PMPH Introduction: Course Organization, Hardware Trends, List Homomorphism

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2024 PMPH Lecture Slides

Intended Learning Outcomes

List Homomorphism: a way of writing inherently-parallel programs.

- explain when and where PMPH'24 lectures and lab sessions are located in space and time,
- explain Moore's law and argue based on hardware trends and technological constraint why all modern and future architectures (will) adopt some form of massive parallelism,
- explain what a list-homomorphic program is, and be able to apply it to build programs;
- illustrate and apply the 1^{st} Theorem of List Homomorphism to transform said programs into inherently parallel ones.

1. Brief History: Parallelism Paves the Path to Higher Performance
2. Course Organization
3. Hardware Trends of Critical Components of a Parallel System
   - Processor
   - Memory
   - Interconnect
4. Technological Challenges/Constraints
   - Power
   - Reliability
   - Wire Delays
   - Design Complexity
   - CMOS Endpoint
5. List Homomorphisms (LH)
   - List Homomorphism Definition and 1st Theorem
   - Almost Homomorphisms Gorlatch'96

Trend towards Ever-Increasing Hardware Parallelism

Moore's Law (1960s)

- "Number of transistors in a dense integrated circuit doubles approximately every two years."

Trend towards Ever-Increasing Hardware Parallelism

Moore's Law (1960s)

- "Number of transistors in a dense integrated circuit doubles approximately every two years."
- Rephrased as:
  - computing power doubles every 19-24 months, and
  - cost effectiveness (`performance/cost`) keeps pace.

Brief History

- ICPP, ISCA (1980/90s): parallel architectures popular topic.
- Whatever happened? Mid90 Killer-Micro:

Trend towards Ever-Increasing Hardware Parallelism

Moore's Law (1960s)

- "Number of transistors in a dense integrated circuit doubles approximately every two years."
- Rephrased as:
  - computing power doubles every 19-24 months, and
  - cost effectiveness (`performance/cost`) keeps pace.

Brief History

- ICPP, ISCA (1980/90s): parallel architectures popular topic.
- Whatever happened? Mid90 Killer-Micro:
  - path of least resistance: ever increasing the speed of Single CPU
  - Commercial arena: multiprocessors just an uniprocessor extension.
- What Changed? Multiprocessors Trend: Academia & Industry:
  - power complexity $P_{dynamic} \sim Freq^3$. Example!

Trend towards Ever-Increasing Hardware Parallelism

Moore's Law (1960s)

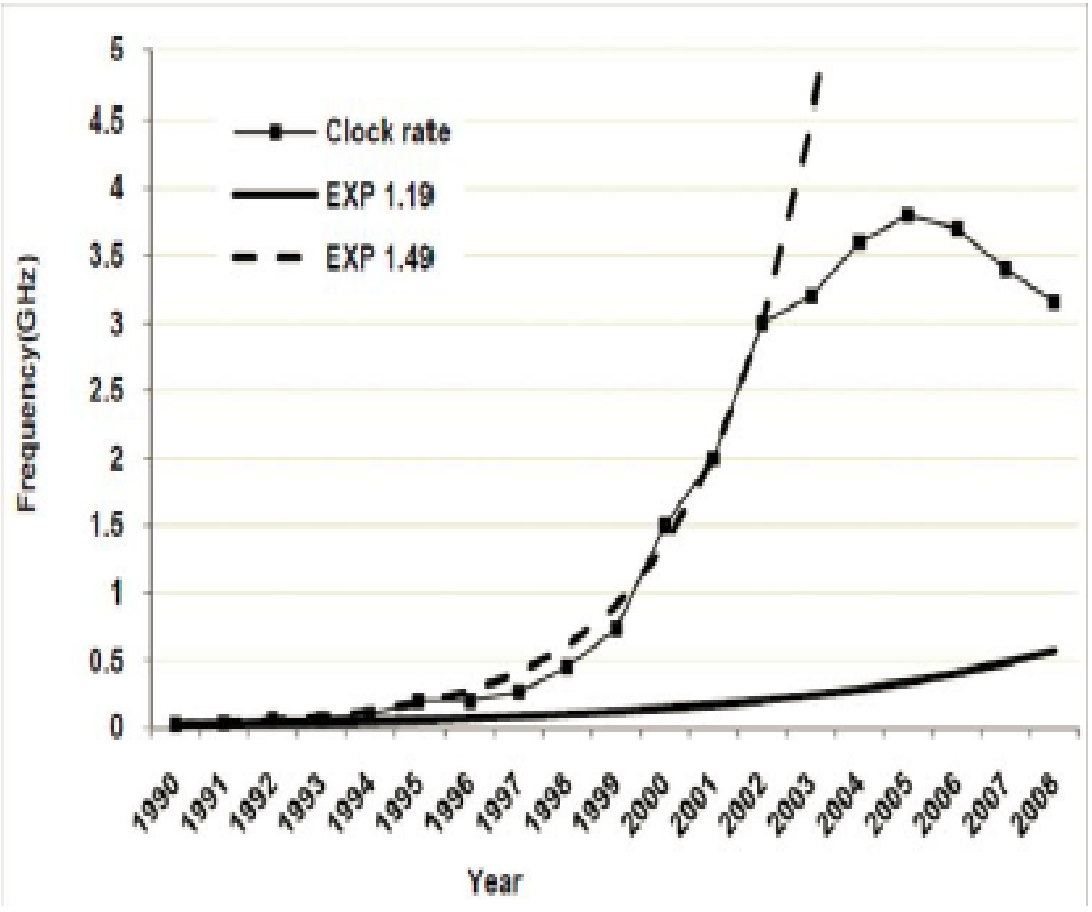
- "Number of transistors in a dense integrated circuit doubles approximately every two years."
- Rephrased as:
  - computing power doubles every 19-24 months, and
  - cost effectiveness (`performance/cost`) keeps pace.

Brief History

- ICPP, ISCA (1980/90s): parallel architectures popular topic.
- Whatever happened? Mid90 Killer-Micro:
  - path of least resistance: ever increasing the speed of Single CPU
  - Commercial arena: multiprocessors just an uniprocessor extension.
- What Changed? Multiprocessors Trend: Academia & Industry:
  - power complexity $P_{dynamic} \sim Freq^3$. Example!
  - Memory WALL: ever-increasing performance gap between processor & memory

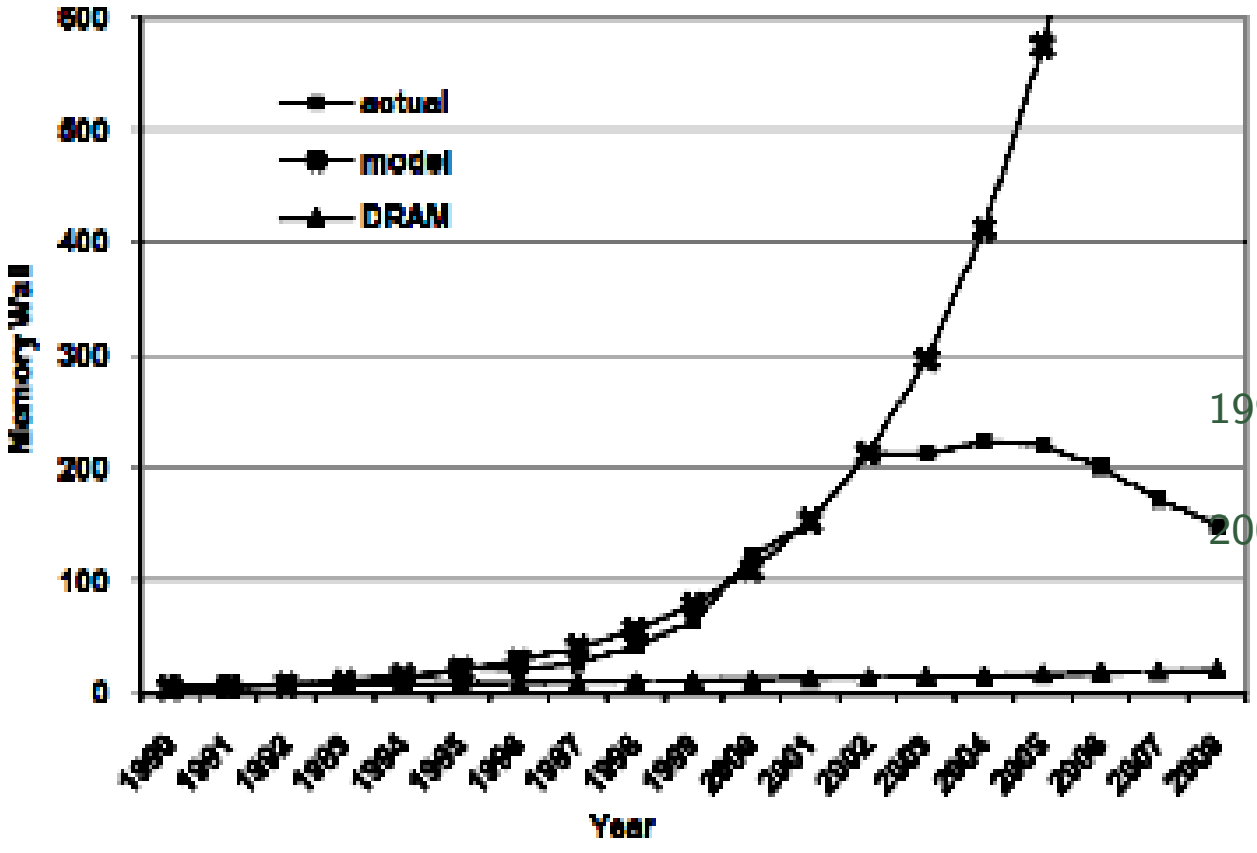
Processor: Clock Frequency/Rate

1990-2004: clock rate has increased exponentially.



2004: Intel cancels Pentium4 @4Ghz and shifts focus to multi-cores.

Memory Wall? Which Memory Wall??



