CSCI 564 Advanced Computer Architecture

Lecture 13: Multiprocessors

Dr. Bo Wu

April 23, 2021

Colorado School of Mines

Why Parallel Computers?

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

- Main Goal
  - Improve performance (Execution time or task throughput)
    - Execution time of a program governed by Amdahl's Law

- Other Goals
  - Reduce power consumption
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - Improve cost efficiency and scalability, reduce complexity
    - Harder to design a single unit that performs as well as N simpler units

Types of Parallelism and How to Exploit Them

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution
  - Dataflow
- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - GPUs
- Task Level Parallelism
  - Different “tasks/threads” can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: compiler vectorization
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task

Multiprocessing Fundamentals

Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

Main Issues in Tightly-Coupled MP

- Shared memory synchronization
  - Locks, atomic operations

- Cache consistency
  - More commonly called cache coherence

- Ordering of memory operations
  - What should the programmer expect the hardware to provide?

- Resource sharing, contention, partitioning
- Communication: Interconnection networks
- Load imbalance

Parallel Speedup Example

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

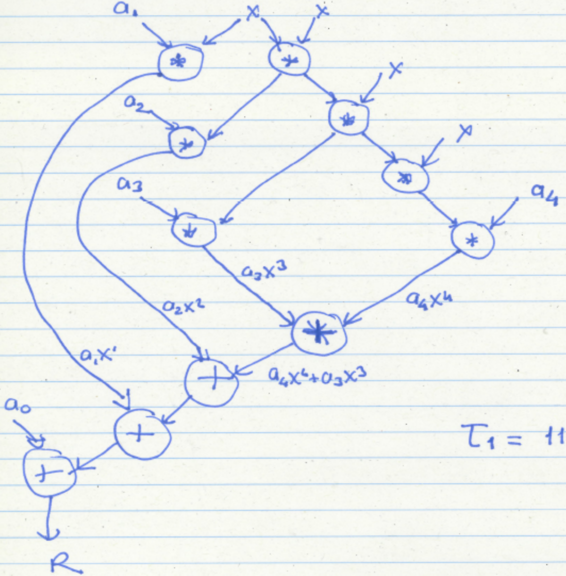
- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions

- How fast is this with 3 processors?

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

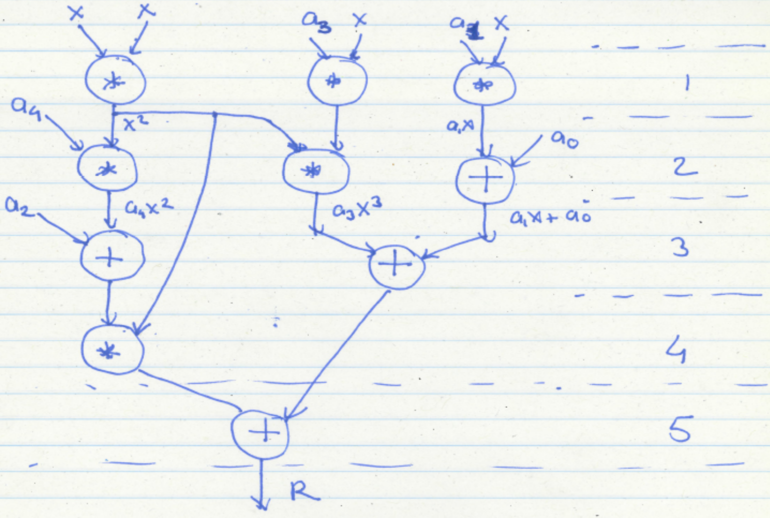
Single processor : 11 operations (DRAW the data flow graph)



