Introduction: Hardware Trends and List Homomorphism

Cosmin E. Oancea and Troels Henriksen
[cosmin.oancea,athas]@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2015 PMPH Lecture Notes

Introduction^2

- Past > 20 years Information Revolution:
  Explosive growth of semiconductor integration + Internet.

- Moore's Low 1960s:
  - computing power doubles every 19-24 months
  - system cost effectiveness = `performance/cost` grows exp.
  - CMOS endpoint is near: miniaturization reaches its limits (complementary metal-oxide semiconductor).

Introduction^2

- Past $>$ 20 years Information Revolution:
  Explosive growth of semiconductor integration + Internet.
- Moore's Low 1960s:
  - computing power doubles every 19-24 months
  - system cost effectiveness = `performance/cost` grows exp.
  - CMOS endpoint is near: miniaturization reaches its limits (complementary metal-oxide semiconductor).
- Improved Chip Design:
  - each new process generation $\Rightarrow$ higher clock rates
  - logic switching speed & amount of on-chip logic increase ($\uparrow$)
  - better circuit design & deep pipelines $\Rightarrow$ fewer gate delays / stage
  - $\uparrow$ on-chip resources $\Rightarrow$ $\uparrow$ throughput by parallelism at all stages.
- Computer Architecture goes back to before 1970s:
  How to Best Utilize the ever-increasing ($\uparrow$) Wealth of Resources?

Brief History

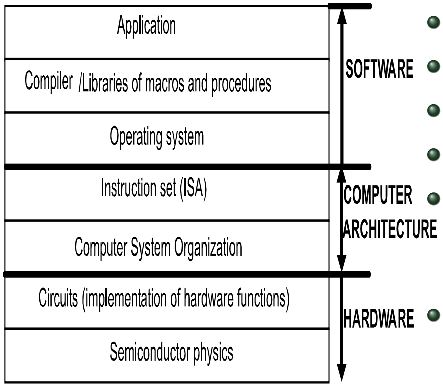
- ICPP, ISCA 1980/90s: parallel architectures popular topic.
  Demise of SingleCPU System: inevitable & fast approaching.

- Whatever happened? Mid90 Killer-Micro:
  - The rapid increase (↑) transistor density ⇒
  - path of least resistance: ever-increasing speed of SingleCPU
  - Complex Out-Of-Order (OoO) processors: 100s instructions/cycle.
  - Commercial arena: multiprocessors just an uniprocessor extension.

- What Changed? Multiprocessors Trend: Academia & Industry:
  - power complexity
  - Memory WALL: ↑ performance gap between processor & memory

- All Future Architectures adopt some form of massive parallelism!

What used to Be Computer Architecture?

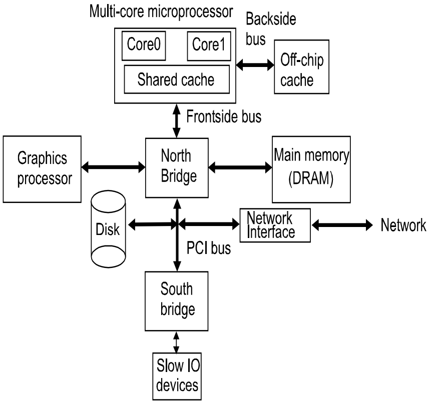

Application
Compiler /Libraries of macros and procedures
Operating system
Instruction set (ISA)
Computer System Organization
Circuits (implementation of hardware functions)
Semiconductor physics
SOFTWARE
COMPUTER ARCHITECTURE
HARDWARE

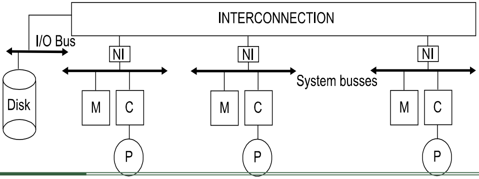
- Multi-Layered: focus competences.
- Each layer uses the ones below it.
- Application C++/Java/SML/Haskell.
- Compiler: machine code + OpSys calls.
- OpSys: extends hardware funct & orchestrates resource sharing among multiple users.
- ISA & Computer Organization: software-hardware interface.

Old Definition ISA – critical role in the success of computer industry:

- Early on, used to be the hallmark of system design ⇒
- Non-Portable Software & No Compiler at that time
- 1960s IBM System360 ISA guarantees backward compatibility ⇒
- Strategy endured test of time & Behemoth company today

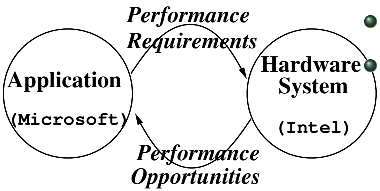
What Is Computer Architecture Today?





- Much Broader Definition: how best to build the computer (includes ISA, but focus changed on organization).
- North Bridge: sys bus connects cores to memory & IO devs.
- PCI bus: IO bus connecting North Bridge to high-speed IO to disk, network & slower dev
- $\leftarrow$ Generic Multiprocessor with Distributed Memory
- Parallel Sys Main Components: (1) Processor, (2) Memory Hierarchy and (3) Interconnection

Software-Hardware Synergy & Biggest Challenge



- Computer Architect Role:
- design trade-offs across HW/SW interf to meet functional & performance requirements within cost constraints.