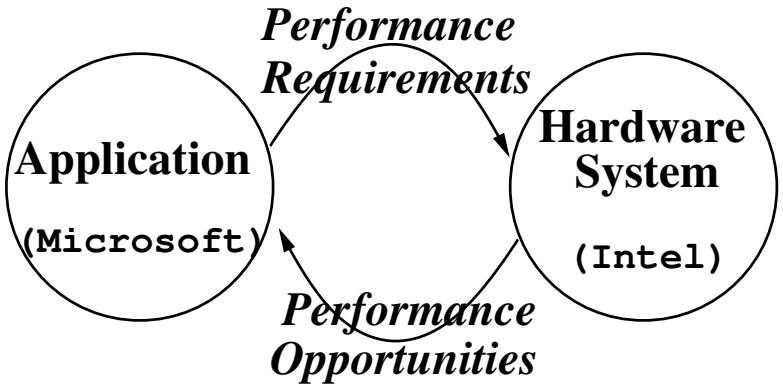
- `MemoryWall = mem_cycle/ proc_cycle`

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

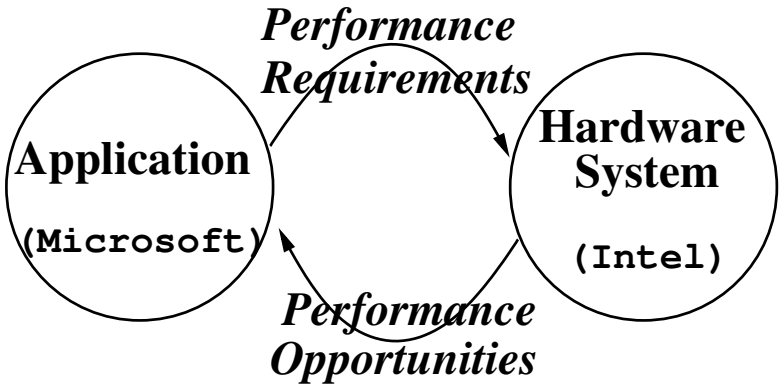
2002 exponential growth `MemoryWall = 200`

- Stalled since then.

Biggest Challenge For Parallel Hardware



Biggest Challenge For Parallel Hardware



Important Juncture:

- Trend Today: the number of cores grows exponentially.
- Biggest Challenge: develop efficient Massively-Parallel Software!
- think programs with parallelism in mind rather than hack some parallelism out of a sequential implementation.

When and Where?

Monday

Lecture: 13:00 - 15:00 (aud - NBB 2.0.G.064/070, Jagtvej 155)

Lab: 15:00 - 17:00 (aud - Aud 02 AKB, Universitetsparken 13)

Wednesday

Lecture: 10:00 - 12:00 (aud - NBB 2.0.G.064/070, Jagtvej 155)

Lab: 13:00 - 15:00 (aud - NBB 2.0.G.064/070, Jagtvej 155)

We also reserved (aud - NBB 2.0.G.064/070, Jagtvej 155) for Wednesday 15:00 – 17:00, just in case ...

Tentative Lecture/Lab Schedule

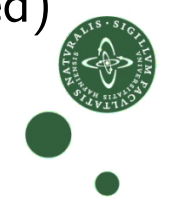
Your TAs:

- Anders Holst ¡anersholst@gmail.com¿ will grade and provide feedback for the weekly assignments & moderates Absalon/Discord discussions.
- Nikolaj will assist you in most of the Labs.

Cosmin will lead the lectures & the labs.

Continuous-evaluation assessment:

- four individual weekly assignments: 40% of final grade
- one group project + final presentation and discussion: 60% of final grade.
  - you may chose from multiple possible projects
  - presented tentatively during Lab at some point (to be announced)
  - or discuss your own project with me.
  - projects can be very practical in CUDA, Futhark, or more theoretical.



What does PMPH studies?

Hardware track studies the design space of the critical components of parallel hardware:

- processor (ILP, intra and inter-core)
- memory hierarchy (coherency)
- interconnect (inter-cores or core-cache routing)

What does PMPH studies?

Hardware track studies the design space of the critical components of parallel hardware:

- processor (ILP, intra and inter-core)
- memory hierarchy (coherency)
- interconnect (inter-cores or core-cache routing)

Software track studies programming models for expressing data parallelism + way to reason and optimize parallelism:

- High-level of abstraction:
  list-homomorphism $\equiv$ functional map-reduce style + flattening
- Low-level of abstraction:
  loops and transformations rooted in data dependency analysis
- Lecture Notes are available for the software track!

What does PMPH studies?

Hardware track studies the design space of the critical components of parallel hardware:

- processor (ILP, intra and inter-core)
- memory hierarchy (coherency)
- interconnect (inter-cores or core-cache routing)

Software track studies programming models for expressing data parallelism + way to reason and optimize parallelism:

- High-level of abstraction:
  list-homomorphism $\equiv$ functional map-reduce style + flattening
- Low-level of abstraction:
  loops and transformations rooted in data dependency analysis
- Lecture Notes are available for the software track!

Lab track applies in practice various optimizations/transformations learned on the software track.

1. Brief History: Parallelism Paves the Path to Higher Performance
2. Course Organization
3. Hardware Trends of Critical Components of a Parallel System
   - Processor
   - Memory
   - Interconnect
4. Technological Challenges/Constraints
   - Power
   - Reliability
   - Wire Delays
   - Design Complexity
   - CMOS Endpoint
5. List Homomorphisms (LH)
   - List Homomorphism Definition and 1st Theorem
   - Almost Homomorphisms Gorlatch'96

Reading

Please read chapter 1, "Motivation, hardware trends and technological constraints" from lecture notes.

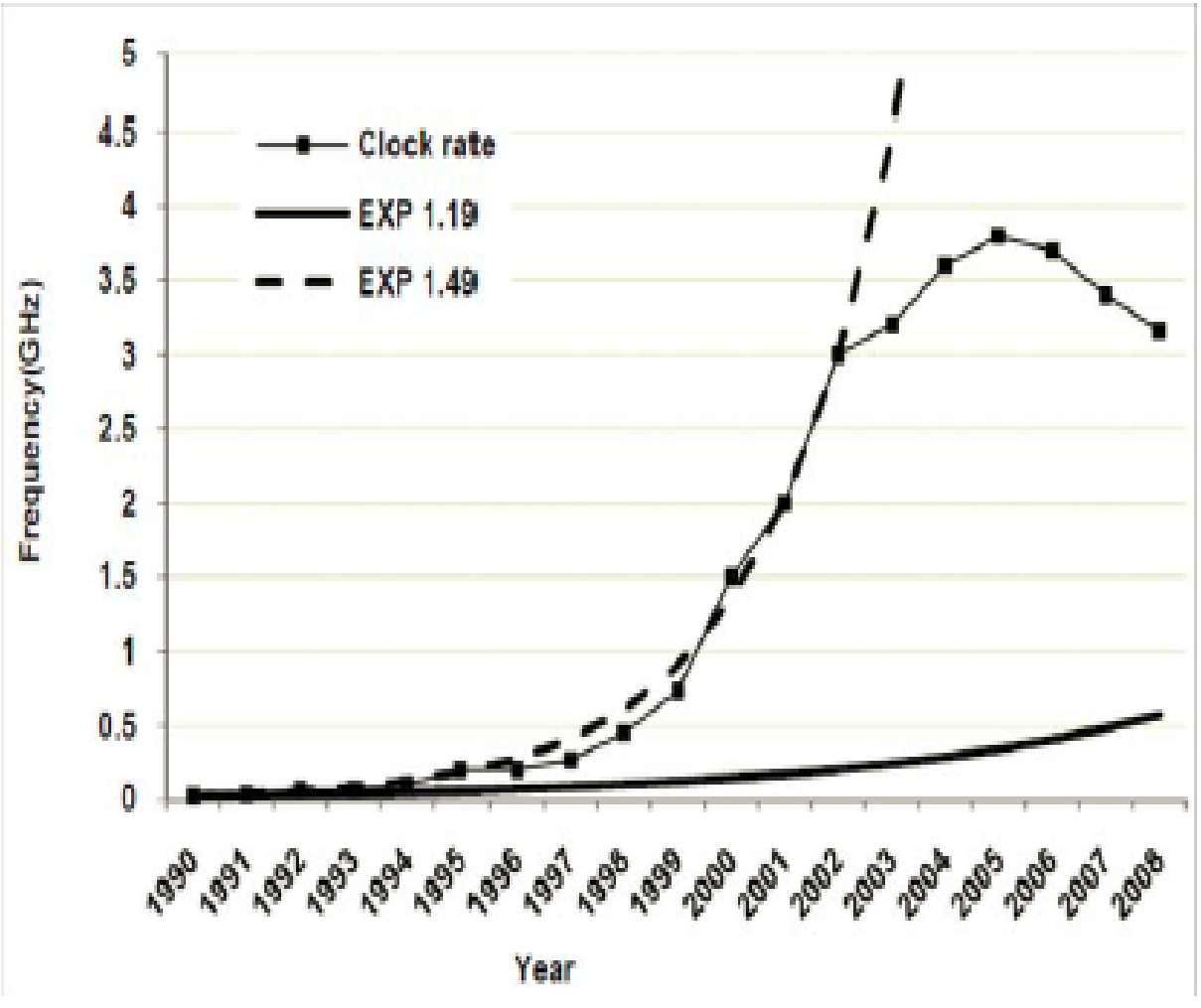
I am going to do a quick overview of it.

