

"single applications": Focus on computational problems
"exploit": Solving faster/better (than previous best solution), utilizing the parallel hardware well

Parallel computing:
The discipline of efficiently utilizing dedicated parallel resources (processors, memories, …) to solve individual, given computational problems.

Typical concerns:
Solving (difficult) problems faster!

Specifically:
Parallel resources with significant inter-communication capabilities, for problems with non-trivial communication and computational demands

Typical keywords: Tightly coupled, dedicated parallel system; multi-core processor, GPGPU, High-Performance Computing (HPC), …

Distributed computing:
The discipline of making independent, non-dedicated resources available and cooperate toward solving specified problem complexes.

Typical concerns:
Correctness, availability, progress, security, integrity, privacy, robustness, fault tolerance, …

Specifically/typically:
No centralized control

Typical keywords/areas: Grid, cloud, internet, agents, autonomous computing, mobile computing, IoT, …

Concurrent computing:
The discipline of managing and reasoning about interacting processes that may (or may not) progress simultaneously

Typical concerns:
Correctness (often formal), e.g. deadlock-freedom, starvation-freedom, mutual exclusion, fairness, …

Typical keywords: Operating systems, synchronization, interprocess communication, locks, semaphores, autonomous computing, process calculi, CSP, CCS, pi-calculus, lock/wait-freeness, …

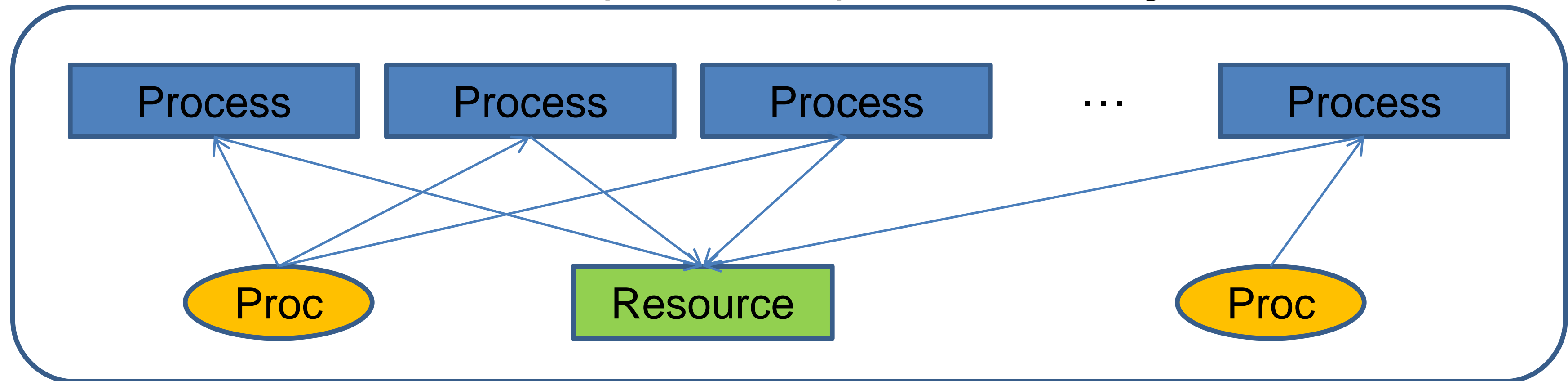
The three disciplines Parallel computing, Distributed computing, and Concurrent computing are intimately related:

- The same problem can often be viewed from all three perspectives, a matter of degree and focus.
- Methods and solutions from one discipline very often relevant to the others

Our focus: Problem and performance oriented parallel computing!

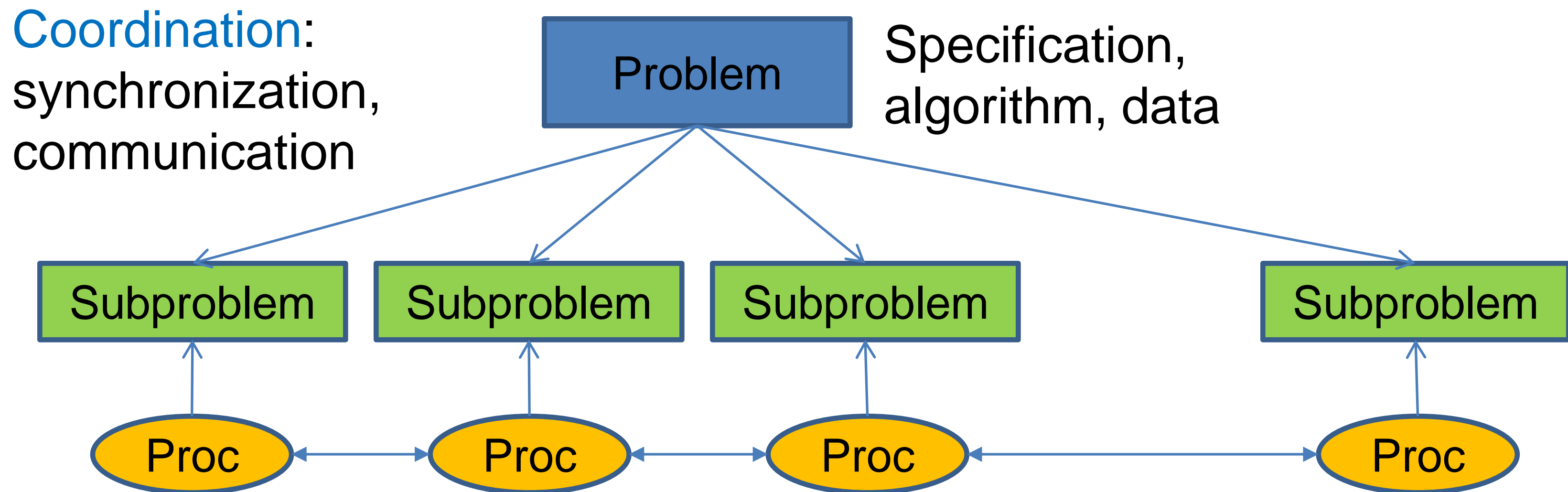
Parallel vs. Concurrent computing

Given problem: Specification, algorithm, data



Memory (locks, semaphores, data structures), device, …

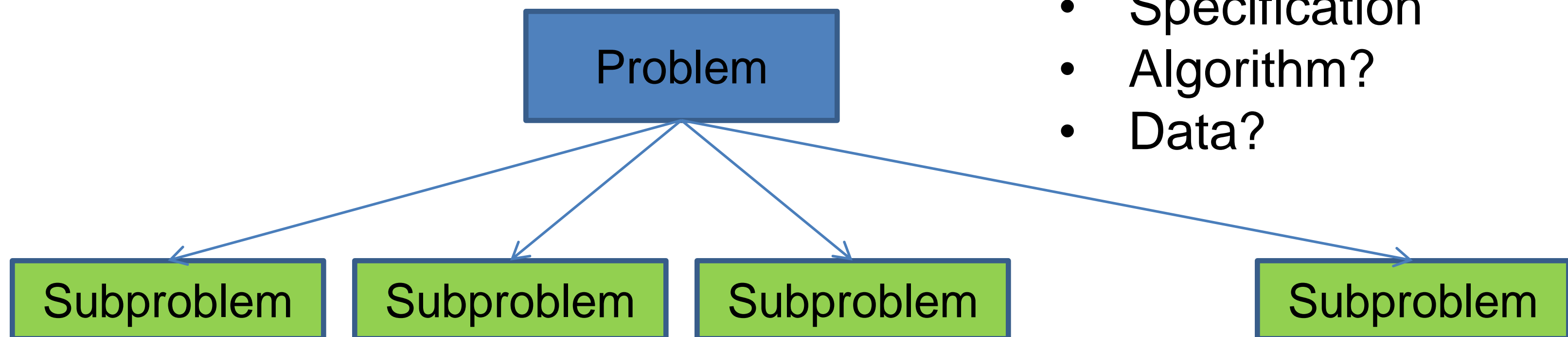
Concurrent computing: Focus on coordination of access to/usage of shared resources (to solve given, computational problem)



Parallel computing: Focus on dividing given problem (specification, algorithm, data) into subproblems that can be solved by dedicated processors (in coordination)

The problem of parallelization

How to divide given problem into subproblems that can be solved in parallel?

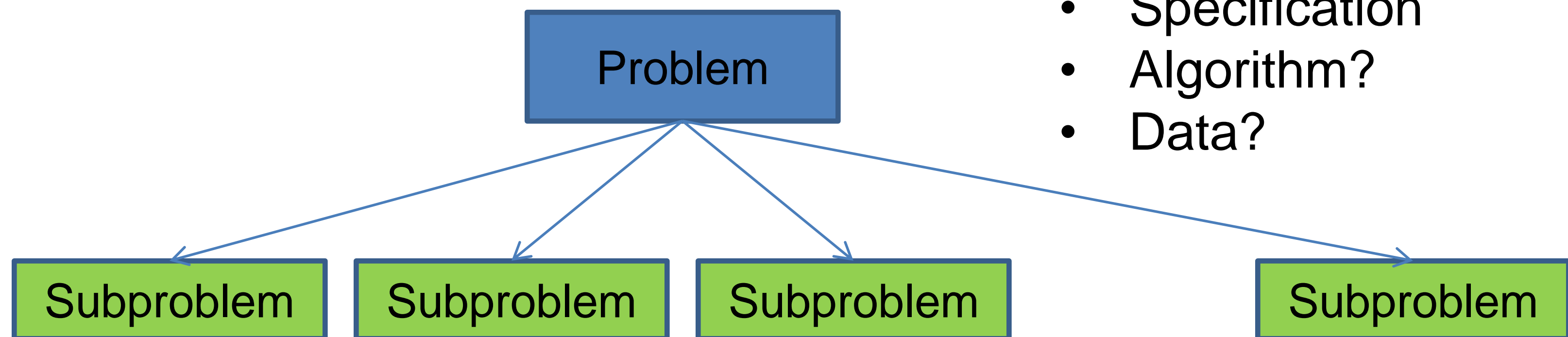


- Specification
- Algorithm?
- Data?

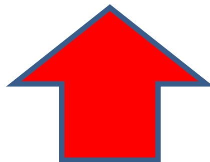
- How is the computation divided? Coordination necessary? Does the sequential algorithm help/suffice?
- Where are the data? Which communication is necessary?

The problem of parallelization

How to divide given problem into subproblems that can be solved in parallel?



- Specification
- Algorithm?
- Data?



Note:
This does not mean division once and for all time, subproblems are typically dynamic and diverse

Aspects of parallelization

- Algorithmic: Dividing computation into independent parts that can be executed in parallel. What kinds of shared resources are necessary? Which kinds of coordination? How can overheads be minimized (redundancy, coordination, synchronization)?

- Scheduling/Mapping: Assigning parts of computation to processors in good order

- Load balancing: (Re)assigning independent parts of computation to processors such that all resources are utilized to the same extent (evenly and efficiently)

- Communication: When must processors communicate? How?
- Synchronization: When must processors agree/wait for each other?