- software: flexible approach to simplifying hardware, e.g., TLB exceptions, FP ops,
- hardware faster ⇒ world of tradeoff in between
- guided by the common case ⇒ trends are important.

- Important Juncture:
  - Higher performance requires Parallel Hardware
  - Biggest Challenge: develop efficient Massively-Parallel Software!

Course Organization

W	HARDWARE		SOFTWARE	LAB/CUDA
1	Trends		List HOM	Intro & Simple
	Vector Machine	$\longleftarrow$	(Map-Reduce)	Map Programming
2	In Order	$\longrightarrow$	VLIW Instr	Scan &
	Processor	$\longleftarrow$	Scheduling	Reduce
3	Cache		Reasoning About	Sparse Vect
	Coherence		Parallelism	Matrix Mult
4	Interconnection		Case Studies &	Transpose & Matrix
	Networks		Optimizations	Matrix Mult
5	Memory		Optimising	Sorting & Profiling &
	Consistency		Locality	Mem Optimizations
6	OoO, Spec		Thread-Level	Project
	Processor		Speculation	Work

Three narative threads: the path to complex & good design:

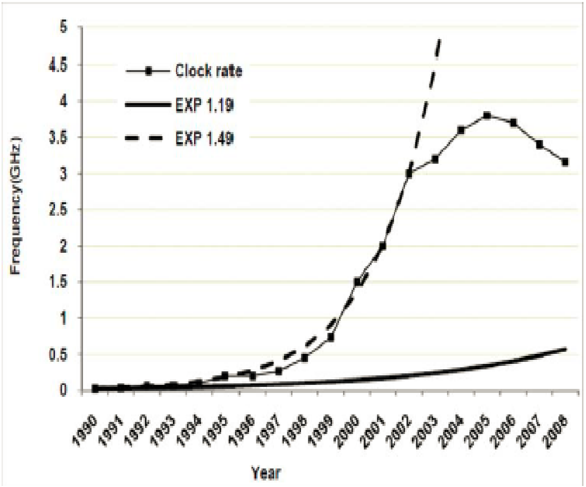
- Design Space tradeoffs, constraints, common case, trends.
- Reasoning: from simple to complex, Applying Concepts.

Abstractions

- A program is to a process/thread what a recipe is for cooking.
- Processor (core): hardware entity capable of sequencing & executing thread's instructions.
- MT Cores multiple threads, each running in its thread context.
- Multiprocessor: set of processors connected to execute a workload
  - mass produced, off-the-shelf, each several cores & levels of cache
  - trend towards migrating system functions on the chip: memory controllers, external cache directories, network interface

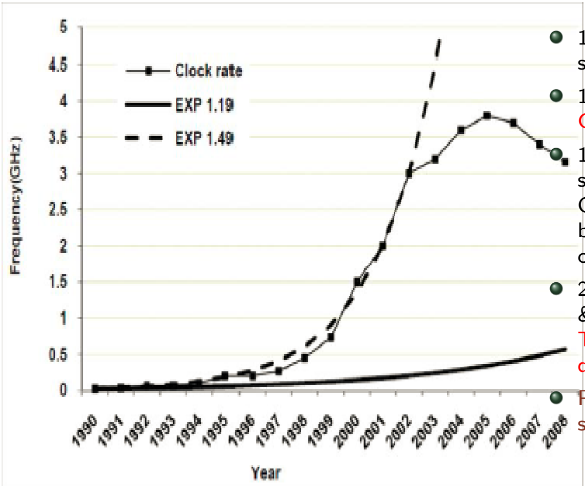
Processor: Clock Frequency/Rate

Historically the clock rate (at which instr are executed) has increased exponentially (1990-2004).



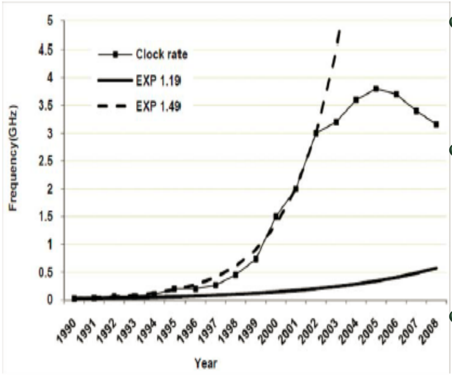
Processor: Clock Frequency/Rate

Historically the clock rate (at which instr are executed) has increased exponentially (1990-2004).



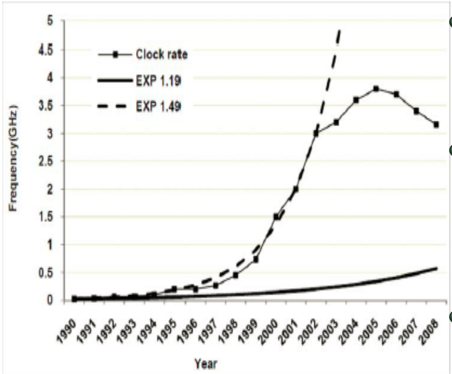
- 1.19× per-year due technology scaling (same hwd on new techn).
- 1990-2002: doubled every 21 months
  Curve 1.49×: 30GHz in 2008!
- 1.49 × −1.19×: very-deep (10-20 stages) pipelines! ILP via speculative OoO: register renaming, reorder buffs, branch prediction, lockup-free caches, memory disambiguation, etc.
- 2004: Intel cancels Pentium4 @4Ghz & Switched Track to Multi-Cores ⇒ Tectonic Shift away from muscled deeply-pipelined uniprocessor.
- Peaked in 2005, but mostly stalled since 2002!