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

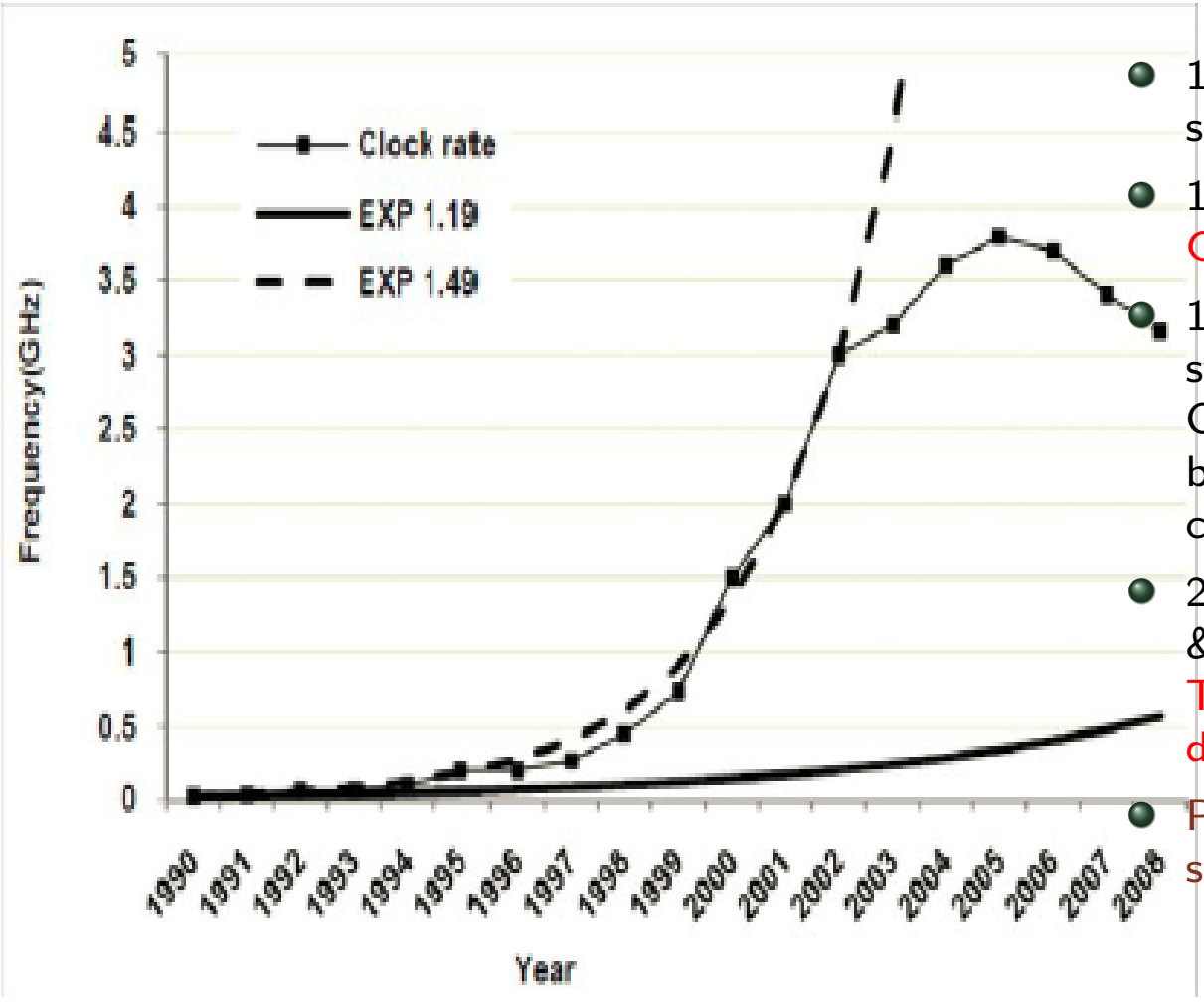
Processor: Clock Frequency/Rate

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



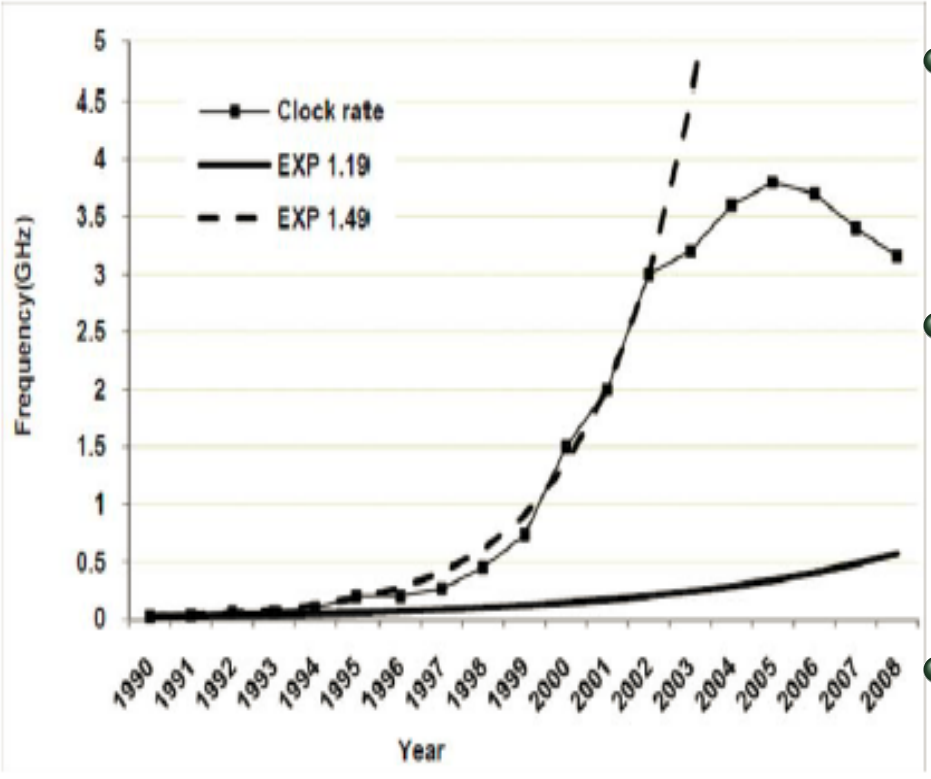
Processor: Clock Frequency/Rate

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



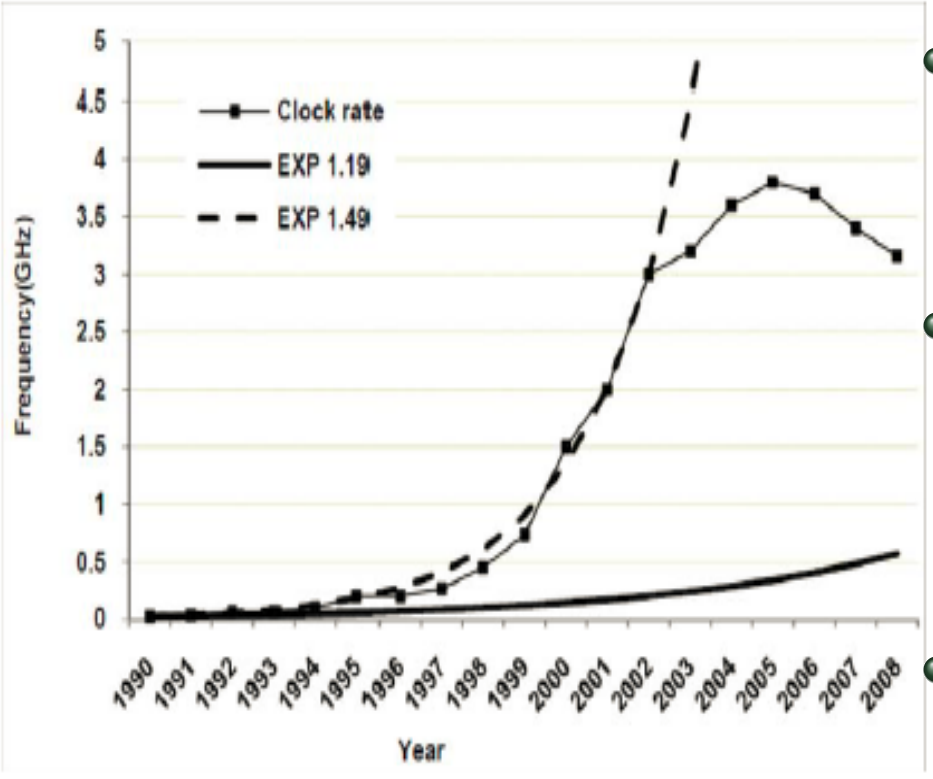
- 1.19× per-year due technology scaling (same hwd on new techn).
- 1990-2002: doubled every 21 months Curve 1.49×: 30GHz in 2008!
- 1.49 × −1.19×: very-deep (10-20 stages) pipelines! ILP via speculative OoO: register renaming, reorder buffs, branch prediction, lockup-free caches, memory disambiguation, 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