- Linguistic: How are the algorithmic aspects expressed? Concepts (programming model) and concrete expression (programming language, interface, library)
- Pragmatic/practical: How does the actual, parallel machine look? What is a reasonable, abstract ("bridging") model?
- Architectural: Which kinds of parallel machines can be realized? How do they look?

Levels of parallelism/which parallelism?

- Gates and functional units
- Instruction Level Parallelism (ILP)
- SIMD/Vector parallelism
- Cores/threads/processes
- Communication
- Synchronization
- Multiple applications
- Coupled applications

Levels of parallelism/which parallelism?

Hardware&Compiler, “Free lunch”

Implicit (processor) parallelism:

- Gates and functional units
- Instruction Level Parallelism (ILP)
- SIMD/Vector parallelism

- Cores/threads/processes
- Communication
- Synchronization

- Multiple applications
- Coupled applications

Levels of parallelism/which parallelism?

Implicit (processor) parallelism:

- Gates and functional units
- Instruction Level Parallelism (ILP)
- SIMD/Vector parallelism
- Cores/threads/processes
- Communication
- Synchronization

"Trivial" parallelization, distributed computing, …

Large-scale ("coarse-grained") parallelism:

- Multiple applications
- Coupled applications

Levels of parallelism/which parallelism?

Implicit (processor) parallelism:

- Gates and functional units
- Instruction Level Parallelism (ILP)

Parallel computing "parallelism"

Explicit ("fine-grained"): parallelism:

- SIMD/Vector parallelism
- Cores/threads/processes
- Communication
- Synchronization

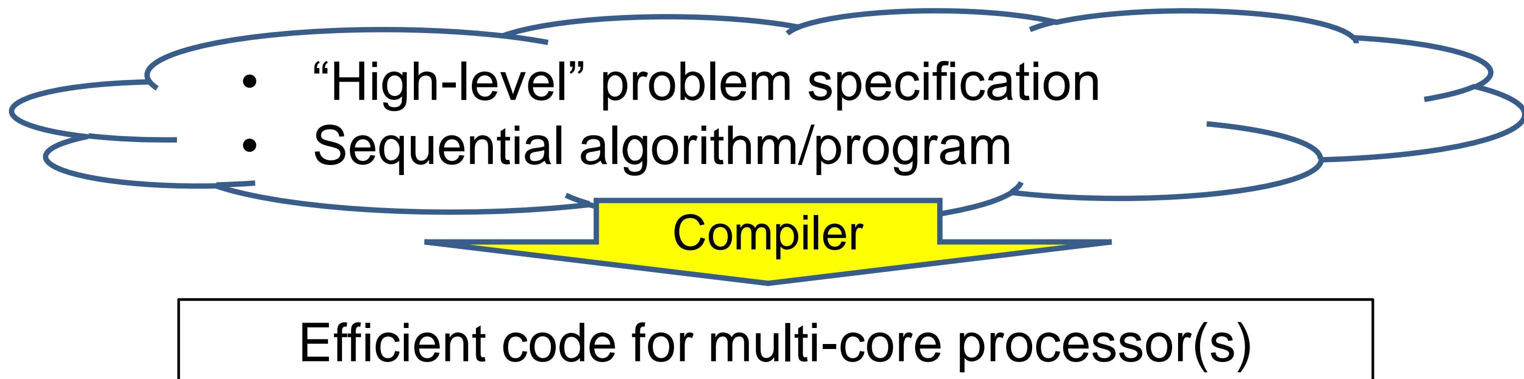
Large-scale ("coarse-grained") parallelism:

- Multiple applications
- Coupled applications

©Jesper Larsson Träff

Compiler parallelization, automatic parallelization

Why must parallelism be explicit? Can't we just leave it all to the compiler?



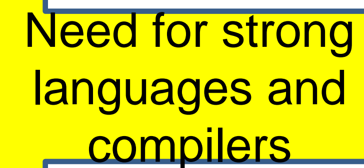
- Successful only to a limited extent:
- Compilers cannot invent a different algorithm
- Hardware parallelism (ILP) not likely to go much further

Samuel P. Midkiff: Automatic Parallelization: An Overview of Fundamental Compiler Techniques. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers 2012

Explicitly parallel programming (parallel computing today):

- Explicitly parallel code in some parallel language/interface
- Support from parallel libraries
- (Domain Specific Languages)

- Compiler does as much as compiler can do



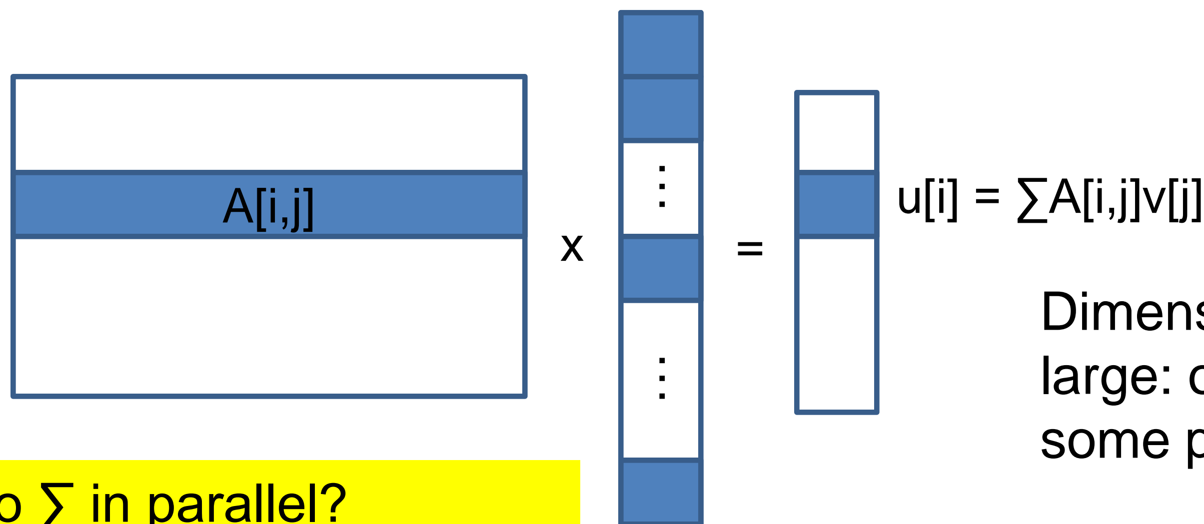
Lot's of interesting problems and tradeoffs, active area of research

- Algorithms
- Languages/interfaces
- Compilers

Some typical "given, individual problems" for this lecture

Problem 1:

Matrix-vector multiplication: Given (n x m) matrix A, m element vector v, compute n element vector u = Av

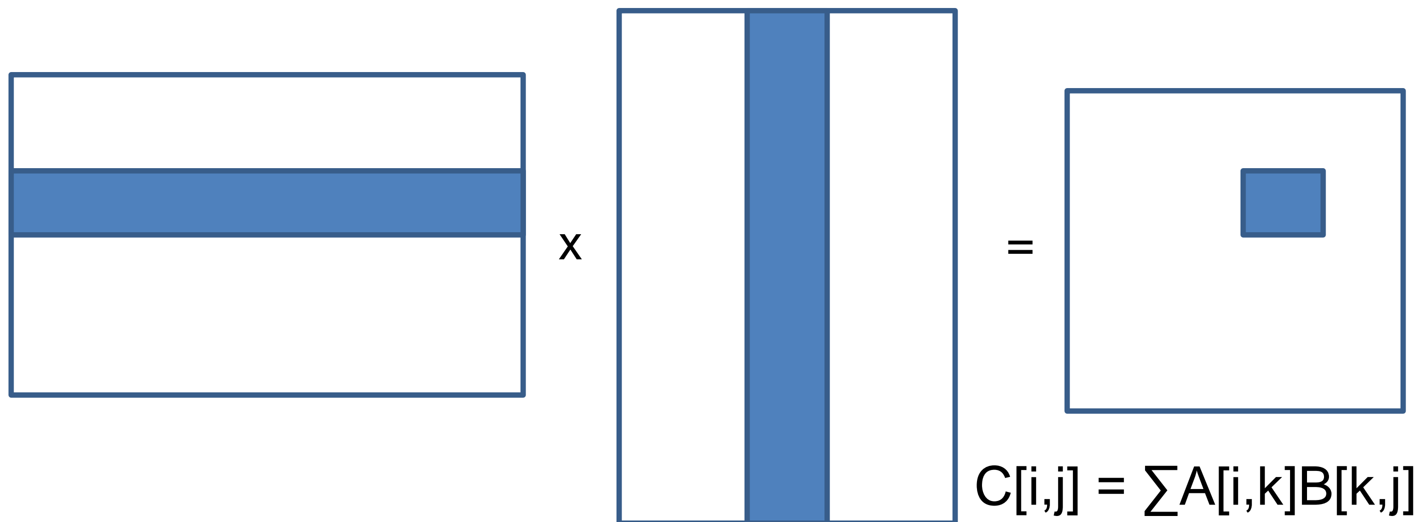


Dimensions n,m large: obviously some parallelism

How to ∑ in parallel?
Access to matrix and vector?

Problem 1a:

Matrix-matrix multiplication: Given (n x l) matrix A, (l x m) matrix B, compute (n x m) matrix product C = AB



©Jesper Larsson Träff

Problem 1b:

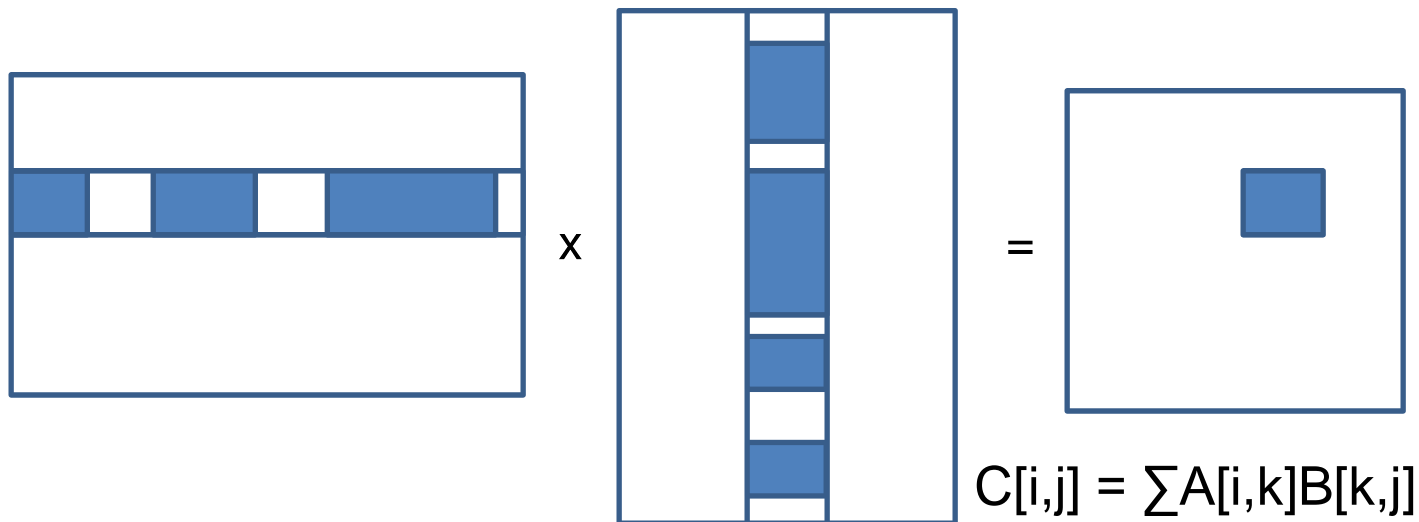
Solving sets of linear equations. Given matrix A and vector b, find x such that Ax = b