Closer Look at Clock Rate



- Technology (process shrinkage): every generation transistors' switching speed increases 41%.
- Pipeline Depth: more stages ⇒ less complex ⇒ less gates/stage
  - # of gates delays dropped by 25% every process generation.
- Improved Circuit Design

Closer Look at Clock Rate

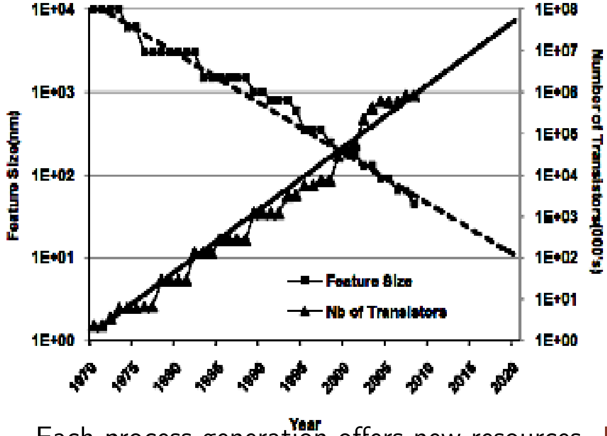


- Technology (process shrinkage): every generation transistors' switching speed increases 41%.
- Pipeline Depth: more stages $\Rightarrow$ less complex $\Rightarrow$ less gates/stage
  - # of gates delays dropped by 25% every process generation.
- Improved Circuit Design

Clock Rate Increase is Not Sustainable:

- Deeper pipelines: difficult to build useful stages with < 10 gates
- Wire delays: wire-transm speed $\uparrow$ much slower than switching,
- Circuits clocked at higher rates consume more power!

Processor: Feature Size & Number of Transistors



- new process generation every 2 years
- feature size reduced 30% every generation
- # of transistors doubles every 2 years (Moore's low). 1 Billion in 2008.

Each process generation offers new resources. How best to use the > 100 billion transistors? Large-Scale CMPs (100s-1000s cores):

- more cache, better memory-system design
- fetch and decode multiple instr per clock
- running multiple threads per core and on multiple cores

Memory Systems

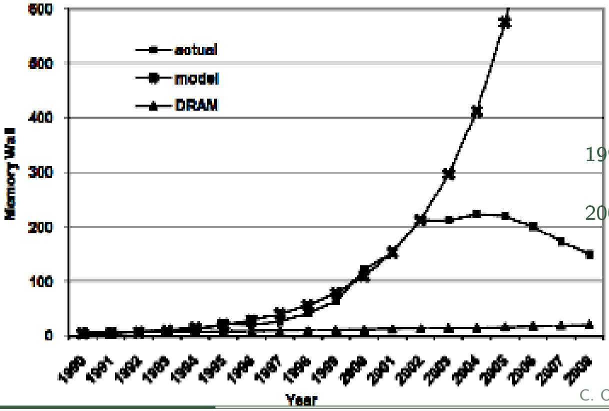
- (Main) Memory Wall: growing gap between processor and memory speed. Processor cannot execute faster than memory system can deliver data and instructions!

- Want Big, Fast & Cheap Memory System
    - access time increases with memory size as it is dominated by wire delays$\Rightarrow$ this will not change in future technologies
    - multi-level hierarchies (relies on principle of locality)
    - efficient management is KEY, e.g., cache coherence.
    - Cost and Size memories in a basic PC in 2008:

Memory	Size	Marginal Cost	Cost Per MB	Access Time
L2 Cache	1MB	\$20/MB	\$20	5nsec
Main Memory	1 GB	\$50/GB	5c	200 nsec
Disk	500GB	\$100/500GB	0.02c	5 msec

Memory Wall? Which Memory Wall??

- DRAM density increases 4× every 3 years, BUT
- DRAM speed ↑ only with 7% per year! (processor speed by 50%)
- Perception was that Memory Wall will last forever!
- Memory Wall Stopped Growing around 2002.
- Multi/Many-Cores ⇒ shifted from Latency to Bandwidth WALL



- `MemoryWall = mem_cycle/ proc_cycle`

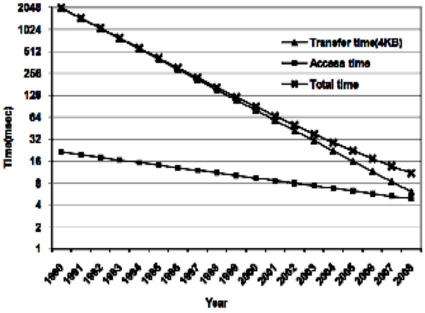
1990 `MemoryWall = 4` (25MHz,150ns)

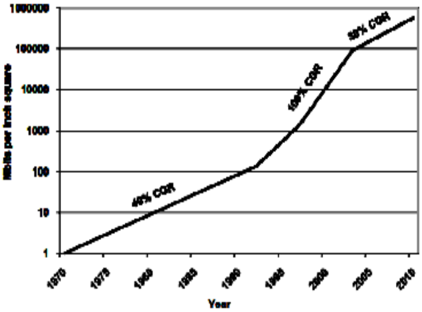
2002 exponential growth `MemoryWall = 200`

- Stalled since then.
- If trend continues: 1 Terabit Main Memory by 2021.