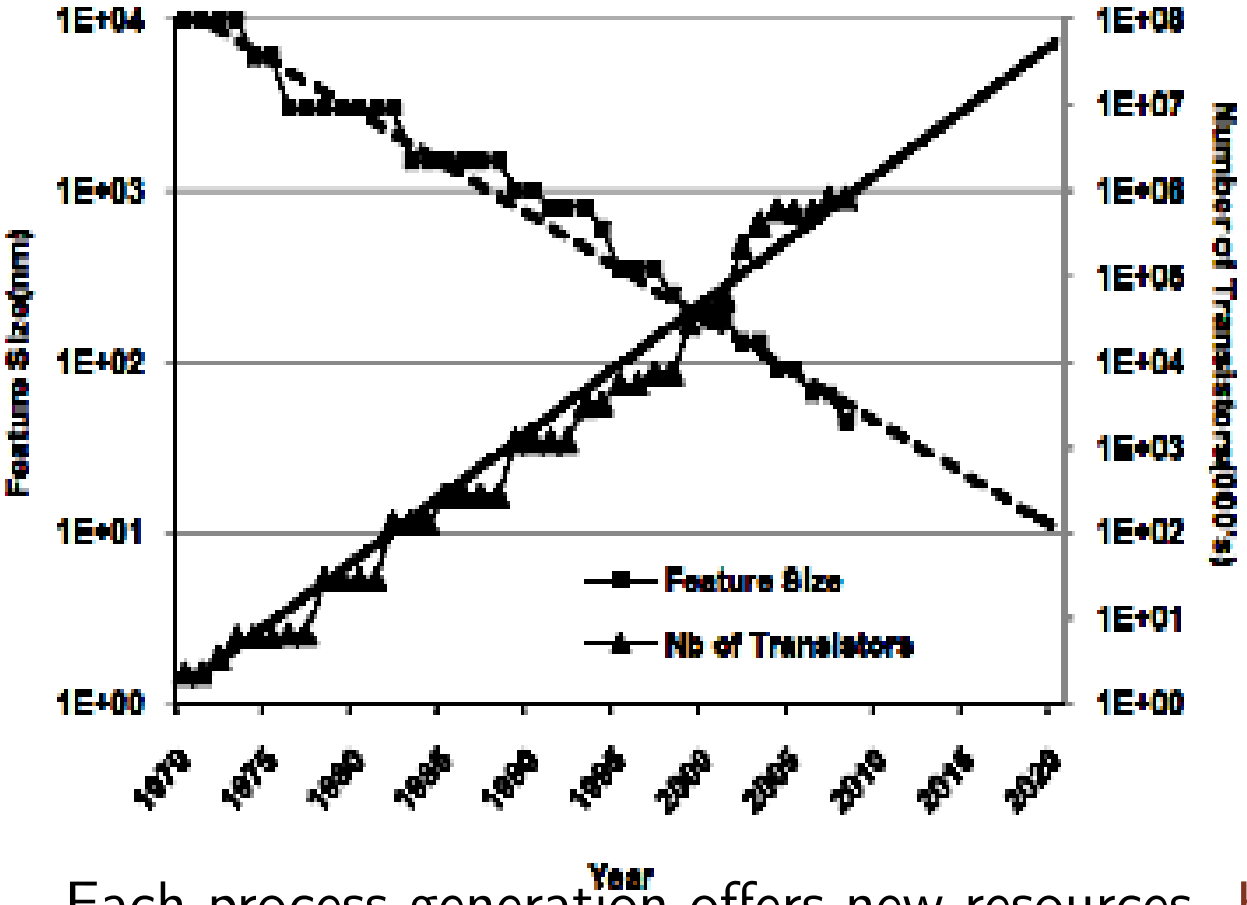


- Technology (process shrinkage): every generation transistors' switching speed increases 41%.
- Pipeline Depth: more stages ⇒ less complex ⇒ less gates/stage
  - # of gates delays dropped by 25% every process generation.
- Improved Circuit Design

Clock Rate Increase is Not Sustainable:

- Deeper pipelines: difficult to build useful stages with < 10 gates
- Wire delays: wire-transm speed ↑ 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

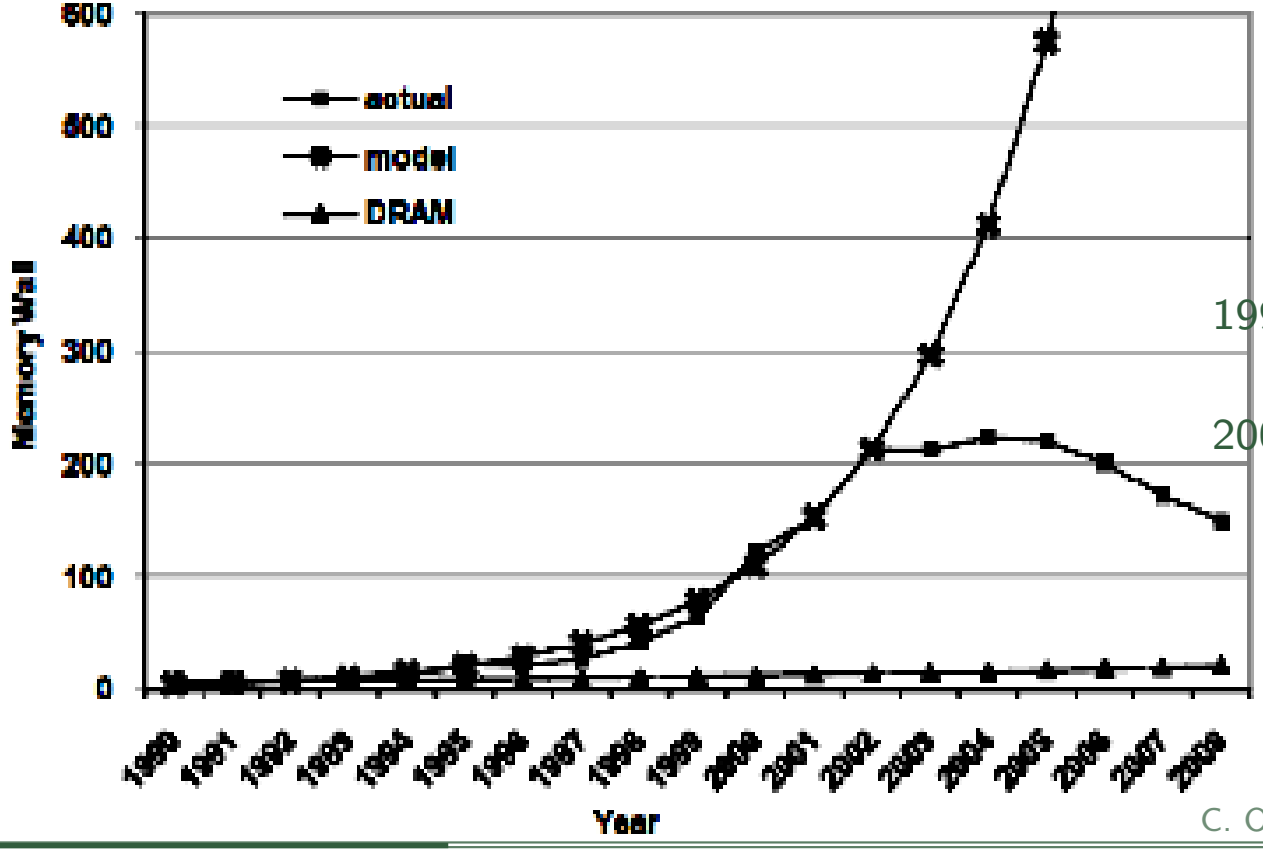
- (Main) Memory Wall: growing gap between processor and memory speed. Processor cannot execute faster than memory system can deliver data and instructions!

- Want Big, Fast & Cheap Memory System
  - access time increases with memory size as it is dominated by wire delays⇒ 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`

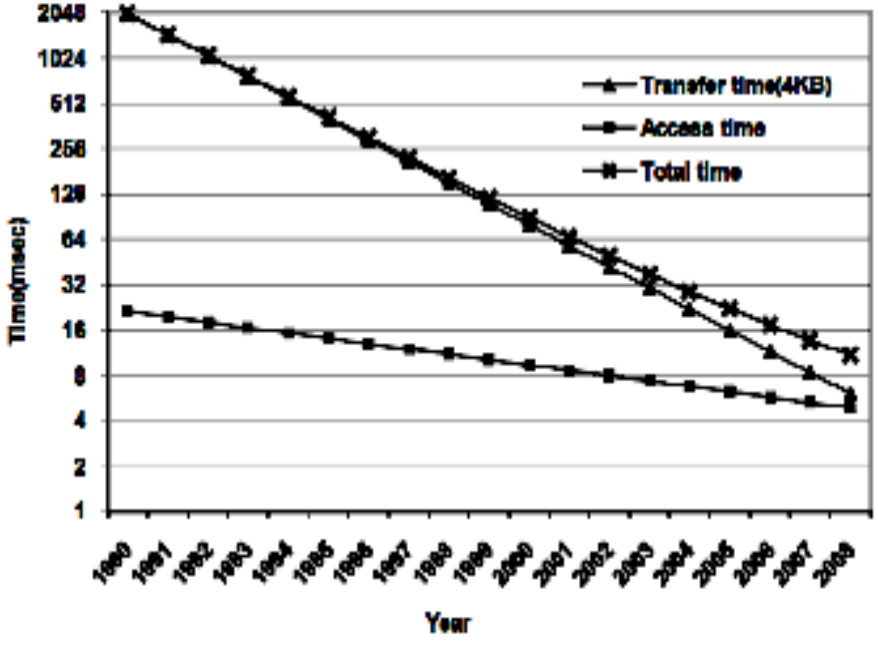
1990 `MemoryWall = 4` (25MHz,150ns)

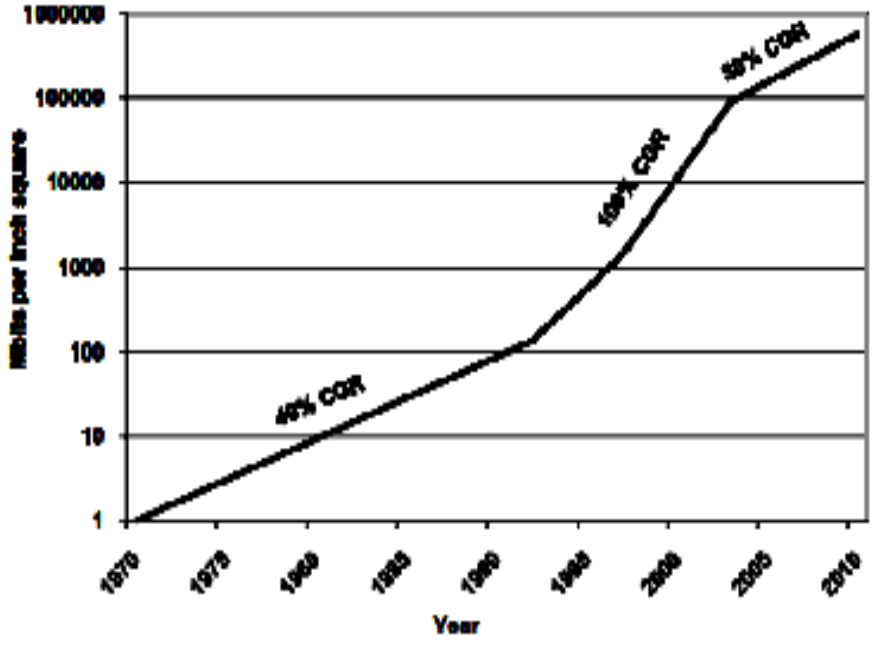
2002 exponential growth `MemoryWall = 200`

- 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).