Preprocess A such that solution to Ax = b can easily be found for any b (LU factorization, …)

“sparse” means: lots of redundant information, e.g., 0-elements, 1-elements, …

Problem 1c:

Sparse matrix-matrix multiplication: Given nxk matrix A, kxm matrix B, compute (n x m) matrix product C = AB

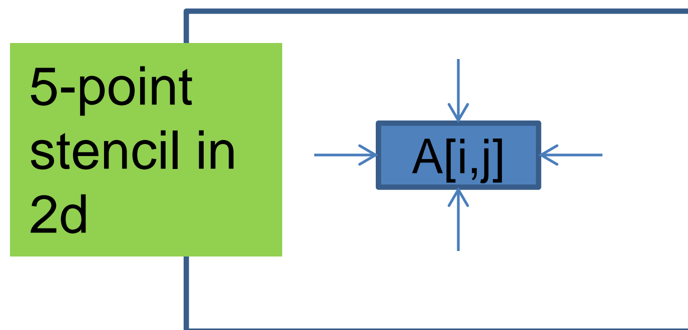


Use: Discretization and solution of certain parallel differential equations (PDEs); image processing; …

Problem 2:

Stencil computation, given nxm matrix A, update as follows, and iterate until some convergence criteria is fulfilled

with suitable handling of matrix border




```
iterate {
  for all (i,j) {
    A[i,j] <-
      (A[i-1,j]+A[i+1,j]+A[i,j-1]+A[i,j+1])/4
  }
} until (convergence)
```

Looks well-behaved, “embarrassingly parallel”? Data-matrix distribution? Conflicts on updates?

Problem 3:

Merge two sorted arrays of size n and m into a sorted array of size n+m

Easy to do sequentially, but sequential algorithm looks… sequential

Problem 3a:
Sort by merging

©Jesper Larsson Träff

Computing $\sum a_i$ for some associative operator + on some set of elements a_i in some order is called reduction

Problem 4:

Computation of all prefix-sums: Given an array A of size n of elements of some type S with an associative operations +, compute for all indices $0 \leq i < n$ the prefix-sums in array B

$$B[i] = \sum_{0 \leq j < i} A[j]$$

Implies solution to problem of computing just $\sum$. Parallel?

All prefix-sums

A: 1 2 3 4 5 6 7 …

B: x 1 3 6 10 15 21 28 …

Problem 5:

Sorting a sequence of objects (“reals”, integers, objects with order relation) stored in array

Hopefully parallel merge solution can be of help. Other approaches? Quicksort? Integers (counting, bucket, radix)?

Problem 6:

Graph search: Given G=(V,E), given start vertex s in V, compute a BFS/DFS traversal of G from s

0 GFLOPS, integer and bitwise operations matter

Graph analysis:

a) Given undirected G, find the connected components (CC)
b) Given directed G, find the strongly connected components (SCC)

All easy (well, almost, SCC) problems for sequential computing, with n=|V|, m=|E| solvable in O(n+m) steps. In parallel?

Are the undirected (CC) and the directed (SCC) problems any different?

Problem 6a:

Graph analytics:
Given G=(V,E), compute the between-ness centrality for all vertices of G (or other graph property…)

 ©Jesper Larsson Träff

Problem 6b:

Graph search: Given weighted G=(V,E), given start vertex s in V, compute a shortest path tree rooted at s (SSSP).

Compute shortest paths between all pairs of nodes in G (APSP).

Are the good, known SP algorithms (Dijkstra, Bellman-Ford, …) useful?

 ©Jesper Larsson Träff

Models in/of parallel computation

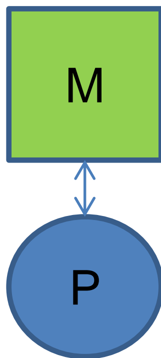
Computational/Architecture/Machine model (formally) describes components of processor (ALU, memory, interconnect), and defines how programs are executed and at what costs

Parallel machine model:

- Processor (core) capabilities
- Memory organization
- Communication and synchronization

- Execution and cost (: time)

Standard sequential model: The RAM (Random-Access Machine)

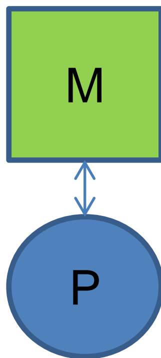


Processor (ALU, PC, registers) capable of executing instructions stored in memory on data in memory

Execution cost model, first assumption: All operations (ALU instructions, memory access) take same (unit) time.

Unit cost RAM model

Standard sequential model: The RAM (Random-Access Machine)



Realistic?

Useful?

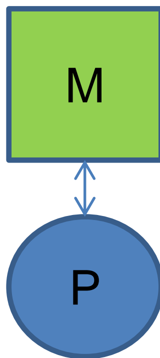
Instructions

- Memory: **load**, **store**, **move**
- Arithmetic (integer, real/floating point numbers)
- Logic: **and**, **or**, **xor**
- Bit/word: bitwise **and**, **or**, **xor**, **shift**
- Branch, compare, procedure call

Useful indeed, basis of much (all?) of sequential algorithmics

Another useful model of computation: The Turing machine

Standard sequential model: The RAM (Random-Access Machine)



Unit cost RAM model

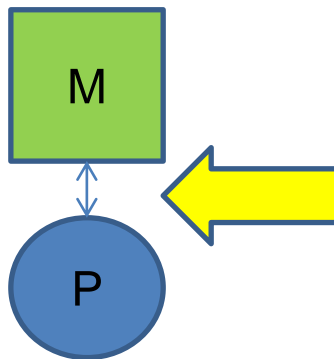
Execution cost model, first assumption: All operations (ALU instructions, memory access) take same (unit) time.

Not realistic: Memory access time is (much) slower than instruction execution

Practical consequence: Application performance determined by

- Instruction time
- Memory access time, memory bandwidth

Standard sequential model: The RAM (Random-Access Machine)

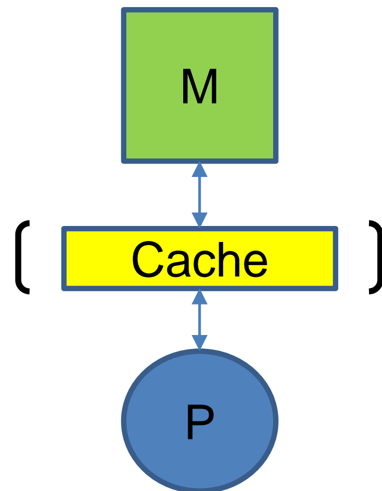


"von Neumann bottleneck": Program and data separate from CPU, performance bounded by memory bandwidth.

RAM aka "von Neumann architecture", or stored program computer, data and program in same memory

- John von Neumann (1903-57), Report on EDVAC, 1945; also Eckert&Mauchly, ENIAC
- John W. Backus: Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. Commun. ACM 21(8): 613-641 (1978)

Standard sequential model: The RAM (Random-Access Machine)



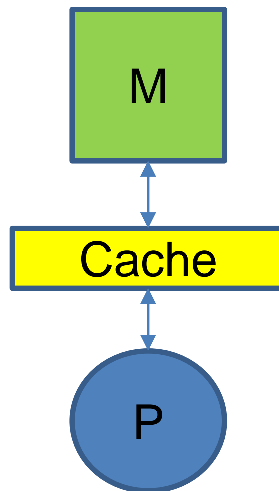
Processor (ALU, PC, registers) capable of executing instructions stored in memory on data in memory

Cache: Smaller memory with faster access time for storing frequently used data; managed by processor ("free lunch")

Clever architectural idea for hiding von Neumann bottleneck: Support unit cost RAM illusion by making data access look fast (unit cost)

Works well for algorithms with high temporal and spatial locality

Standard sequential model: The RAM (Random-Access Machine)



Processor (ALU, PC, registers) capable of executing instructions stored in memory on data in memory

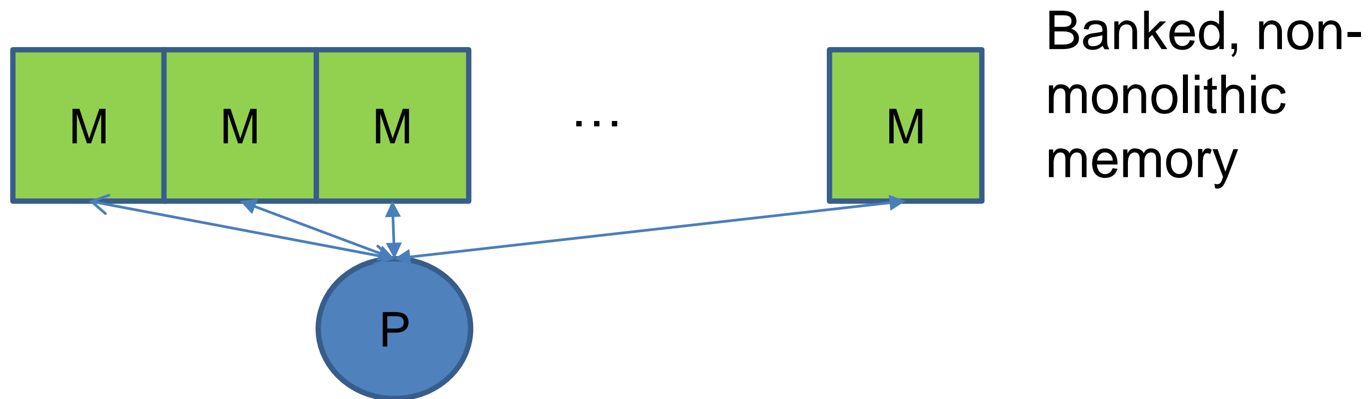
Cache: Smaller memory with faster access time for storing frequently used data; managed by processor ("free lunch")

More realistic?