Disk Memory

- Historically disk performance improved by 40% per year
- `DiskTime=AccessTime+TransferTime` (`AccessTime=Seek+Latency`)
- Historically, transfer time have dominated, but
- Today: transfer and access time are of the same msecs order
- Future, Access Time will dominate, but proc-disk gap still large





Seek Time: head to reach right track, latency: time to reach the first record on track, both depend on rotation speed & independent on block size.

Interconnection Networks

Present at many layers:

- On-Chip Interconnects: forward values between pipeline stages, AND between execution units AND connect cores to shared cache banks.
- System Interconnects: connect processors (CMPs) to memory and IO
- I/O Interconnects, usually bus e.g., PCI, connect various devices to the System Bus
- Inter-Systems Interconnects: connect separate systems (chassis or boxes) & include
  - SANs: connect systems at very short distance
  - LANs, WANs (not interesting for us).
- Internet: global world-wide interconnect (not interesting for us).

Technological Contraints

- In the Past: tradeoff between cost (area) and time (performance).
- Today: design is challenged by several technological limits
  - Major new contraint is Power
  - wire delays
  - reliability
  - complexity of design
- It seems that parallelism addresses well all these constraints.

Power

- `Total Power = Dynamic + Static (Leakage)`
  $P_{dynamic} = \alpha C V^2 f$ consumed by a gate when it switches state
  $P_{static} = V I_{sub} \sim V e^{-k V_T / T}$ (caches)
- Dynamic power favors parallel processing over higher clock rate
  - $P_{dynamic} \sim F^3$ mostly dissipated in processor

Power

- `Total Power = Dynamic + Static (Leakage)`
  $P_{dynamic} = \alpha C V^2 f$ consumed by a gate when it switches state
  $P_{static} = V I_{sub} \sim V e^{-kV_T/T}$ (caches)
- Dynamic power favors parallel processing over higher clock rate
  - $P_{dynamic} \sim F^3$ mostly dissipated in processor
  - increase clock freq $4\times \Rightarrow$

Power

- `Total Power = Dynamic + Static (Leakage)`
  $P_{dynamic} = \alpha C V^2 f$ consumed by a gate when it switches state
  $P_{static} = V I_{sub} \sim V e^{-k V_T / T}$ (caches)

- Dynamic power favors parallel processing over higher clock rate
  - $P_{dynamic} \sim F^3$ mostly dissipated in processor
  - increase clock freq $4\times \Rightarrow 4\times$ speedup @ $64\times$ dynamic power!
  - replicate a uniprocessor $4\times \Rightarrow 4\times$ speedup @ $4\times$ power

- Static Power: dissipated in all circuits, at all time, nomatter of frequency and whether it switches or not.
  - negligible 15 years ago, but as feature size decreased so did the threshold voltage V_T every generation
  - Recently overtook dynamic power as major source of dissipation!

- Power/Energy are Critical Problems
  e.g., costly & many battery operated devices.

Reliability

- Transient Failures (Soft Errors):
  - Charge stored in a transistor `Q = C V`
  - Supply voltage V decreases every generation (consequence of features-size shrinking)
  - As Q decreases, it is easier to flip bits
  - Corruption Sources: cosmic rays, alpha particles radiating from the packaging material, electrical noise
  - Device operational but values have been corrupted
  - DRAM/SRAM error detection and correction capability
- Intermittent/Temporary Failures:
  - last longer, should try to continue execution
  - aging or temporary environmental variation, e.g., temperature
- Permanent Failures: device will never function again, must be isolated & replaced by spare
- Chip Mutiprocessors: promote better reliability
  - using threads for redundant execution,
  - faulty cores can be disabled $\Rightarrow$ natural failsafe degradation

Wire Delays

- Miniaturization $\Rightarrow$ transistors switch faster, but the propagation delay of signals on wire does not scale as well.
- Wire Delay Propagation $\sim RC$. $R \sim L/CS_{area}$. Miniaturization $\Rightarrow$ cross-section area keeps shrinking each generation, annuls the benefit of length shrinking.
- Wires can be pipelined like logic.
- Deeper pipelines are better because communication limited to only few stages.
- Impact of wire delays also favors multiprocessors, since communication traffic is hierarchical:
  - most communication is local
  - inter-core communication is occasional

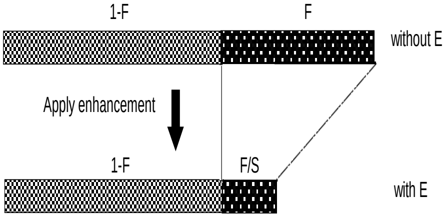
Design Complexity

- Design Verification has become the dominant cost of chip development today, major design constraint.
- Chip density increases much faster than the productivity of verification engineers (new tools and speed of systems):
  - register-transfer language level, i.e., logic is correct
  - core level, i.e., correctness of forwarding, memory disambiguation,
  - multi-core level, e.g., cache coherence, memory consistency.
- Vast majority of chip resources dedicated to storage also due to verification complexity:
  - trivial to increase the size of: caches, store buffers, load/store/ fetch queues, reorder buffers, directory for cache coherence, etc.
- Design Trend Favors Multiprocessors: easier to replicate the same structure multiple times than to design a large, complex one.