1. Brief History: Parallelism Paves the Path to Higher Performance
2. Course Organization
3. Hardware Trends of Critical Components of a Parallel System
   - Processor
   - Memory
   - Interconnect
4. Technological Challenges/Constraints
   - Power
   - Reliability
   - Wire Delays
   - Design Complexity
   - CMOS Endpoint
5. List Homomorphisms (LH)
   - List Homomorphism Definition and 1st Theorem
   - Almost Homomorphisms Gorlatch'96

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
  $P_{static} = V I_{sub} \sim V e^{-kV_T/T}$ (caches)

  V supply voltage, f clock rate, α activity factor, αf the rate at which gates switch, T temperature
- 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)

  V supply voltage, f clock rate, α activity factor, αf the rate at which gates switch, T temperature
- 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
  $P_{static} = V I_{sub} \sim V e^{-kV_T/T}$ (caches)

  V supply voltage, f clock rate, α activity factor, αf the rate at which gates switch, T temperature
- 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, no matter of frequency and whether it switches or not.

  Proportional to the area of circuit but independent of clock rates and circuit activity.
  - 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):
  - Corruption Sources: cosmic rays, alpha particles radiating from the packaging material, electrical noise;
  - 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
  - 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 (?)

1. Brief History: Parallelism Paves the Path to Higher Performance
2. Course Organization
3. Hardware Trends of Critical Components of a Parallel System
   - Processor
   - Memory
   - Interconnect
4. Technological Challenges/Constraints
   - Power
   - Reliability
   - Wire Delays
   - Design Complexity
   - CMOS Endpoint
5. List Homomorphisms (LH)
   - List Homomorphism Definition and 1st Theorem
   - Almost Homomorphisms Gorlatch'96

Shape of a List-Homomorphic Program (LH)

Realm of finite lists:

- ++ denotes list concatenation:
  `[1, 2, 3] ++ [4, 5, 6, 7]` $\equiv$ `[1, 2, 3, 4, 5, 6, 7]`
- empty list `[]` is the neutral element: `[] ++ x` $\equiv$ `x ++ []` $\equiv$ `x`

LH: a special form of divide and conquer programming:

```
h [ ] = e
h [x] = f x
h (x ++ y) = (h x) ⊙ (h y)
```

Shape of a List-Homomorphic Program (LH)

Realm of finite lists:

- ++ denotes list concatenation:
  `[1, 2, 3] ++ [4, 5, 6, 7]` $\equiv$ `[1, 2, 3, 4, 5, 6, 7]`
- empty list `[]` is the neutral element: `[] ++ x` $\equiv$ `x ++ []` $\equiv$ `x`

LH: a special form of divide and conquer programming:

```
h [ ] = e
h [x] = f x
h (x ++ y) = (h x) ⊙ (h y)
```

A well-defined program requires that no matter how the input list is partitioned into `x ++ y`, the result is the same!

```
--computes the length of a list,
--(how many elements a list has)
len :: [T] -> Int
len [ ]     = ???
len [x]     = ???
len (x++y) = (len x) ??? (len y)
```

Shape of a List-Homomorphic Program (LH)

Realm of finite lists:

- ++ denotes list concatenation:
  `[1, 2, 3] ++ [4, 5, 6, 7]` $\equiv$ `[1, 2, 3, 4, 5, 6, 7]`
- empty list `[]` is the neutral element: `[] ++ x` $\equiv$ `x ++ []` $\equiv$ `x`

LH: a special form of divide and conquer programming:

```
h [ ]      = e
h [x]      = f x
h( x ++ y) = (h x) ⊙ (h y)
```

A well-defined program requires that no matter how the input list is partitioned into `x ++ y`, the result is the same!

```
-- one :: Int -> Int
-- one(x) = 1
len :: [T] -> Int
len [ ]      = 0
len [x]      = one x -- ≡ 1
len (x ++ y) = (len x) + (len y)
```

Shape of a List-Homomorphic Program (LH)

Realm of finite lists:

- ++ denotes list concatenation:
  `[1, 2, 3] ++ [4, 5, 6, 7]` $\equiv$ `[1, 2, 3, 4, 5, 6, 7]`
- empty list `[]` is the neutral element: `[] ++ x` $\equiv$ `x ++ []` $\equiv$ `x`

LH: a special form of divide and conquer programming:

```
h [ ]      = e
h [x]      = f x
h ( x ++ y) = (h x) ⊙ (h y)
```

A well-defined program requires that no matter how the input list is partitioned into `x ++ y`, the result is the same!

```
--Assume p :: T -> Bool given,
--compute whether all elements of
--a list satisfy predicate p.
all_p :: [T] -> Bool
all_p [ ]   = ???
all_p [x]   = ???
all_p (x++y) = (all_p x) ??? (all_p y)
```

Shape of a List-Homomorphic Program (LH)

Realm of finite lists:

- ++ denotes list concatenation:
  `[1, 2, 3] ++ [4, 5, 6, 7]` $\equiv$ `[1, 2, 3, 4, 5, 6, 7]`
- empty list `[]` is the neutral element: `[] ++ x` $\equiv$ `x ++ []` $\equiv$ `x`

LH: a special form of divide and conquer programming:

```
h [ ]     = e
h [x]     = f x
h (x ++ y) = (h x) ⊙ (h y)
```

A well-defined program requires that no matter how the input list is partitioned into `x ++ y`, the result is the same!

```
--Assume p :: T -> Bool given,
--compute whether all elements of
--a list satisfy predicate p.
all_p :: [T] -> Bool
all_p [ ]   = True
all_p [x]   = p x
all_p (x++y) = (all_p x) && (all_p y)
```

Why would it be incorrect to say that `all_p [] = False`?

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.

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 $\forall u, v \in S$, $h(u \oplus v) \equiv h(u) \odot h(v)$.

Shape of a List-Homomorphic Program (LH)

Realm of finite lists:

- ++ denotes list concatenation:
  `[1, 2, 3] ++ [4, 5, 6, 7]` $\equiv$ `[1, 2, 3, 4, 5, 6, 7]`
- empty list `[]` is the neutral element: `[] ++ x` $\equiv$ `x ++ []` $\equiv$ `x`

LH: a special form of divide and conquer programming:

```
h [ ]     = e
h [x]     = f x
h (x ++ y) = (h x) ⊙ (h y)
```

A well-defined program requires that no matter how I partition the input list into `(x ++ y)`, the result is the same!

EXERCISE: prove that $(Img(h), \odot)$ is a monoid with neutral element e.

Basic Blocks of Parallel Programming: Map

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

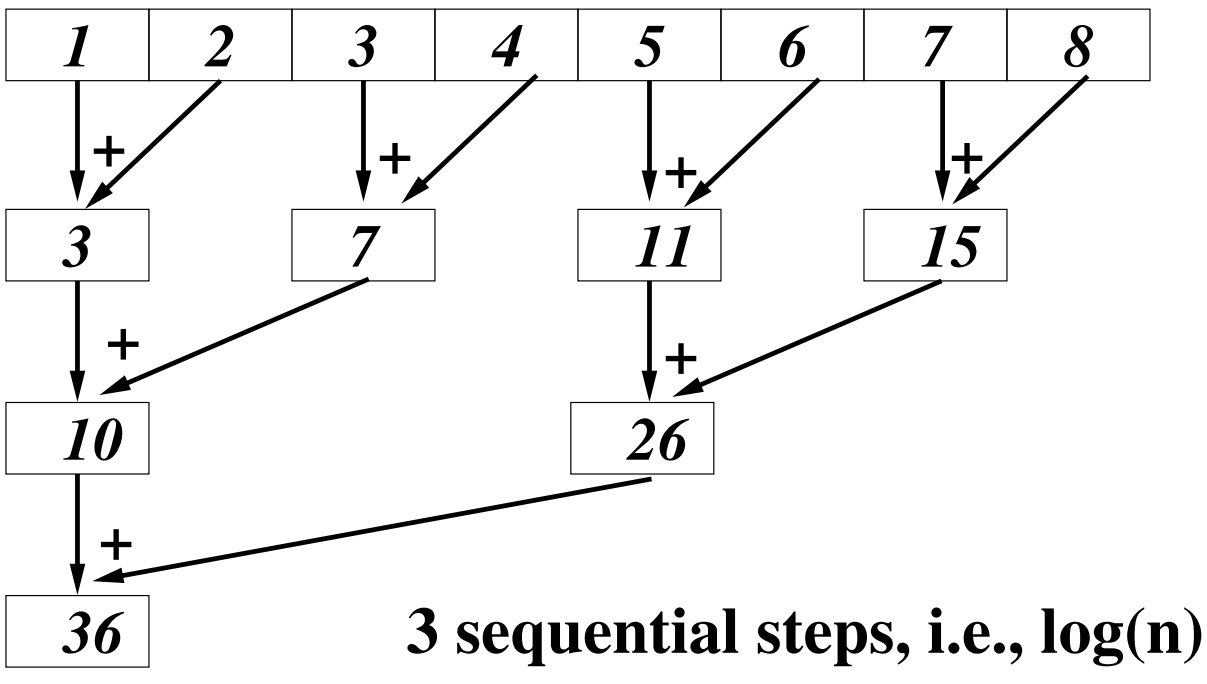
$$
\begin{array}{lllllll}
\mathsf{x} = & \mathsf{map}\ \mathsf{f}\ [& a_1, & a_2, & .., & a_n &] \\
 & & \downarrow & \downarrow & & \downarrow & \\
\mathsf{x} \equiv & [& \mathsf{f}\ a_1, & \mathsf{f}\ a_2, & .., & \mathsf{f}\ a_n &]
\end{array}
$$

Basic Blocks of Parallel Programming: Reduce

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

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.



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

1st List-Homomorphism Theorem [Meertens]

```
h [ ]      = e
h [x]      = f(x)                    ⇔    h  ≡  (reduce ⊙ e) ∘ (map f)
h (x ++ y) = (h x) ⊙ (h y)
```

Important Note: $\odot$ needs to be associative and e needs to be the neutral element of $\odot$!