Complex, non-uniform memory cost, hierarchy of different types of memory (caches)

More difficult for algorithm designer: External memory algorithms, cache-aware algorithms, cache-oblivious algorithms, …

(Parallel) RAM variations



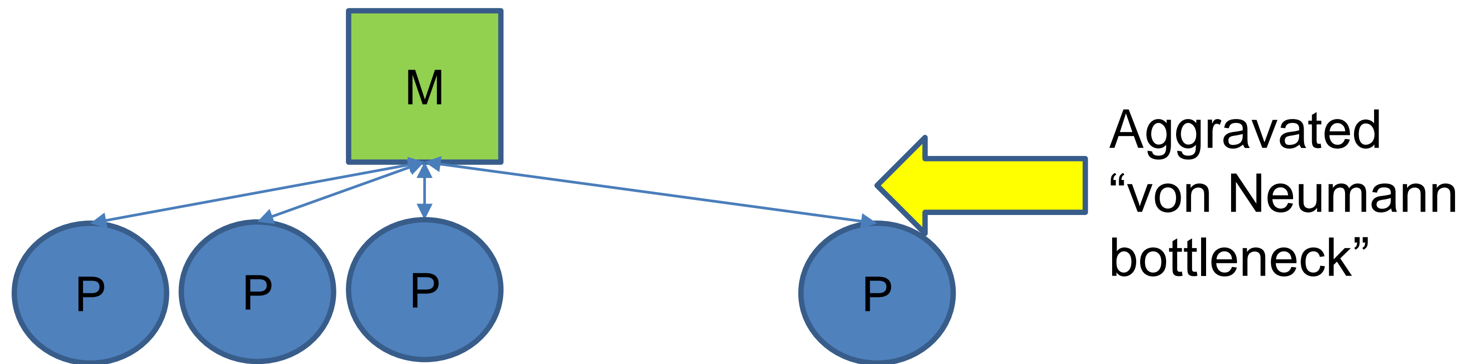
Banked, non-monolithic memory

Overcoming the von Neumann bottleneck:
Increasing memory bandwidth by more complex, banked memory (also: prefetching)

Processor, (parallel) instructions:

- Vector operations (arithmetic, logical) on multiple or larger words, single instruction operates on multiple data (SIMD)

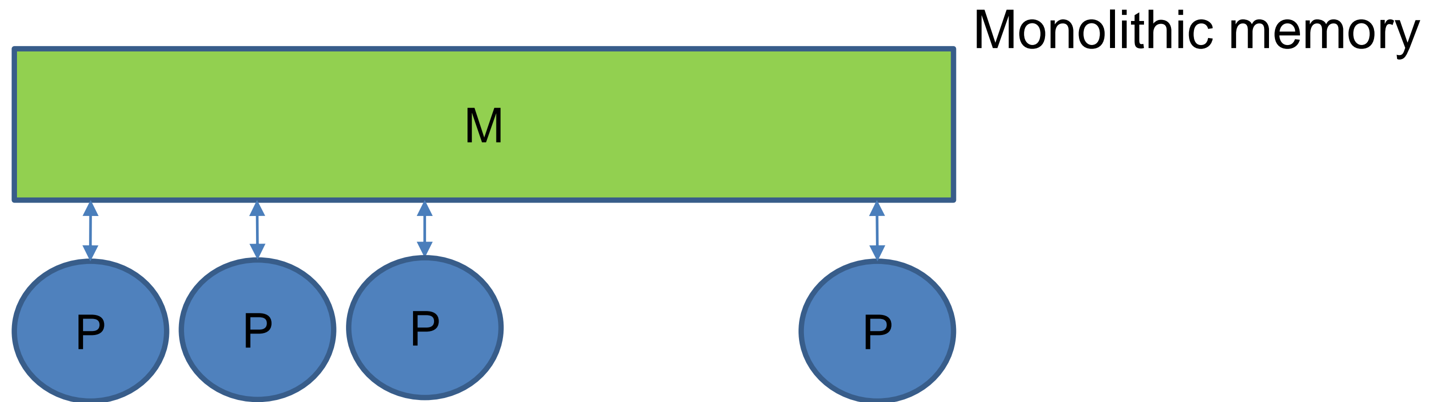
Parallel RAM variations



Shared-memory model (bus based). Parallelism through many processors, communication/coordination through shared memory

- How many processors?

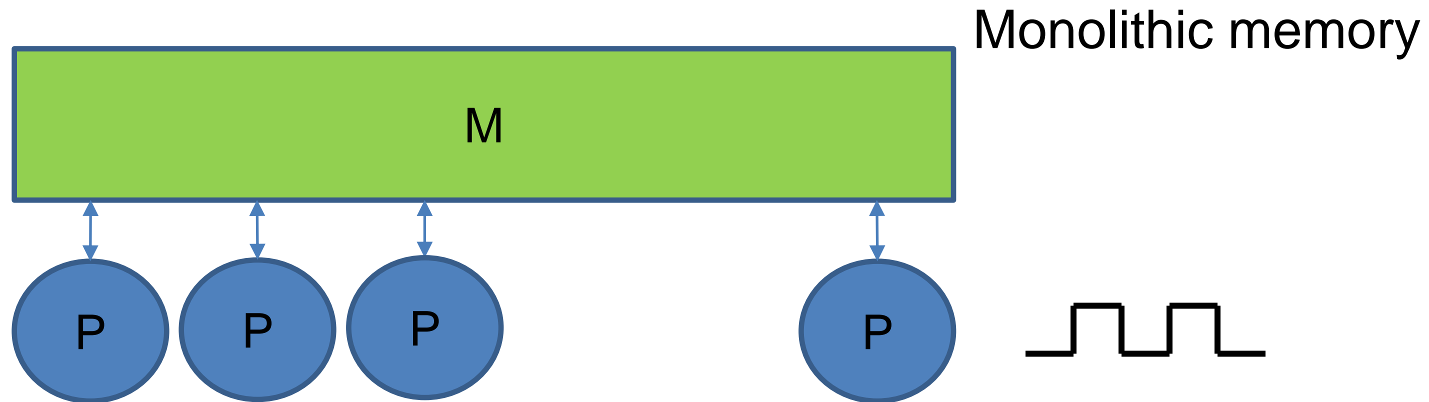
Parallel RAM variations



Shared-memory model (multi-ported, memory network)

- What can the memory do?
- How are the processors synchronized?
- What are the costs?

Parallel RAM variations



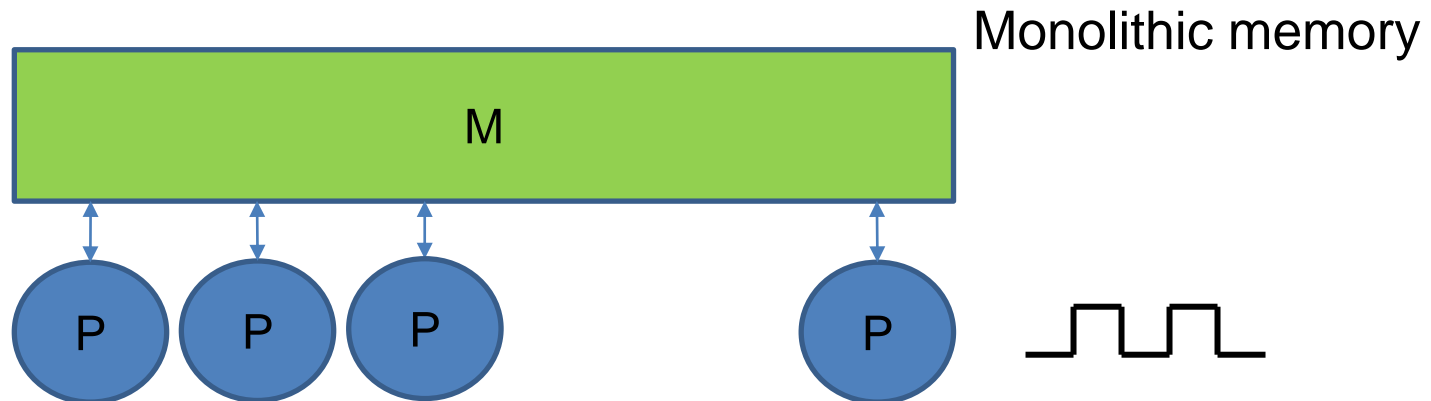
All processors operate under control of the same clock:
Synchronous, shared-memory model

PRAM a unit cost model

<u>Parallel RAM (PRAM)</u>:

- Processors work in lock-step (all same program, or individual programs), all perform an instruction in each step (clock tick)
- Unit-time instruction and memory access (uniform)

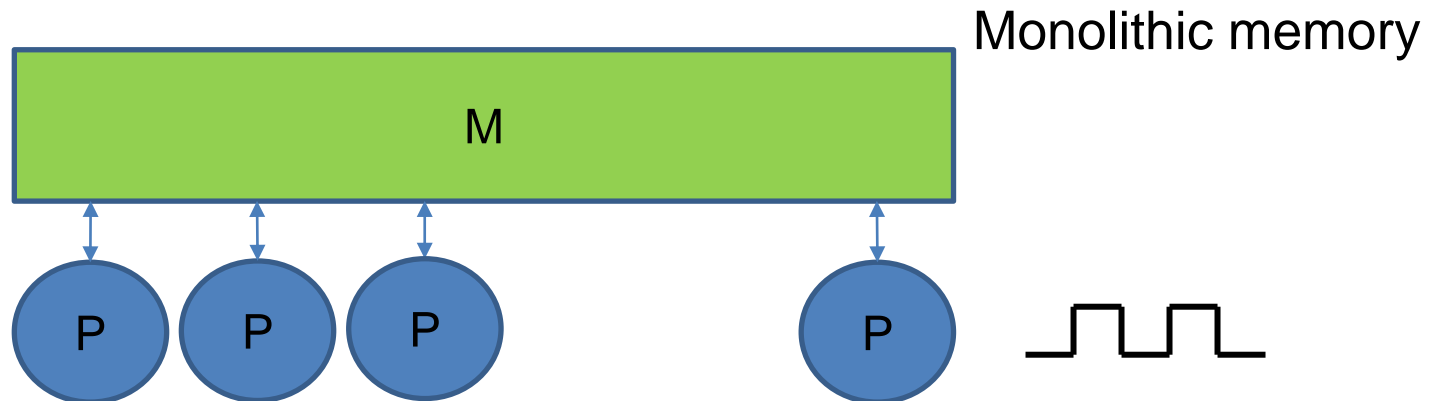
Parallel RAM variations



PRAM conflict resolution: What happens if several processors in the same time step access the same memory location?

- A memory location is either read or written in a time step
- EREW PRAM: Not allowed, neither read nor write
- CREW PRAM: Concurrent reads allowed, concurrent writes not
- CRCW PRAM: Both concurrent read and write

Parallel RAM variations



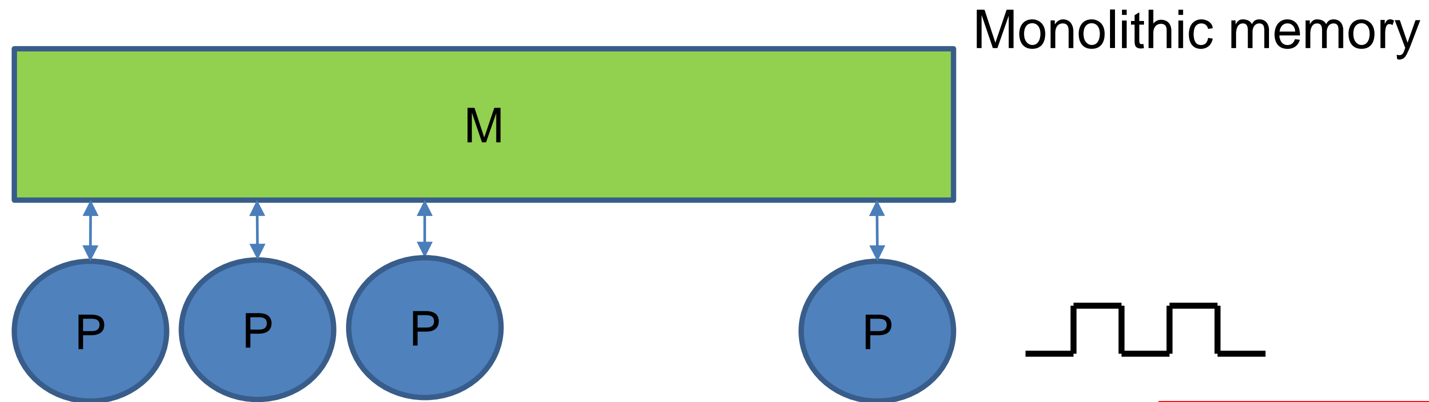
Realistic?