CMOS (Endpoint) Meets Quantum Physics

- CMOS is rapidly reaching the limits of miniaturization,
- Feature size: half pitch distance (half the distance between two metal wires). Gate length $\sim 1/2$ feature size.
- If present trends continues feature size $< 10nm$ by 2020
- Radius of atom: $0.1 \sim 0.2nm \Rightarrow$ gate length quickly reaches atomic distances that are governed by quantum physics, i.e., binary logics replaced with probabilistic states.
- Not clear what will follow (?)

Amdahl's Law



Enhancement accelerates a fraction F of the task by a factor S:

$$T_{exe}(withE) = T_{exe}(withoutE) \times [(1 - F) + \frac{F}{S}]$$

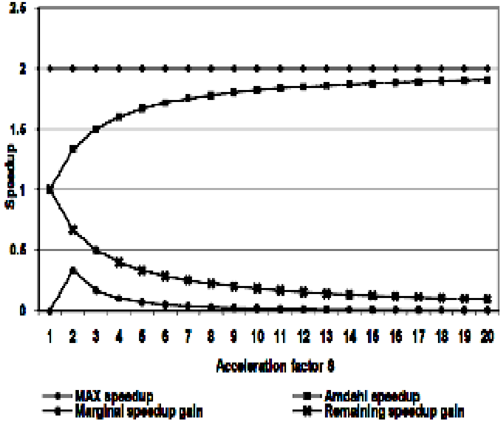
$$Speedup(E) = \frac{T_{exe}(withoutE)}{T_{exe}(withE)} = \frac{1}{(1-F)+\frac{F}{S}}$$

Amdahl's Law

1 Improvement is limited by the $1 - F$ part of the execution that cannot be optimized: $Speedup(E) < \frac{1}{1-F}$

2 Optimize the common case & execute the rare case in software.

3 Low of diminishing returns



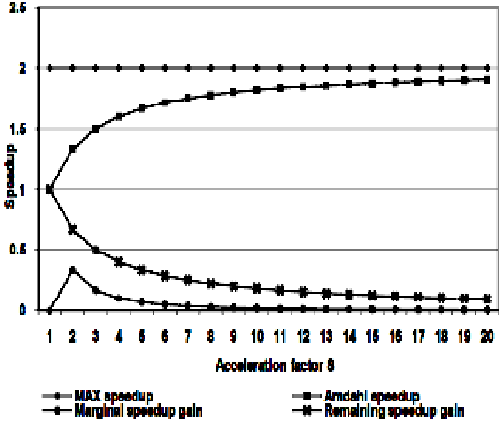
F=0.5

Amdahl's Law

1 Improvement is limited by the $1 - F$ part of the execution that cannot be optimized: $Speedup(E) < \frac{1}{1-F}$

2 Optimize the common case & execute the rare case in software.

3 Low of diminishing returns

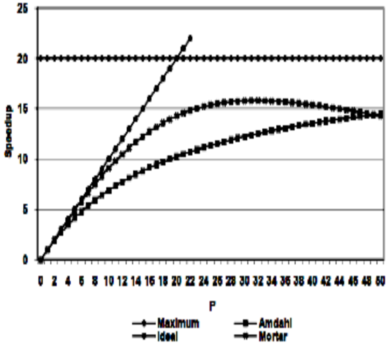


F=0.5

- every increment of S consumes new resources and is less rewarding:
- $S = 2 \Rightarrow 33\%$ speedup,
- $S = 5 \Rightarrow 6.67\%$ speedup.

Amdahl's Law: Parallel Speedup

$$S_P = \frac{T_1}{T_P} = \frac{P}{F+P(1-F)} < \frac{1}{1-F}$$

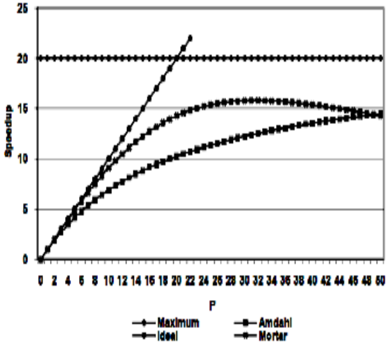


F=0.95

- Typically: speedup is sublinear, e.g., due to inter-thread communic.
- Sometimes superlinear speedup due to cache effects.
- Unforgiving Law: even if 99% is parallelized, $S_\infty < 100$.

Amdahl's Law: Parallel Speedup

$$S_P = \frac{T_1}{T_P} = \frac{P}{F+P(1-F)} < \frac{1}{1-F}$$



F=0.95

- Typically: speedup is sublinear, e.g., due to inter-thread communic.
- Sometimes superlinear speedup due to cache effects.
- Unforgiving Law: even if 99% is parallelized, $S_\infty < 100$.