1st List-Homomorphism Theorem [Meertens]

```
h [ ]      = e
h [x]      = f(x)
h (x ++ y) = (h x) ⊙ (h y)
```

$\Leftrightarrow$ `h` $\equiv$ `(reduce ⊙ e) ∘ (map f)`

Important Note: $\odot$ needs to be associative and e needs to be the neutral element of $\odot$!

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

`len` $\equiv$

1st List-Homomorphism Theorem [Meertens]

```
h [ ]      = e
h [x]      = f(x)
h (x ++ y) = (h x) ⊙ (h y)
```

$\Leftrightarrow$ `h ≡ (reduce ⊙ e) ∘ (map f)`

Important Note: $\odot$ needs to be associative and e needs to be the neutral element of $\odot$!

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

```
len ≡ (reduce (+) 0) ∘
      (map one)
```

```
all_p [ ]      = True
all_p [x]      = p(x)
all_p (x++y) = (all_p x) && (all_p y)
```

`all_p ≡`

1st List-Homomorphism Theorem [Meertens]

```
h [ ]      = e
h [x]      = f(x)
h (x ++ y) = (h x) ⊙ (h y)
```

$\Leftrightarrow$ `h ≡ (reduce ⊙ e) ∘ (map f)`

Important Note: $\odot$ needs to be associative and e needs to be the neutral element of $\odot$!

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

```
len ≡ (reduce (+) 0) ∘
      (map one)
```

```
all_p [ ]      = True
all_p [x]      = p(x)
all_p (x++y)   = (all_p x) && (all_p y)
```

```
all_p ≡ (reduce (&&) True) ∘
        (map p)
```

Demistifying function composition $\circ$

Definition: $(g \circ f)\, x \equiv g(f\ x)$

It follows that:

$(\ (\texttt{reduce}\ \odot\ e_{\odot})\ \circ\ (\texttt{map}\ f)\ )\ xs \equiv \texttt{reduce}\ \odot\ e_{\odot}\ (\texttt{map}\ f\ xs)$

or as a program:

```
let temp_array = map f xs
in  reduce ⊙ e⊙ temp_array
```

List Homomorphism Invariants

Theorem (List-Homomorphism Promotions)

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

1. $(\texttt{map}\ f) \circ (\texttt{map}\ g) \equiv \texttt{map}\ (f \circ g)$

2. $(\texttt{map}\ f) \circ (\texttt{reduce}\ (++)\ []) \equiv (\texttt{reduce}\ (++)\ []) \circ (\texttt{map}\ (\texttt{map}\ f))$

3. $(\texttt{reduce}\ \odot\ e_{\odot}) \circ (\texttt{reduce}\ (++)\ []) \equiv (\texttt{reduce}\ \odot\ e_{\odot}) \circ (\texttt{map}\ (\texttt{reduce}\ \odot\ e_{\odot}))$

- 2. 3. $\Rightarrow$ code generation for coarse paralleism: 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.

More Notation

Assume $\texttt{distr}_p$ distributes a list into p sublists of roughly the same number of elements.

Assume $\texttt{distr}_{shp}$ distributes a list according to a given shape, e.g., if `arr = [1,2,3,4,5]` and `shp = [2,3]` then $\texttt{distr}_{shp}$ `arr = [[1,2], [3,4,5]]`.

Assume the representation of a list of lists is flat, i.e., we keep a shape around.

List Homomorphism Invariants: Demo $\Rightarrow$

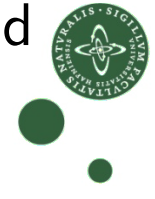
2. $(\texttt{map}\ f) \circ (\texttt{reduce}\ (++)\ []) \Rightarrow (\texttt{reduce}\ (++)\ []) \circ (\texttt{map}\ (\texttt{map}\ f))$

Useful for coarse-parallel code generation: a map computation can be implemented by chunking the original array into number-of-processors p subarrays. Then each processor performs the map computation sequentially on its subarray.

Observation: $(\texttt{reduce}\ (++)\ []) \circ \texttt{distr}_p$ is the identity $id(x) = x$.

$$map\ f \equiv (map\ f) \circ id \equiv (map\ f) \circ (\texttt{reduce}\ (++)\ []) \circ \texttt{distr}_p$$
$$\stackrel{2}{\Rightarrow} (\texttt{reduce}\ (++)\ []) \circ (\texttt{map}\ (\texttt{map}\ f)) \circ \texttt{distr}_p$$

For coarse-grain parallelism, the outer map is executed in parallel and the inner map is sequentialized.



List Homomorphism Invariants: Demo ⇐

2. $(\texttt{map}\ f) \circ (\texttt{reduce}\ (+\!+)\ []) \equiv (\texttt{reduce}\ (+\!+)\ []) \circ (\texttt{map}\ (\texttt{map}\ f))$

Useful for flattening or form load balancing: assume we have a computation f that has to be applied to each element of an (irregular) array of arrays, i.e., `map (map f)`, where subarrays may have very different lengths. Assume the hardware has much more parallelism than any of the two levels of map can saturate. How do we transform the program to a flat-parallel?

Composing each side of the 2^{nd} promotion lemma by $\texttt{distr}_{shp}$ yields:

$$\texttt{map}\ (\texttt{map}\ f) \equiv \texttt{distr}_{shp} \circ (\texttt{map}\ f) \circ (\texttt{reduce}\ (+\!+)\ [])$$

We have just transformed two-level parallelism into one-level (flat) parallelism. If we have more parallelism than number of processors we apply the coarse-parallel code generation discussed before that results in load-balanced computation.

Optimizing Map-Reduce Computation (Exercise)

Theorem (Optimized Map Reduce)

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

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

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

What is the Interpretation of this Theorem?

Theorem (Optimized Map Reduce)

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

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

Why is this useful?

What is the Interpretation of this Theorem?

Theorem (Optimized Map Reduce)

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

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

Why is this useful?

The theorem famously states that the inter-processor communication of a map-reduce computation can be minimized by:

- splitting the array into number-of-processor p subarrays,
- then by having each processor perform the original computation sequentially on its assigned subarray,
- and by reducing in parallel the p local results across processors.

1. Brief History: Parallelism Paves the Path to Higher Performance
2. Course Organization
3. Hardware Trends of Critical Components of a Parallel System
   - Processor
   - Memory
   - Interconnect
4. Technological Challenges/Constraints
   - Power
   - Reliability
   - Wire Delays
   - Design Complexity
   - CMOS Endpoint
5. List Homomorphisms (LH)
   - List Homomorphism Definition and 1st Theorem
   - Almost Homomorphisms Gorlatch'96

Maximum Segment Sum Problem (MSSP)

"Systematic Extraction and Implementation of Divide-and-Conquer Parallelism", Sergei Gorlatch, 1996.

Intuition: a non-homomorphic function g can be sometimes lifted to a homomorphic one f, by computing a baggage of *extra info*.

The initial fun obtained by projecting the homomorphic result:
$g = \pi \circ f$

Maximum-Segment Sum Problem (MSSP):
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments.
The result is only the maximal sum (not the segment's members).
For simplicity lets assume we are interested only in **positive sums**.

E.g., `mss` [1, -2, 3, 4, -1, 5, -6, 1] = 11
(the corresponding segment is [3, 4, -1, 5]).

MSSP: Preliminary Reasoning

Maximum-Segment Sum Problem (MSS):
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments.
The result is only the maximal sum (not the segment's members).
For simplicity lets assume we are interested only in **positive sums**.

A first straightforward/naive attempt:

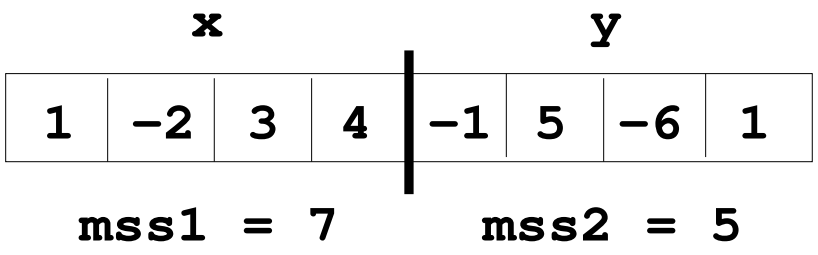
```
mss [ ]       = 0
mss [a]       = a ↑ 0  //↑ denotes Max
mss (x ++ y) = mss(x) ??? mss(y)
```

MSSP: Preliminary Reasoning

Maximum-Segment Sum Problem (MSS):
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments.
The result is only the maximal sum (not the segment's members).
For simplicity lets assume we are interested only in **positive sums**.

A first straightforward/naive attempt:

```
mss [ ]       = 0
mss [a]       = a ↑ 0   //↑ denotes Max
mss (x ++ y) = mss(x) ??? mss(y)
```



Which case is problematic?

How to combine mss1 and mss2?

`mss1 + mss2 = 12` **Incorrect!**

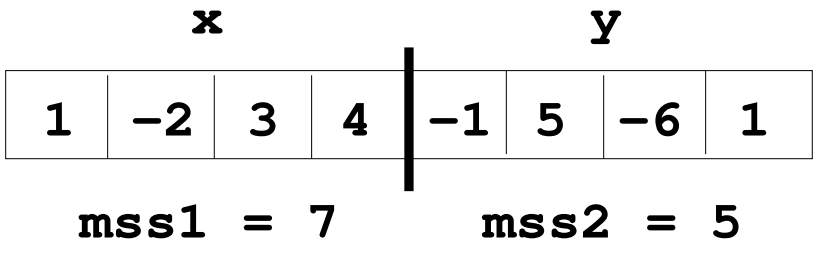
`max(mss1,mss2) = 7` **Incorrect!**

MSSP: Preliminary Reasoning

Maximum-Segment Sum Problem (MSS):
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments.
The result is only the maximal sum (not the segment's members).
For simplicity lets assume we are interested only in **positive sums**.