Useful?

CRCW PRAM write conflict resolution:

- COMMON: Conflicting processors must write same value
- ARBITRARY: One write succeeds
- PRIORITY: A priority scheme determines which

Parallel RAM variations



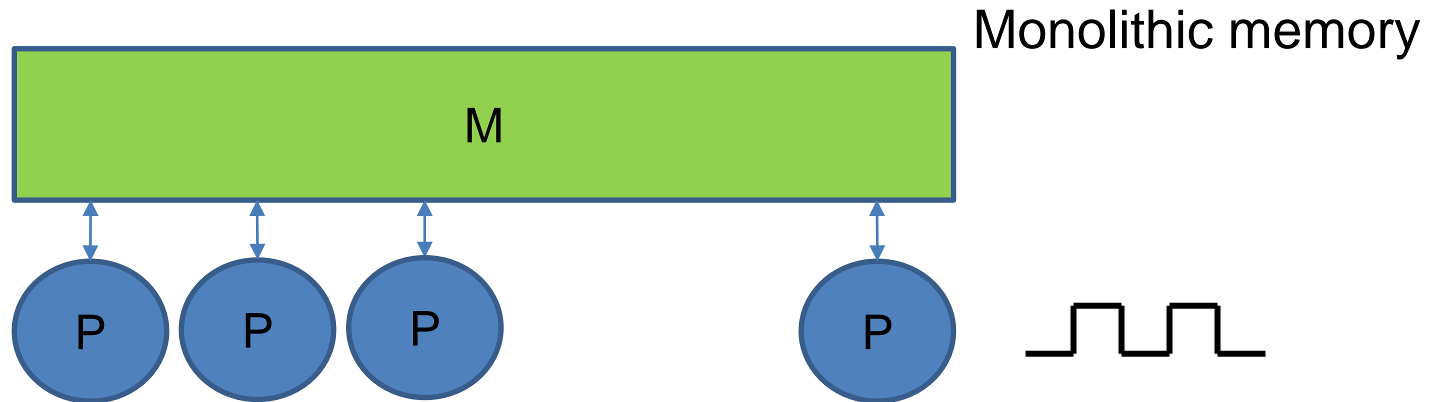
Realistic?

Useful?

Not realistic! Unit-time, uniform memory access, synchronous control, … all problematic assumptions
But extremely useful theoretical tool (late 70ties-mid 90ties): Fast algorithms, algorithmic techniques and ideas, and lower bounds

Joseph JáJá: An Introduction to Parallel Algorithms. Addison-Wesley 1992, ISBN 0-201-54856-9

Parallel RAM variations



PRAM algorithm design:

- Assume as many processors per step as convenient, dependent or independent of input size n (dependent: p=f(n) processors)
- How fast can some given problem of size n be solved, how many parallel steps are needed?
- What is the total number of operations carried out? What is the (maximum) number of processors required (for a step)?

Examples, PRAM pseudo-code

Initializing an array a of n elements:

```
par (0<=i<n) a[i] = i*i;
```

"**par**" construct indicates that some operation is to be performed for each i in the specified range. If a processor is available for each such i, the operations can be done in O(1) parallel time steps. There are O(n) operations to be done in all, n being the size of the range

```
for (k=1; k<n; k<<=1) {
  par (0<=i<n) a[i] = i/k;
}
```

At most ceil($\log_2$ n) parallel steps, each in O(1), total number of operations O(n log n)

PRAM algorithm design:

- Assume as many processors per step as convenient, dependent or independent of input size n (dependent: p=f(n) processors)
- Alternatively: Assume fixed number of processors p for input of (variable size) n
- How fast can some given problem of size n be solved, how many parallel steps are needed?
- What is the total number of operations carried out? What is the (maximum) number of processors required (for a step)?
- Is the number of processors (resources) used realistic? How can we judge this?
- Which PRAM variant is needed (EREW "weaker" than CRCW)
- Are there problems for which no reasonable PRAM algorithm exist?

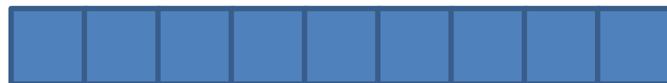
Yes, probably. But not in this lecture

PRAM Example: Fast maximum finding

Problem: Given n numbers in shared memory array a, find the maximum

Idea: Perform all n^2 comparisons (a[i] vs. a[j]) in parallel, eliminate those numbers that cannot be maximum. Use $p=n^2$ processors

… Input array a



… Elimination array b: `b[i]==false` means a[i] is not maximum

PRAM Example: Fast maximum finding

Problem: Given n numbers in shared memory array a, find the maximum

Idea: Perform all n^2 comparisons (a[i] vs. a[j]) in parallel, eliminate those numbers that cannot be maximum. Use $p=n^2$ processors

```
1. par (0<=i<n) b[i] = true;       // a[i] could be
2. par (0<=i<n, 0<=j<n)
     if (a[i]<a[j]) b[i] = false; // a[i] is not
3. par (0<=i<n) if (b[i]) x = a[i];
```

The algorithm consist of three parallel steps, with different number of processors in each step, **par**-construct allocates processors to array indices

Claim: `b[i]==true` iff a[i] is maximum among a[0], a[1], …,a[n-1]

```
1. par (0<=i<n) b[i] = true;       // a[i] could be
2. par (0<=i<n, 0<=j<n)
     if (a[i]<a[j]) b[i] = false; // a[i] is not
3. par (0<=i<n) if (b[i]) x = a[i];
```

Three parallel steps, in each the allocated processors perform a constant number of operations, O(1) time per step with n, n^2 and n PRAM processors, respectively.
CRCW capability needed in Steps 2 and 3

Theorem:
On a Common CRCW PRAM, the maximum of n numbers can be found in O(1) time steps and O(n^2) operations in total

Theorem:
On a Common CRCW PRAM, the maximum of n numbers can be found in $O(1)$ time steps and $O(n^2)$ operations in total

Observations:
- Constant time algorithm with polynomial resources (operations, processors)
- Total number of operations (number of allocated processors over all steps) is counted as the resource measure

Is this a good algorithm?

Answer later

…but it is fast

On constants in PRAM algorithms

If we agree on the cost (number of steps) of individual instructions, either at PRAM assembly level, or at the pseudo-code level, e.g.,

```
par (0<=i<n) b[i] = true;                      10+1
par (0<=i<n, 0<=j<n)
  if (a[i]<a[j]) b[i] = false;                 10+1+1
par (0<=i<n) if (b[i]) x = a[i];               10+1+1 = 35 steps
```

Say cost 10, independent of n, for simple processor assignment

we can analyze many simple PRAM programs and give exact running times (ignoring lower order terms, perhaps). We normally do not do so. But constants matter!

Another idea: PRAM maximum finding by pairwise comparisons

```
nn = n;
while (nn>1) {
  k = (nn>>1)+(nn&0x1); // bitwise ceil(nn/2)
  par (0<=i<k) {
    if (i+k<nn) a[i] = max(a[i],a[i+k]);
  }
  nn = k;
}
```

Another idea: PRAM maximum finding by pairwise comparisons

```
nn = n;
while (nn>1) {
  k = (nn>>1)+(nn&0x1); // bitwise ceil(nn/2)
  par (0<=i<k) {
    if (i+k<nn) a[i] = max(a[i],a[i+k]);
  }
  nn = k;
}
```

Another idea: PRAM maximum finding by pairwise comparisons

```
nn = n;
while (nn>1) {
  k = (nn>>1)+(nn&0x1); // bitwise ceil(nn/2)
  par (0<=i<k) {
    if (i+k<nn) a[i] = max(a[i],a[i+k]);
  }
  nn = k;
}
```

Another idea: PRAM maximum finding by pairwise comparisons

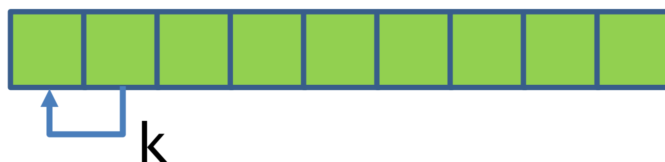
```
nn = n;
while (nn>1) {
   k = (nn>>1)+(nn&0x1); // bitwise ceil(nn/2)
   par (0<=i<k) {
      if (i+k<nn) a[i] = max(a[i],a[i+k]);
   }
   nn = k;
}
```

One step with ceil(nn/2) processors

CR

No Concurrent Read or Write

max(a[i],a[i+k])



··· Input array a

In each iteration of **while** loop, the number of elements that can be maximum is halved. Thus, **while** loop performs ceil($\log_2 n$) iterations

Parallel Computing
TU WIEN Informatics

Theorem:
On a CREW PRAM, the maximum of n numbers can be found in O(log n) time steps, and O(n) operations in total

Proof: The number of (sequential) operations in the while loop outside the parallel part is constant, O(1). The number of processors per iteration is k, over all iterations n/2+n/4+n/8+… ≤ n, plus at most $\log_2 n$ where k is odd, thus the number of operations in total is O(n). All processors read k and nn

In which respects is this a better algorithm than the fast maximum algorithm?

Answer later

Theorem: Can also be done on an EREW PRAM

Proof: Exercise… (think about this)

Wasting processors: PRAM maximum again

```
nn = n;
while (nn>1) {
  k = (nn>>1)+(nn&0x1); // bitwise ceil(nn/2)
  par (0<=i<n) {
    if (i+k<nn) a[i] = max(a[i],a[i+k]);
  }
  nn = k;
}
```

One step with n processors

Theorem (inferior):
On a CREW PRAM, the maximum of n numbers can be found in O(log n) time steps, and O(n log n) operations in total using n processors

Using given, fixed number of processors p: PRAM maximum again

```
nn = n;
while (nn>1) {
  k = (nn>>1)+(nn&0x1); // bitwise ceil(nn/2)
  par (i=0, (nn/p), 2*(nn/p), ..., <nn) {
    for (j=i, j<i+(nn/p); j++) {
      if (j+k<nn) a[i] = max(a[j],a[j+k]);
    }
  }
  nn = k;
}
```

nn/p steps with p processors

Theorem:

On a CREW PRAM, the maximum of n numbers can be found in O(n/p+log n) time steps, and O(n) operations in total using p processors

Parallel Computing TU WIEN Informatics

Matrix-matrix multiplication on a PRAM, easy version

```
par (0<=i<n) {
  par (0<=j<m) {
    C[i,j] = 0;
    for (k=0; k<l; k++) {
      C[i,j] += A[i,k]*B[k,j];
    }
  }
}
```

Nested parallelism

Not parallel

Possible to do better? In what respects?