Hardware Trend is to ever increase the number of cores.

Amdhal's Law: reason about parallelism asymptotically (∞ # cores), i.e., systematically exploit all levels of application's parallelism.

Math Preliminaries: Monoid & Homomorphism

Definition (Monoid)

Assume set S and $\odot : S \times S \rightarrow S$. $(S, \odot)$ is called a Monoid if it satisfies the following two axioms:
(1) Associativity: $\forall x, y, z \in S$ we have $(x \odot y) \odot z \equiv x \odot (y \odot z)$ and
(2) Identity Element: $\exists e \in S$ such that $\forall a \in S$, $e \odot a \equiv a \odot e \equiv a$.

($(S, \odot)$ is called a group if it also satisfies that any element is invertible, i.e., $\forall a, \exists a^{-1}$ such that $a \odot a^{-1} \equiv a^{-1} \odot a \equiv e$.)

E.g., $(\mathbb{N}, +)$, $(\mathbb{Z}, \times)$, $(\mathbb{L}_T, ++)$, where
$\mathbb{L}_T$ denotes lists of elements of type T, and $++$ list concatenation.

Math Preliminaries: Monoid & Homomorphism

Definition (Monoid)

Assume set S and $\odot : S \times S \to S$. $(S, \odot)$ is called a Monoid if it satisfies the following two axioms:
(1) Associativity: $\forall x, y, z \in S$ we have $(x \odot y) \odot z \equiv x \odot (y \odot z)$ and
(2) Identity Element: $\exists e \in S$ such that $\forall a \in S$, $e \odot a \equiv a \odot e \equiv a$.

($(S, \odot)$ is called a group if it also satisfies that any element is invertible, i.e., $\forall a, \exists a^{-1}$ such that $a \odot a^{-1} \equiv a^{-1} \odot a \equiv e$.)

E.g., $(\mathbb{N}, +)$, $(\mathbb{Z}, \times)$, $(\mathbb{L}_T, ++)$, where
$\mathbb{L}_T$ denotes lists of elements of type T, and $++$ list concatenation.

Definition (Monoid Homomorphism)

A monoid homomorphism from monoid $(S, \oplus)$ to monoid $(T, \odot)$ is a function $h : S \to T$ such that $\forall u, v \in S$, $h(u \oplus v) \equiv h(u) \odot h(v)$.

List Homomorphism (LH)

- Finite lists with concatenation, denoted ++, forms a Monoid:
  - `[1,2,3,4] ++ [5,6,7,8] = [1,2,3,4,5,6,7,8]`,
  - neutral elem e_{++} is the empty list `[]`, i.e., `[1,2]++[] = [1,2]`,
  - later we will look at them as vectors rather than linked lists.

List Homomorphism (LH)

- Finite lists with concatenation, denoted ++, forms a Monoid:
  - `[1,2,3,4] ++ [5,6,7,8] = [1,2,3,4,5,6,7,8]`,
  - neutral elem e_{++} is the empty list `[]`, i.e., `[1,2]++[] = [1,2]`,
  - later we will look at them as vectors rather than linked lists.

Definition (List Homomorphism)

$h :: [T_1] \rightarrow T_2$ *over finite lists is a* list homomorphism *if there exists a(n associative) binary operator* $\odot :: T_2 \rightarrow T_2 \rightarrow T_2$*, such that:*

$$h\ (x{+}{+}y) = (h\ x)\ \odot\ (h\ y)$$

We denote $h = hom\ (\odot)\ f\ \ e$*, where* $f(x) = h([x])$ *and* $e = h([])$*.*

- If the program is well behaved, i.e., same result for any `x++y` splitting of the input list, then $(T_2, \odot)$ is necessarily a Monoid.
- Similar to a divide and conquer algorithm, where `h(x)` and `h(y)` can be computed in parallel and results can be merged.

List Homomorphism (LH) Examples

Definition (List Homomorphism)

$h :: [T_1] \rightarrow T_2$ *over finite lists is a* list homomorphism *if there exists an associative binary operator* $\odot :: T_2 \rightarrow T_2 \rightarrow T_2$*, such that:*

$$h\ (x{+}{+}y) = (h\ x)\ \odot\ (h\ y)$$

We denote $h = hom\ (\odot)\ f\ e$*, where* $f\ x = h\ [x]$ *and* $e = h\ []$.

Examples of List Homomorphisms, id x = x is the identity function

```
-- len = hom (+) one 0, one x = 1
len :: [T] -> Integer
len []     = 0
len [x]    = 1
len (x++y) = (len x) + (len y)

-- flatten = hom (++) id []
flatten :: [[T]] -> [T]
flatten []     = []
flatten [x]    = x
flatten (x++y) = (flatten x) ++
                 (flatten y)
```

```
-- maxList = hom (max) id −∞
maxList :: [Int] -> Int
maxList []     = −∞
maxList [x]    = x
maxList (x++y) = (maxList x) ‘max‘
                 (maxList y)
-- Assume p :: T -> Bool given,
-- all_p = hom (&&) p True
all_p :: [T] -> Bool
all_p []     = ???
all_p [x]    = ???
all_p (x++y) = ???
```