A first straightforward/naive attempt:

```
mss [ ]       = 0
mss [a]       = a ↑ 0   //↑ denotes Max
mss (x ++ y) = mss(x) ??? mss(y)
```



How to combine mss1 and mss2?

mss1 + mss2 = 12 **Incorrect!**

max(mss1,mss2) = 7 **Incorrect!**

Which case is problematic?

Answer: when the segment of interest lies partly in x and partly in y!

MSSP: A Better Reasoning

The problematic case is when the segment of interest lies partly in $\mathtt{x}$ and partly in $\mathtt{y}$!

We need to compute extra information:

MSSP: A Better Reasoning

The problematic case is when the segment of interest lies partly in `x` and partly in `y`!

We need to compute extra information:

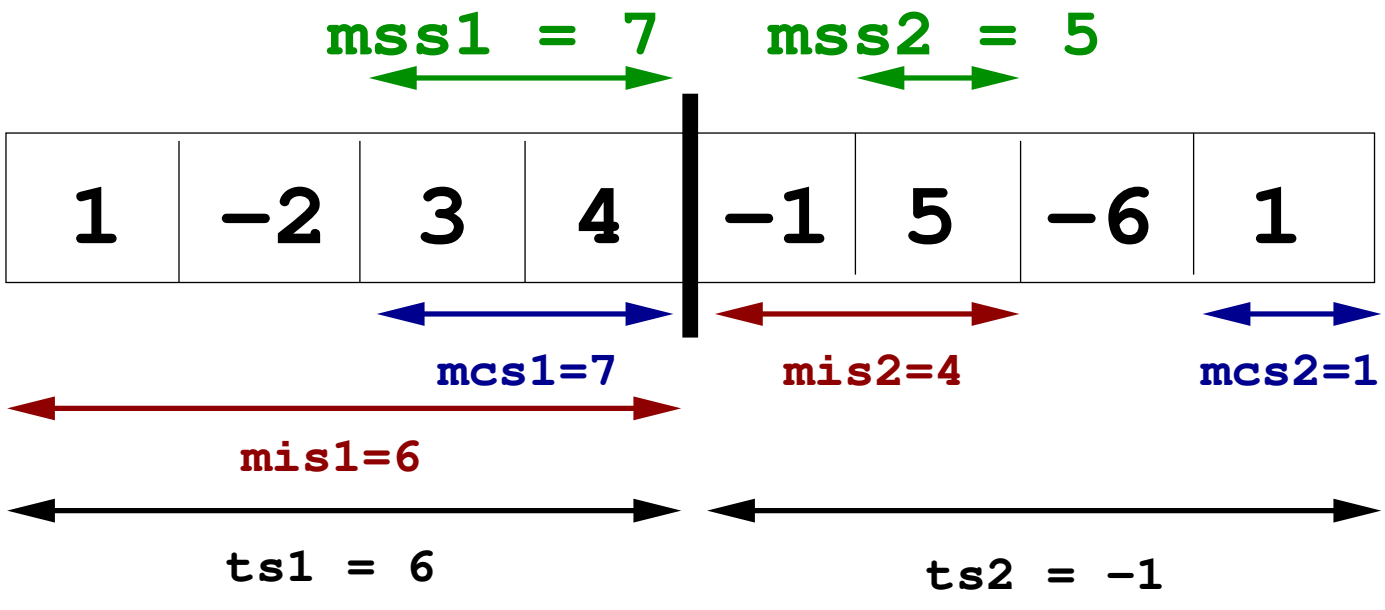
- maximum concluding segment
- maximum initial segment
- total segment sum

MSSP: A Better Reasoning

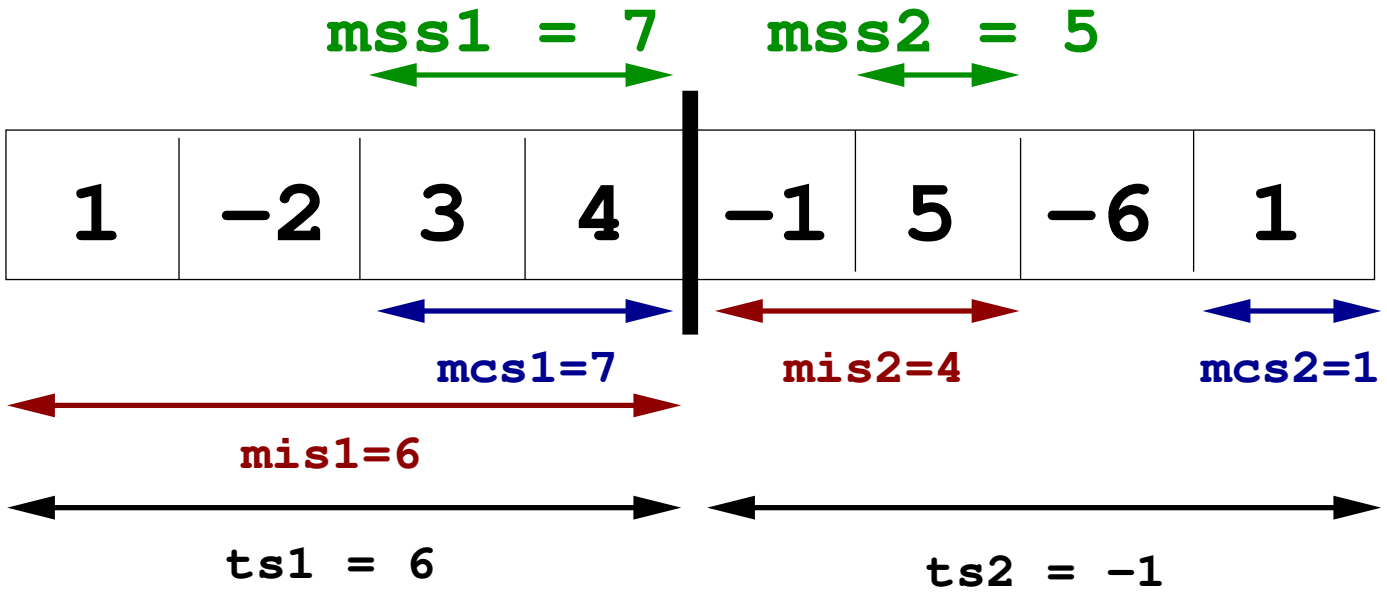
The problematic case is when the segment of interest lies partly in `x` and partly in `y`!

We need to compute extra information:

- maximum concluding segment
- maximum initial segment
- total segment sum



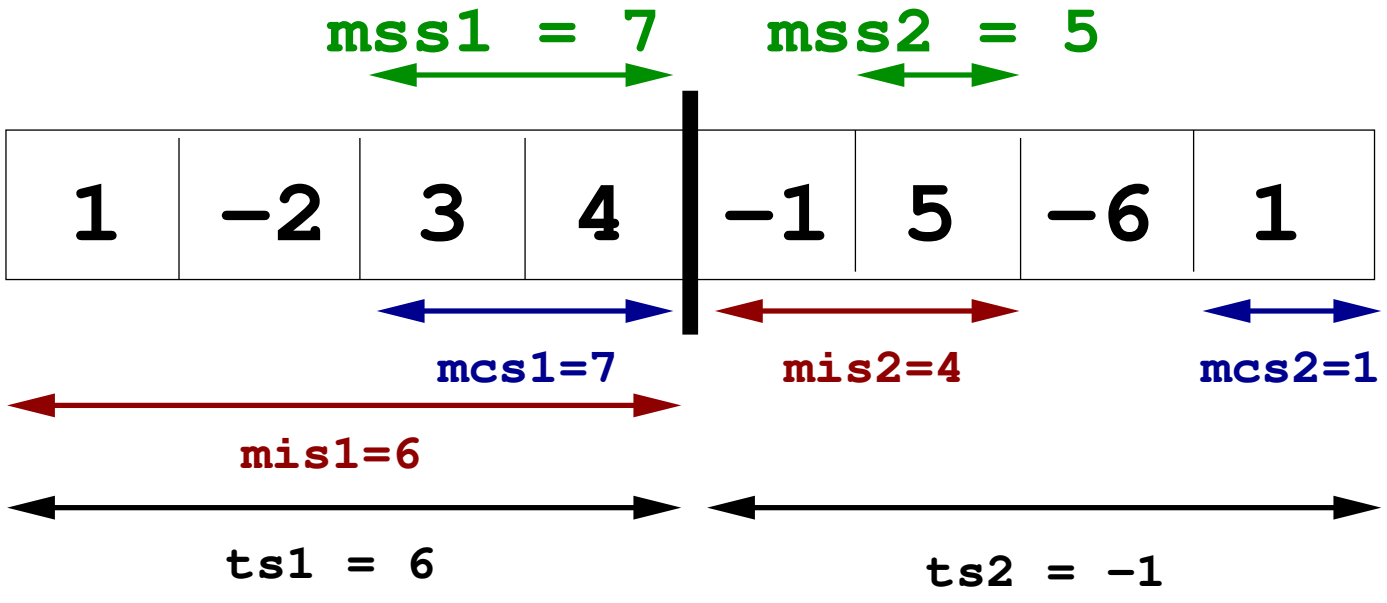
MSSP: Deriving the Implementation



Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result (i.e., for the concatenation of the two segments). $\uparrow$ denotes `max`.