Theorem:
On a CREW PRAM, matrix-matrix multiplication can be done in O(l) steps, and O(nml) operations in total using (nm) processors (assuming n,m,l known to all processors)

Matrix-matrix multiplication on a PRAM, easy version without nesting

```
par (0<=i<n, 0<=j<m) {
  C[i,j] = 0;
  for (k=0; k<l; k++) {
    C[i,j] += A[i,k]*B[k,j];
  }
}
```

Pseudo-code used liberally and judiciously: It must be possible in principle to execute the algorithm on the given PRAM

Theorem:
On a CREW PRAM, matrix-matrix multiplication can be done in O(l) steps, and O(nml) operations in total using n*m processors

Theorem: Can also be done on an EREW PRAM (think about this…)

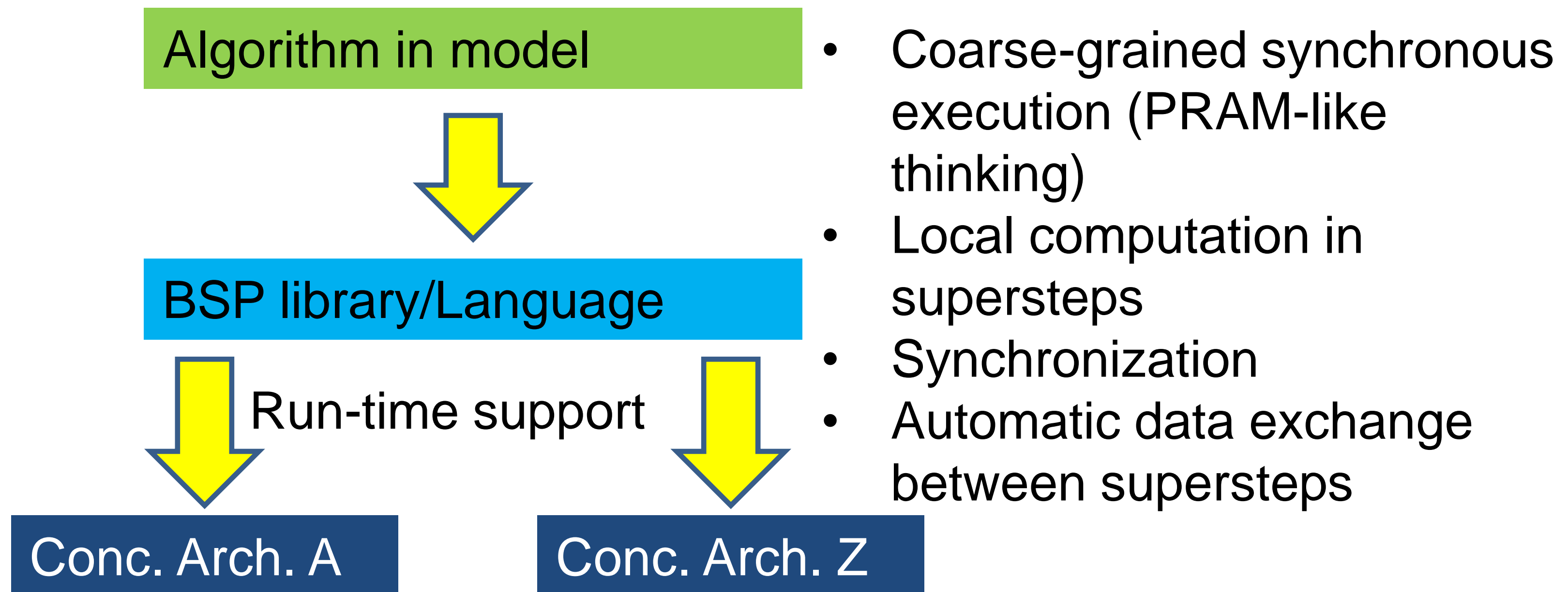
PRAM algorithms analyzed and judged on

- Number of parallel time steps needed (ultra-fast, fast, slow, …)
- Number of operations performed by the assigned processors over the time steps (work)
- Number of processors

What are the criteria for judging whether an algorithm is good or bad?

- Sequential base line, best-known algorithm (see later)
- Lower bounds

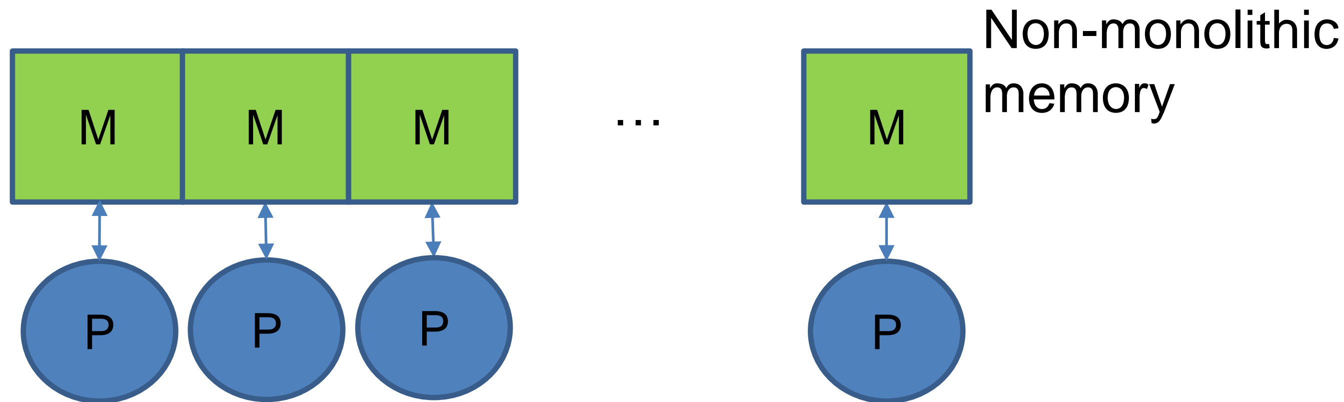
Valiant's bridging model proposal: Bulk Synchronous Parallel



- Coarse-grained synchronous execution (PRAM-like thinking)
- Local computation in supersteps
- Synchronization
- Automatic data exchange between supersteps

Leslie G. Valiant: A Bridging Model for Parallel Computation. Commun. ACM 33(8): 103-111 (1990)

Parallel RAM variations



Non-monolithic memory

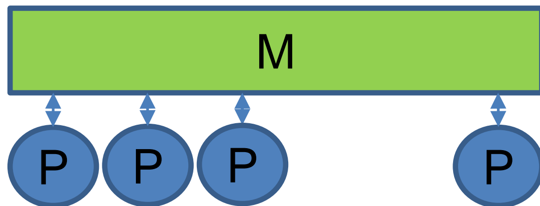
Shared-memory model, banked, memory-network, processors not synchronized (asynchronous, not lock-step, no common clock)

- How to synchronize: Atomic operations, barriers
- Memory consistency: When can some processor "see" what some other processor has written into memory?

Complicated semantics

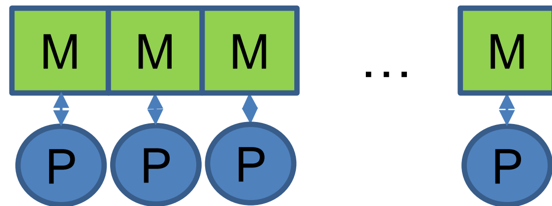
Parallel memory access cost terminology

UMA (Uniform Memory Access):
Access time to memory location independent of location and accessing processor, e.g., O(1), O(log M), …



Examples: RAM is UMA, PRAM is UMA (unit cost)

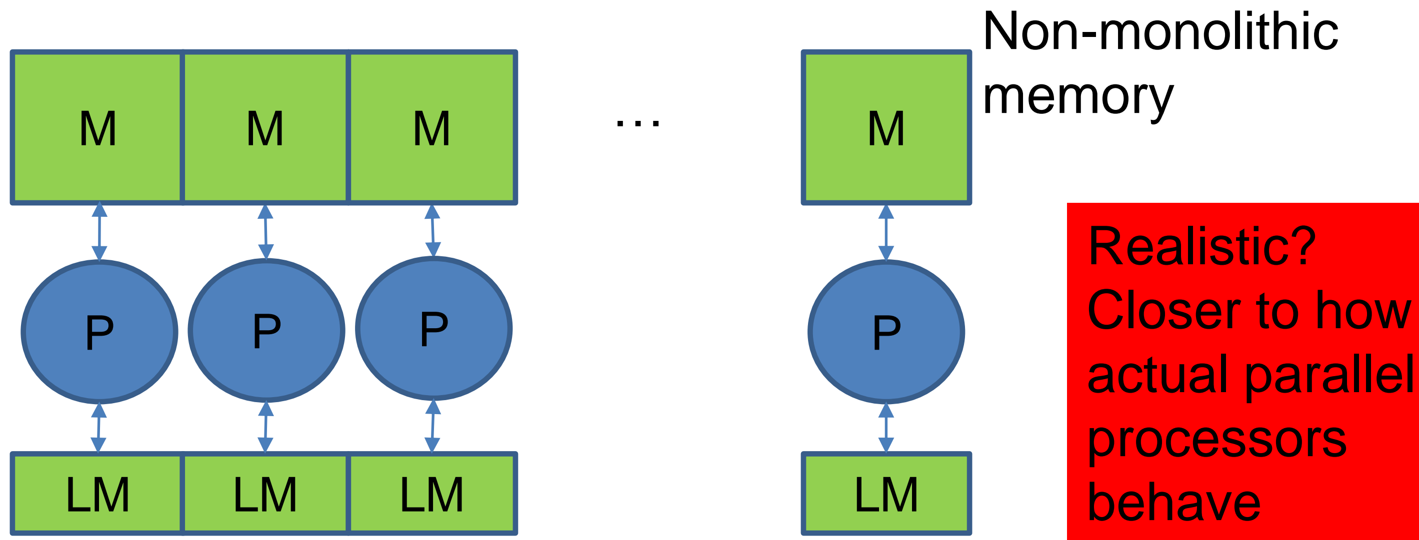
NUMA (Non-Uniform Memory Access):
Access time depends on processor and location.