Exercise: Implement the above LHs in Haskell

Implementation Sample for all_p:

```
import System.Environment -- access to arguments etc.
-- helper to simulate (x++y) pattern
split :: [a] -> ([a], [a])
split []  = ([],[] )
split [x] = ([],[x])
split xs  = let mid = (length xs) `div` 2  in  (take mid xs, drop mid xs)

-- all_p ≡ alln p = hom (&&) p True
alln :: (a -> Bool) -> [a] -> Bool
alln p []      = True
alln p [x]     = p x
alln p xs      = let (x, y) = split xs   in   (alln p x) && (alln p y)

----- Compile with: ghc -O2 -o test LHegHaskell.hs -----
main :: IO()
main = do args <- getArgs
          let inp = if null args then [0,2,4,6] else read (head args)
              p x = x `mod` 2 == 0
              res = alln p inp
          putStrLn ("Computed: " ++ show res)
```

Basic Blocks of Parallel Programming: Map

map :: $((\alpha \to \beta), [\alpha]) \to [\beta]$ has *inherently parallel semantics*.

$$\begin{array}{lllllll}
x = & \text{map}(f, \{ & a_1, & a_2, & .., & a_n & \}) \\
 & & \downarrow & \downarrow & & \downarrow & \\
x \equiv & \{ & f(a_1), & f(a_2), & .., & f(a_n) & \}
\end{array}$$

Map Fusion: (higher-order transformation)

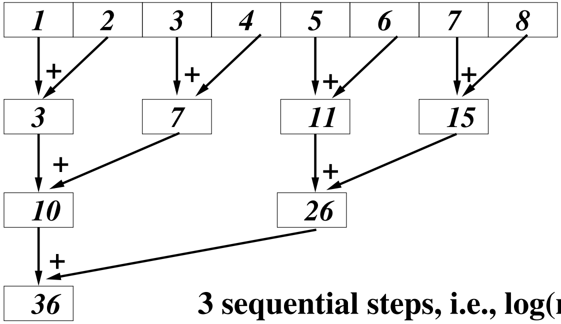
$$\begin{array}{lclcl}
a = & \{ a_1, a_2, .., a_n \} & & a = \{ a_1, a_2, .., a_n \} \\
x = & \text{map}(f, a) & & \\
y = & \text{map}(g, x) & \equiv & y = \text{map}(g \text{ o } f, a) \\
 & \downarrow & & \\
y \equiv & \{g(f(a_1)),g(f(a_2)),..,g(f(a_n))\} & \equiv & \{g(f(a_1)),g(f(a_2)),..,g(f(a_n))\}
\end{array}$$

Basic Blocks of Parallel Programming: Reduce

reduce :: $((\alpha \to \alpha \to \alpha), \alpha, [\alpha]) \to \alpha$

reduce($\odot$, e, $\{a_1, a_2, \ldots, a_n\}$) $\equiv e \odot a_1 \odot a_2 \odot \ldots \odot a_n$

where $\odot$ is an associative binary operator.



3 sequential steps, i.e., log(n)

Build programs by combining map, reduce and other such operators.

Map, Reduce, and Scan Types and Semantics

- $\texttt{map :: } (\alpha \to \beta) \to [\alpha] \to [\beta]$
  $\texttt{map f } [x_1,\ldots,x_n] = [f(x_1),\ldots, f(x_n)]$,
  i.e., $x_i :: \alpha, \forall i$, and $f :: \alpha \to \beta$.

- $\texttt{reduce :: } (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$
  $\texttt{reduce } \odot \texttt{ e } [x_1, x_2, .., x_n] = e \odot x_1 \odot x_2 \odot \ldots \odot x_n$,
  i.e., $e :: \alpha,\ x_i :: \alpha, \forall i$, and $\odot :: \alpha \to \alpha \to \alpha$.

- $\texttt{scan}^{exc} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$
  $\texttt{scan}^{exc}\ \odot \texttt{ e } [x_1, \ldots, x_n] = [e, e \odot x_1, \ldots, e \odot x_1 \odot \ldots x_{n-1}]$
  i.e., $e :: \alpha,\ x_i :: \alpha, \forall i$, and $\odot :: \alpha \to \alpha \to \alpha$.

- $\texttt{scan}^{inc} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$
  $\texttt{scan}^{inc}\ \odot \texttt{ e } [x_1, \ldots, x_n] = [e \odot x_1, \ldots, e \odot x_1 \odot \ldots x_n]$
  i.e., $e :: \alpha,\ x_i :: \alpha, \forall i$, and $\odot :: \alpha \to \alpha \to \alpha$.

1st List-Homomorphism Theorem [Meertens]

Theorem (1st List Homomorphism Theorem (Meertens))

Any homomorphism $h = hom\ (\odot)\ f\ e$ can be written as the functional composition of a reduce and a map:

$$h = hom\ (\odot)\ f\ e_\odot \ = \ (reduce\ (\odot)\ e_\odot)\ .\ (map\ f)$$

Conversely, each such composition is a homomorphism.