`mis =`

MSSP: Deriving the Implementation

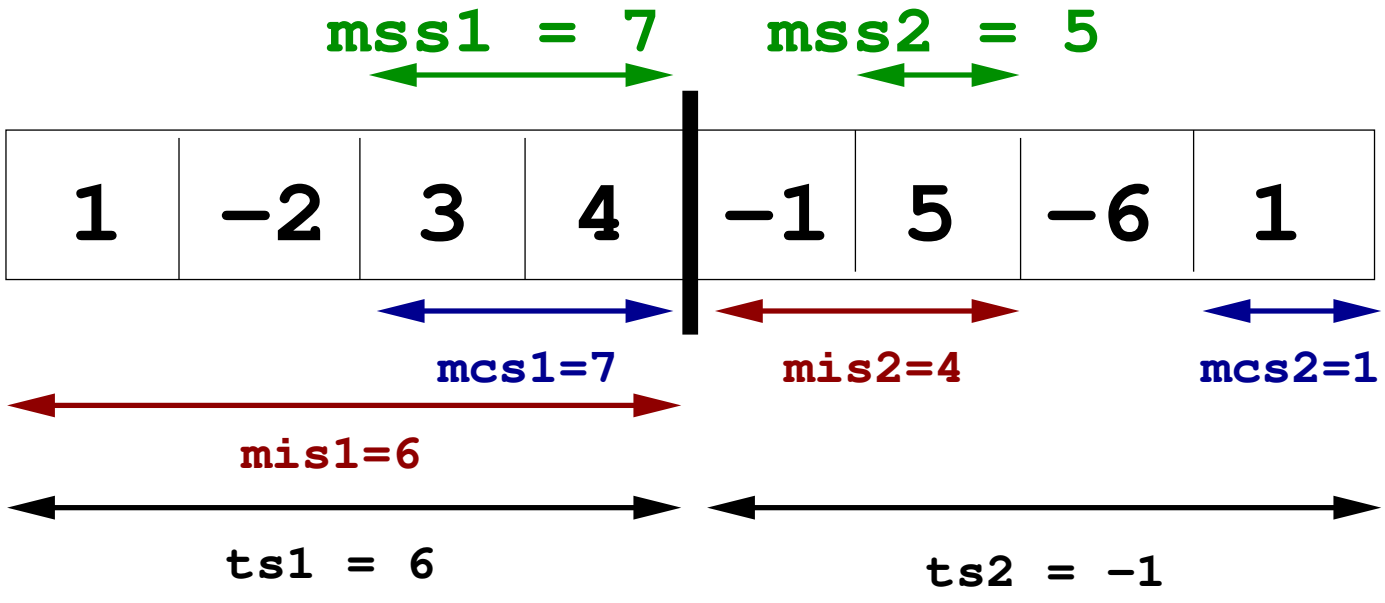


Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result (i.e., for the concatenation of the two segments). ↑ denotes `max`.

```
mis = mis1 ↑ (ts1 + mis2)

mcs =
```

MSSP: Deriving the Implementation



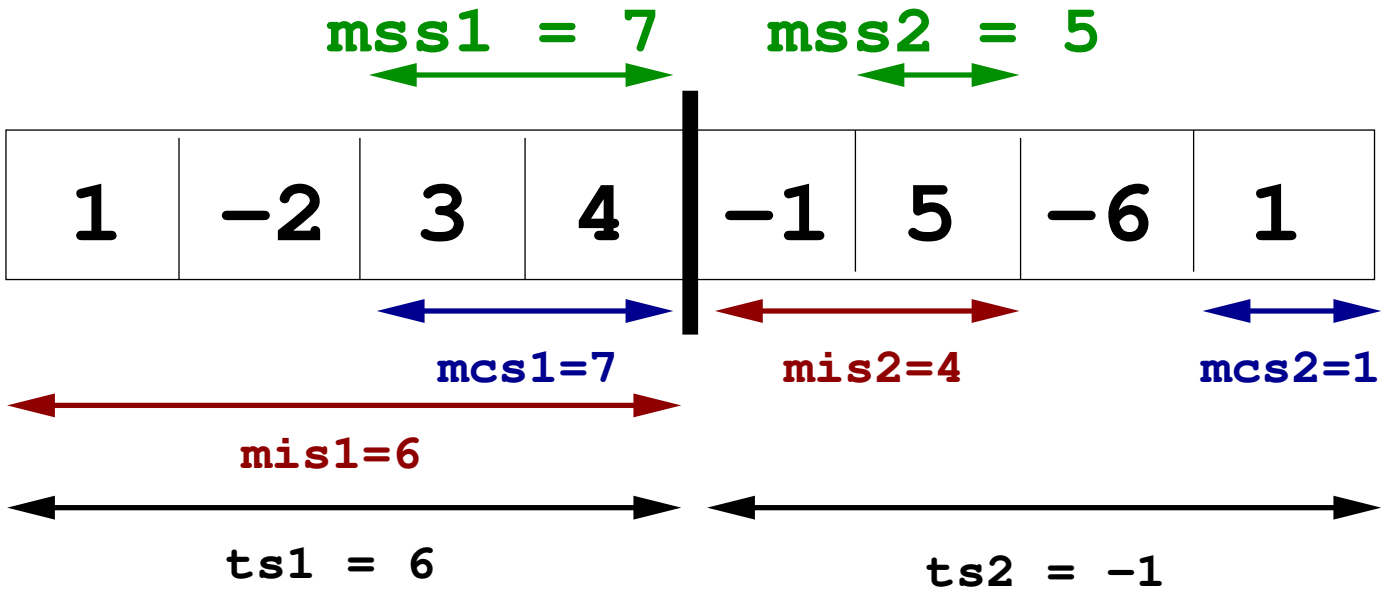
Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result (i.e., for the concatenation of the two segments). ↑ denotes `max`.

```
mis = mis1 ↑ (ts1 + mis2)

mcs = mcs2 ↑ (mcs1 + ts2)

mss =
```

MSSP: Deriving the Implementation



Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result (i.e., for the concatenation of the two segments). ↑ denotes `max`.

```
mis = mis1 ↑ (ts1 + mis2)

mcs = mcs2 ↑ (mcs1 + ts2)

mss = mss1 ↑ mss2 ↑ (mcs1 + mis2)

ts  = ts1 + ts2
```

Maximum-Segment Sum in Futhark

```
let max (x:i32, y:i32) = i32.max x y

let redOp (mssx, misx, mcsx, tsx)
          (mssy, misy, mcsy, tsy)
        : (i32, i32, i32, i32) =
  let mss = max (mssx, max (mssy, mcsx + misy))
  let mis = max (misx, tsx+misy)
  let mcs = max (mcsy, mcsx+tsy)
  let ts  = tsx + tsy
  in  (mss, mis, mcs, ts)

let mapOp (x: i32): (i32,i32,i32,i32) =
  ( max(x,0), max(x,0), max(x,0), x )

let main(xs: []i32): i32 =
  let (x, _, _, _) =
    reduce redOp (0,0,0,0) (map mapOp xs)
  in x
```

The baggage: 3 extra integers (`misx, mcsx, tsx`) and a constant number of integer operations per communication stage.

Longest Satisfying Segment Problems

- Class of problems which requires to find the longest segment of a list for which some property holds, such as:
- longest sequence of zeros, or longest sequence made from the same number, or longest sorted sequence.
- Not all predicates can be written as a list homomorphism, e.g., longest sequence whose sum is 0.

Restrict The Shape of the Predicate to:

```
p []           = True
p [x]          = ...
p [x, y]       = ...
p [x : y : zs] = (p [x,y]) ∧ p (y : zs)
```

$\wedge$ denotes the logical AND operator.

Longest Satisfying Segment Problems

Restrict the Shape of the Predicate:

```
zeros [x]   = x == 0          same [x]   = True     sorted [x]   = True
zeros [x,y] =  (zeros [x])    same [x,y] = x == y   sorted [x,y] = x <= y
             ∧ (zeros [y])
```

Extra Baggage:

- As before, the length of the longest initial/concluding satisfying segments (`lis`/`lcs`), and the total list length (`tl`).
- When considering the concatenation of the `(lcs, lis)` pair, it is not guaranteed that the result satisfies the predicate, e.g., `(sorted x)` $\wedge$ `(sorted y)` $\not\Rightarrow$ `sorted x++y`.

Longest Satisfying Segment Problems

Restrict the Shape of the Predicate:

```
zeros [x]   = x == 0          same [x]   = True     sorted [x]   = True
zeros [x,y] =  (zeros [x])    same [x,y] = x == y   sorted [x,y] = x <= y
             ∧ (zeros [y])
```

Extra Baggage:

- As before, the length of the longest initial/concluding satisfying segments (`lis`/`lcs`), and the total list length (`tl`).
- When considering the concatenation of the `(lcs, lis)` pair, it is not guaranteed that the result satisfies the predicate, e.g., `(sorted x)` $\wedge$ `(sorted y)` $\not\Rightarrow$ `sorted x++y`.
- We also need the *last* element of `lcs` and the *first* elem of `lis`,
- in order to compute whether `(lcs x)` is *connected* to `(lis y)`, i.e., `p [lastx,firsty] == True`

Longest Satisfying Segment Problem: Exercise

The task is to implement this operator by filling in the blanks

```
let max (x:i32, y:i32) = i32.max x y

let redOp (pred2 : i32 -> i32 -> bool)
          (lssx: i32, lisx: i32, lcsx: i32, tlx: i32, firstx: i32, lastx: i32)
          (lssy: i32, lisy: i32, lcsy: i32, tly: i32, firsty: i32, lasty: i32)
        : (i32,i32,i32,i32,i32,i32) =

  let connect= ...
  let newlss = ...
  let newlis = ...
  let newlcs = ...
  let newtl  = ...
  let first  = if tlx == 0 then firsty else firstx
  let last   = if tly == 0 then lastx else lasty in
  (newlss, newlis, newlcs, newtl, first, last)

let mapOp (pred1 : i32 -> bool) (x: i32) : (i32,i32,i32,i32,i32,i32) =
  let xmatch = if pred1 x then 1 else 0 in
  (xmatch, xmatch, xmatch, 1, x, x)

let lssp (pred1 : i32 -> bool)
         (pred2 : i32 -> i32 -> bool)
         (xs    : []i32) : i32 =
  let (x,_,_,_,_,_) =
        map (mapOp pred1) xs
     |> reduce (redOp pred2) (0,0,0,0,0,0)
  in  x
```

Conclusion

What have we talked about today?

- Hardware Parallelism:

Conclusion

What have we talked about today?

- Hardware Parallelism:
  the only way of scalably increasing the compute power.
- Big Challenge:

Conclusion

What have we talked about today?

- Hardware Parallelism:
  the only way of scalably increasing the compute power.
- Big Challenge:having parallel commodity software.
- List-Homomorphism:
  a way of reasoning about parallelism and
  of building inherently parallel programs.