(Almost) All "real" processors are NUMA

 ©Jesper Larsson Träff

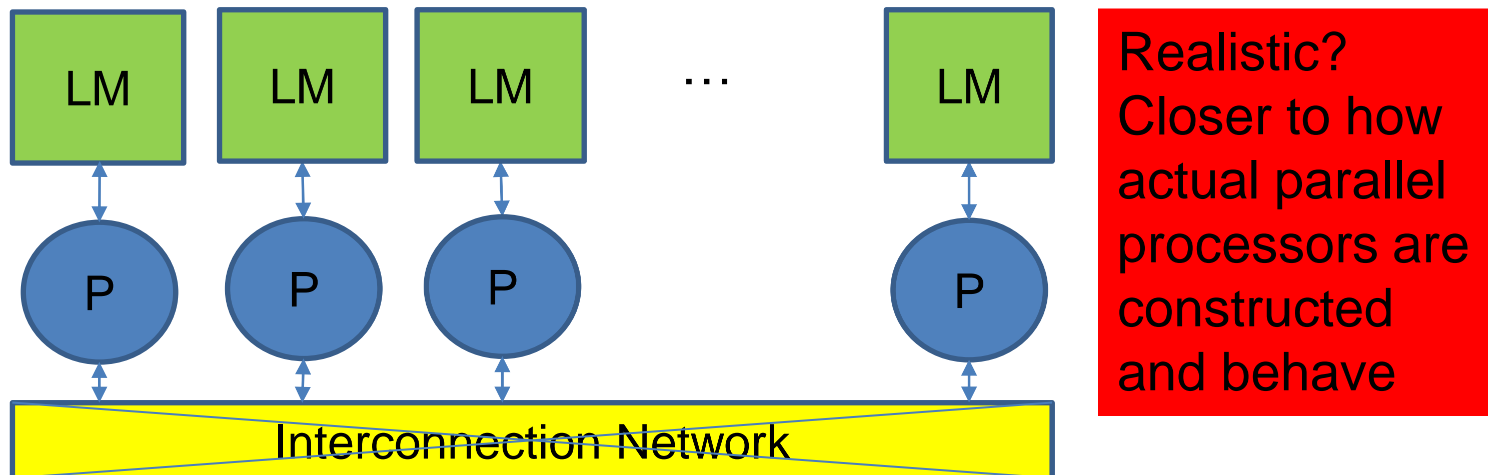
Parallel RAM variations



Locality: Some memory locations are closer (faster access) to processor than others; some memory may be entirely local, non-shared, only accessible by the processor to which it belongs

Challenge: Programming with locality

Parallel, distributed memory RAM



Memory is distributed over processors, memory is local to the processors, each processor can directly access only its own, local memory, communication through dedicated network

- Explicit communication needed

Parallel model summary

- Types/power of processors (instructions, functional units)
- Number of processors (fixed, bounded, unbounded, …)
- Memory organization (shared/distributed/both, cache-hierarchical), wordsize (fixed, bounded, unbounded)

- Communication (shared memory, network), operations
- Synchronization operations
- Memory behavior, atomic operations

Level of detail and formality depends on purpose:

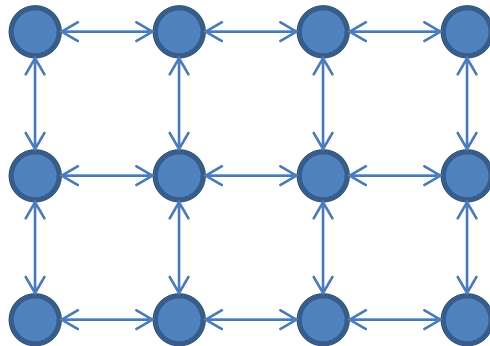
- Studying (limits to) parallelization, complexity theory
- Designing and analyzing algorithms
- Analyzing and predicting application performance

Execution cost model

- What is the cost (: time) of different types of operations (same unit cost, or dependent on type of operation, int/float)?
- What is the cost of memory access (UMA, NUMA)?
- Communication and synchronization costs? (latency and bandwidth)

Yet another parallel architecture model (totally non-RAM):

Cellular automaton, systolic array, … : Simple processors without memory (finite state automata, FSA), operate in lock step on (unbounded) grid, local communication only



State of cell (i,j) in next step determined by

- Own state
- State of neighbors in some neighborhood, e.g., (i,j-1), (i+1,j), (i,j+1), (i-1,j)

John on Neumann, Arthur W. Burks: Theory of self-reproducing automata, 1966
H. T. Kung: Why systolic architectures? IEEE Computer 15(1): 37-46, 1982
T. Toffoli, N. Margolus: Cellular Automata Machines: A new environment for modeling. MIT, 1987

Another architecture classification/taxonomy

- How many instructions can be carried out simultaneously?
- How much data (words) can be accessed simultaneously?

Flynn's taxonomy:
Instruction stream(s) and data stream(s) in computing system

M. J. Flynn: Some computer organizations and their effectiveness. IEEE Trans. Comp. C-21(9):948-960, 1972

	Instruction stream	
Data stream	SISD Single Instruction Single Data	MISD Multiple Instruction Single Data
	SIMD Single Instruction Multiple Data	MIMD Multiple Instruction Multiple Data

M. J. Flynn: Some computer organizations and their effectiveness. IEEE Trans. Comp. C-21(9):948-960, 1972

Flynn computing system organizations

- SISD: Single processor, single stream of instructions, operates on single stream of data. Example: Sequential architecture (e.g. RAM)

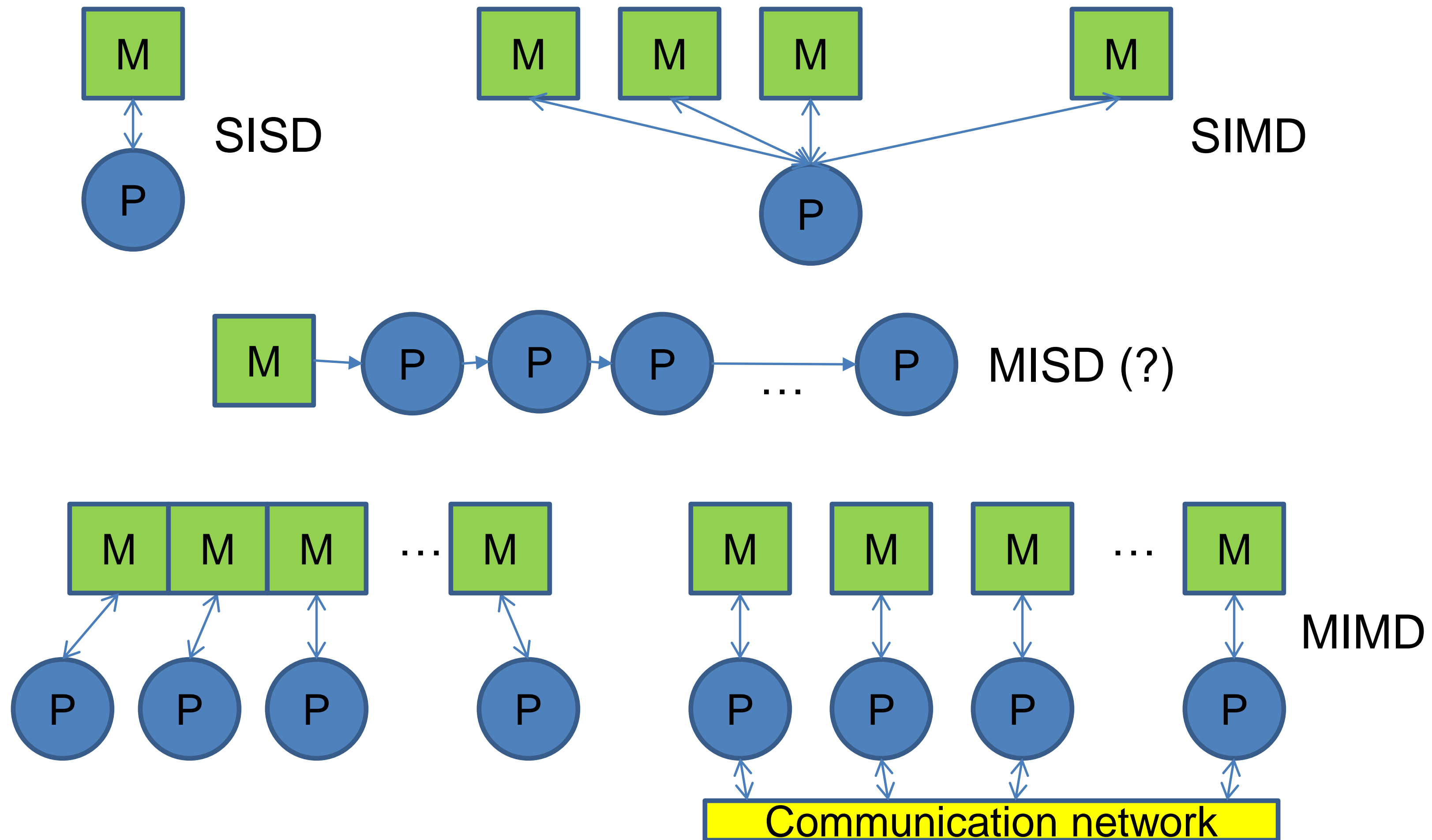
- SIMD: Single processor, single stream of operations, operates on multiple data per instruction. Example: traditional vector processor, SIMD-extensions, GPU(?) (PRAM, some variants)

- MISD: Multiple instructions operate on single data stream. Example: Pipelined architectures, streaming architectures(?), systolic arrays (70ties architectural idea)

Some say: Empty

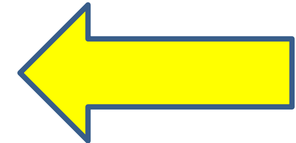
- MIMD: Multiple instruction streams, multiple data streams (PRAM, distributed memory architecture)

Typical Flynn taxonomy systems



Example: PRAM (also: GPU)

```
par (0<=i<n) {
  if (i+k<nn) {
    a[i] = max(a[i],a[i+k]);
  } else {
    … // something totally different
  }
}
```

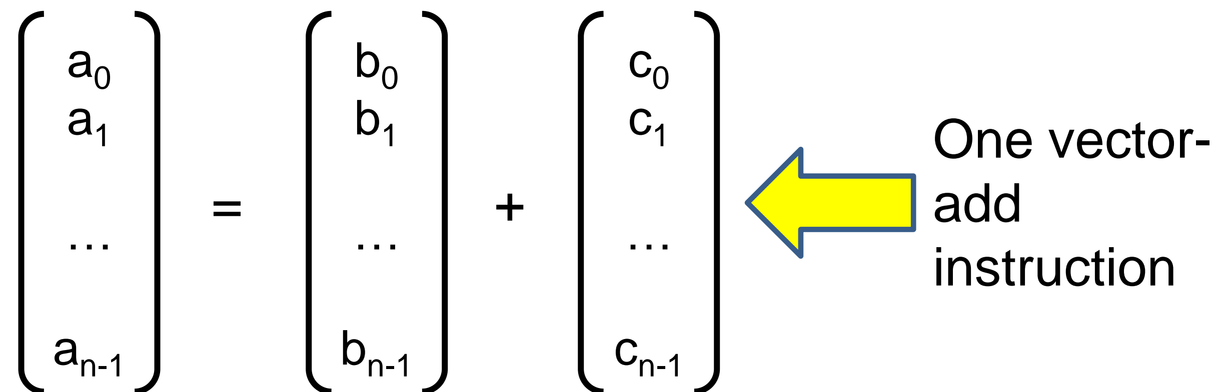


Processors do something different depending on the condition: MIMD, but under control of same program