Apply Theorem to Re-Write LH to Map-Reduce!

```
-- len = hom (+) one 0, one x = 1
len :: [T] -> Integer
len []      = 0
len [x]     = 1
len (x++y) = (len x) + (len y)

-- flatten = hom (++) id []
flatten :: [[T]] -> [T]
flatten []      = []
flatten [x]     = x
flatten (x++y) = (flatten x) ++
                 (flatten y)
```

```
-- maxList = hom (max) id −∞
maxList :: [Int] -> Int
maxList []      = −∞
maxList [x]     = x
maxList (x++y) = (maxList x) ‘max‘
                 (maxList y)
-- Assume p :: T -> Bool given,
-- all_p = hom (&&) p True
all_p :: [T] -> Bool
all_p []      = ???
all_p [x]     = ???
all_p (x++y) = ???
```

1st List-Homomorphism Theorem [Meertens]

Theorem (1st List Homomorphism Theorem (Meertens))

Any homomorphism $h = hom\ (\odot)\ f\ e$ can be written as the functional composition of a reduce and a map:

$$h = hom\ (\odot)\ f\ e_{\odot}\ \ =\ \ (reduce\ (\odot)\ e_{\odot})\ .\ (map\ f)$$

Conversely, each such composition is a homomorphism.

Theorem tells how to parallelize LHs based on `map-reduce` skeletons.

Map-Reduce Definition for the Discussed LH Examples

```
-- len = hom (+) one 0, one x = 1
len = (reduce (+) 0) . (map one)

-- sum = hom (+) id 0, id x = x
sum = (reduce (+) 0) . (map id)

-- flatten = hom (++) id [],
flatten = (reduce (++) []) . (map id)
```

```
-- maxList = hom (max) id −∞
maxList = (reduce (max) (−∞)) .
          (map id)
```

$\texttt{-- all}_p \texttt{ = hom (\&\&) p True}$

$\texttt{all}_p \texttt{ = (reduce (\&\&) True) .}$
$\texttt{\ \ \ \ \ \ \ (map p)}$

List Homomorphism Invariants

Theorem (List-Homomorphism Promotions)

Given unary functions f, g and an associative binary operator $\odot$ then:

1. $(\texttt{map}\ f)\ .\ (\texttt{map}\ g)\ \equiv\ \texttt{map}\ (f\ .\ g)$
2. $(\texttt{map}\ f)\ .\ (\texttt{reduce}\ (++)\ [])\ \equiv\ (\texttt{reduce}\ (++)\ [])\ .\ (\texttt{map}\ (\texttt{map}\ f))$
3. $(\texttt{reduce}\ \odot\ e_\odot)\ .\ (\texttt{reduce}\ (++)\ [])\ \equiv$
   $(\texttt{reduce}\ \odot\ e_\odot)\ .\ (\texttt{map}\ (\texttt{reduce}\ \odot\ e_\odot))$

- 2. 3. $\Rightarrow$ code generation: list is segmented, segments are distributed on different processors, computation proceeds locally on each processor, and the local results are reduced.
- 2. 3. $\Leftarrow$ flattening optimization: uncovers more parallelism
- e.g., map f [1..4] $=$ (map f) . (red $++$) [[1,2],[3,4]] $=^{prom2}$?

Optimizing Map-Reduce Computation (Exercise)

Theorem (Optimized Map Reduce)

Assume $\texttt{distr}_p :: [\alpha] \rightarrow [[\alpha]]$ *distributes a list into p sublists, each containing about the same number of elements. Denoting* $\texttt{redomap}\ \odot\ f\ e_\odot \equiv (\texttt{reduce}\ \odot\ e_\odot)\ .\ (map\ f)$*, the equality holds:*

$$\texttt{redomap}\ \odot\ f\ e_\odot \equiv (\texttt{reduce}\ \odot\ e_\odot)\ .\ (\texttt{map}\ (\texttt{redomap}\ \odot\ f\ e_\odot))\ .\ \texttt{distr}_p$$

- Prove it using the promotion Lemmas before!
- Hint: $(\texttt{reduce}\ (++)\ [])\ .\ \texttt{distr}_p \equiv id$, hence
- $\texttt{redomap}\ \odot\ f\ e_\odot \equiv$
  $(\texttt{reduce}\ \odot\ e_\odot)\ .\ (map\ f)\ .\ (\texttt{reduce}\ (++)\ [])\ .\ \texttt{distr}_p$

Are All List Homomorphism Efficient?

If the combine operator involves concatenation then does map-reduce provides efficient parallelization?

Merge Sort

```
-- merge two sorted lists
merge :: Ord T => [T] -> [T] -> [T]
merge [] y  = y
merge x  [] = x
merge (x:xs) (y:ys) =
  if ( x <= y )
  then x : merge xs (y:ys)
  else y : merge (x:xs) ys
```

```
-- mSort = hom merge [.] []
-- [.] x = [x]
mSort :: Ord T => [T] -> [T]
mSort []     = []
mSort [x]    = [x]
mSort (x++y) = (mSort x) `merge`
               (mSort y)
```

In the naive merged sort, the `merge` reduction operator traverses sequentially the whole list, hence this map-reduce does not give efficient parallelization!

Conclusion

What have we talked about today?

- Hardware Parallelism:
  the only way of scalably increasing the compute power.
  - demonstrated by hardware trends:
  - power, reliability, wire delays, design complexity.
- Big Challenge: having parallel commodity software.
- List-Homomorphism:
  a way of reasoning about parallelism and
  of building inherently parallel programs.