SIMD restriction: Both branches are executed, for some processors as **noop**, depending on condition. On GPU's this is called branch/thread divergence, and can cause severe performance loss

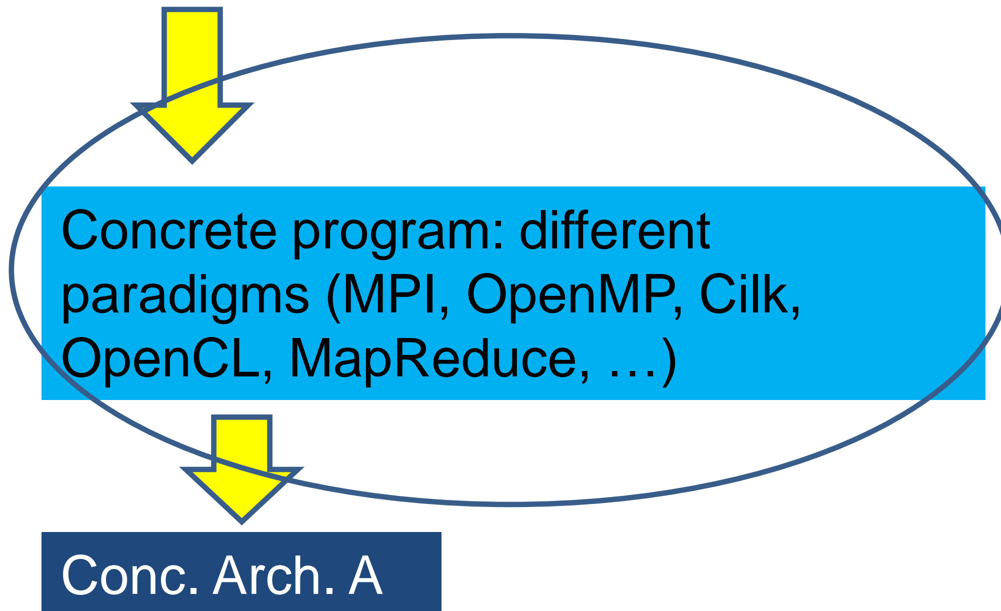
Example: Vector processor, the classical SIMD architecture

One instruction controls operation on a vector of data: Vector addition, …



- “Traditional” vector processor: support for arbitrarily long vectors
- Vector extensions: SSE, AVX, AVX512, smaller vectors of 4-8 double words

Programming models



Parallel programming model for parallel language or framework:

Mental model for programming and implementation of algorithms in given language

Parallel programming model:

Abstraction (close to programming language) defining

- parallel resources
- management of parallel resources
- parallelization paradigms
- memory structure, memory model
- synchronization and communication features

and their semantics and execution cost

Parallel programming language, or library (interface) is the concrete implementation of one (or more: multi-modal, hybrid) parallel programming model(s)

Some sequential programming models

- Imperative (C, …)
- Object-oriented, higher-order functional (C++, …)
- Functional (LISP, Haskell, …)
- Logical (Prolog, …)

Algorithm in model

Implementation

Concrete program (C, C++, Java, Haskell, Fortran,…)

Compilation

Concrete architecture

Challenge: Programming model that is useful, convenient, expressive, …, and close enough to concrete architecture to allow realistic performance analysis (prediction)

Challenge: How to support programming model efficiently on RAM-like architecture?

Parallel programming model defines

- Parallel resources: Processes, threads, tasks, …
- Expression of parallelism: Explicit or implicit
- Level and granularity of parallelism

- Memory model: Shared, distributed, hybrid
- Memory semantics ("when operations take effect/become visible")
- Data structures, data distributions

- Methods of synchronization (implicit/explicit)
- Methods and modes of communication

Examples:

1. Threads, shared memory, arrays, “parallel loops”, fork-join parallelism (OpenMP)
2. Processes, distributed memory, explicit message passing, collective communication, one-sided communication (MPI)
3. Tasks, spawn-join, dependencies, shared-memory (Cilk, OpenMP)
4. Shared arrays, implicit communication, “parallel-loops”, owner-computes (UPC, PGAS languages*)
5. Data parallel SIMD (CUDA, OpenCL)
6. …

Not this lecture

This lecture:
- OpenMP
- MPI
- (Cilk)

*PGAS: Partitioned Global Address Space

Flynn’s taxonomy as programming model description

Flynn’s taxonomy often used to characterize programming models (MIMD, SIMD)

- MIMD: Different threads/processes may execute different programs
- SIMD: One instruction flow operates on many data elements

Programming model classification (SPMD subcase of MIMD)

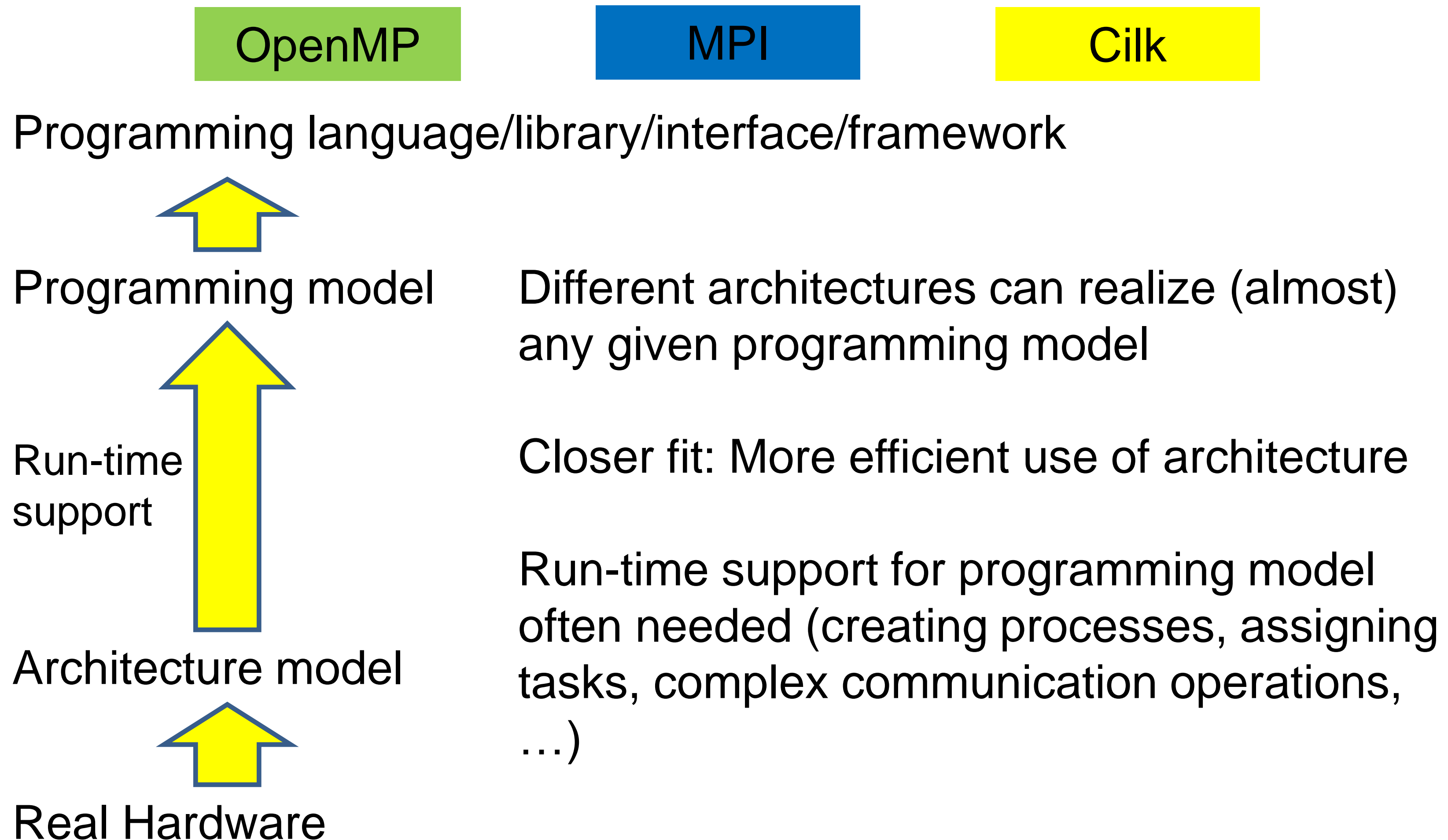
Sometimes useful and convenient to assume (restrict) that all processes execute the Same Program:

- Same objects (variables, procedures) exist for all processes, concepts like "remote procedure call", "active messages", "remote-memory access" make sense
- Processes may be executing different parts of the program at the same time

Programming model that makes this requirement is termed SPMD (Same Program Multiple Data)

All code in this lecture will be SPMD

F. Darema at al.: A single-program-multiple-data computational model for EPEX/FORTRAN, 1988



Different architectures can realize (almost) any given programming model

Closer fit: More efficient use of architecture

Run-time support for programming model often needed (creating processes, assigning tasks, complex communication operations, …)

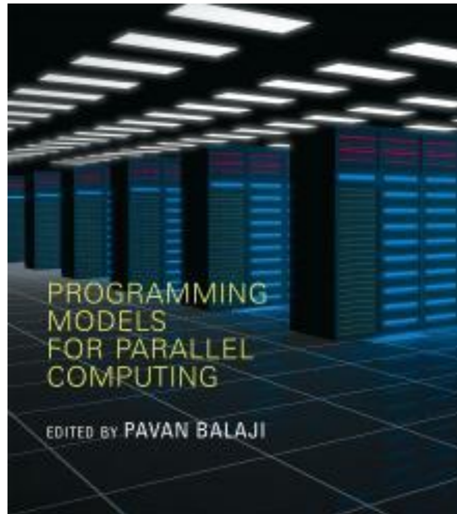
Examples:

- OpenMP programming interface/language for shared-memory model, intended for shared memory architectures. Can be implemented with DSM (Distributed Shared Memory) on distributed memory architectures; but performance usually not good

- MPI interface/library for distributed memory model, can be used on shared-memory architectures, too. Needs algorithmic support (e.g., “collective operations”)

- Cilk language (extended C) for shared-memory model, for shared-memory architectures; “task parallel”, needs run-time support (scheduling by “work-stealing”)

More examples of programming models and interfaces :



No attempt at defining what a programming model is, but an overview of current parallel/HPC interfaces and language extensions:

MPI, OpenMP, Cilk, OpenSHMEM, UPC, Chapel, Charm++, TBB, CUDA, and OpenCL

www.wikipedia.org is also not strong on definitions (in this area)

Lecture summary, checklist

- Parallel computing is everywhere.
- Moore’s “law”, “free lunch”

- Flynn’s taxonomy: MIMD, SIMD, SISD. SPMD restriction

- Models for parallel computation: Architecture, programming
- RAM, PRAM (EREW, CREW, CRCW), shared-memory, distributed memory, UMA, NUMA