$\tau_1 = 11$ cycles

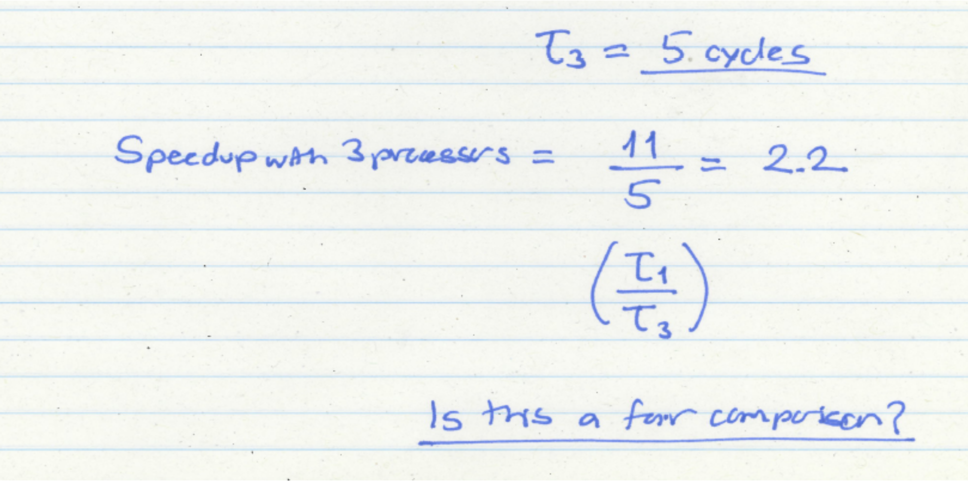
$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

Three processors: τ_3 (exec. time with 3 proc.)



τ_3 = 5 cycles

Speedup with 3 Processors



$\tau_3 = \underline{5 \text{ cycles}}$

Speedup with 3 processors $= \frac{11}{5} = 2.2$

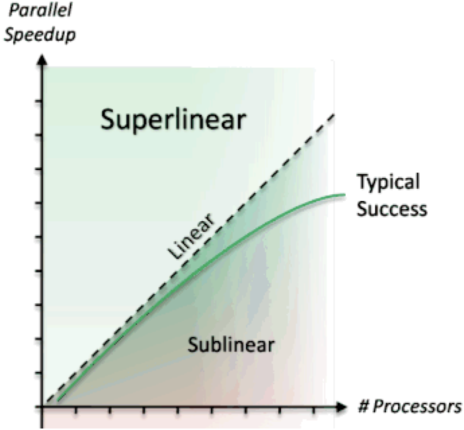
$$\left(\frac{\tau_1}{\tau_3}\right)$$

Is this a fair comparison?

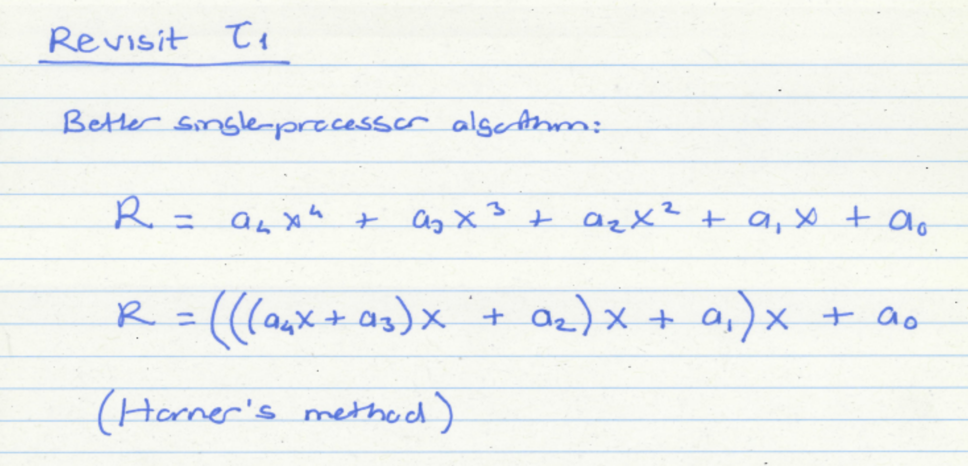
Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Cache effects
- Working set effects



Revisiting the Single-Processor



Revisit τ_1

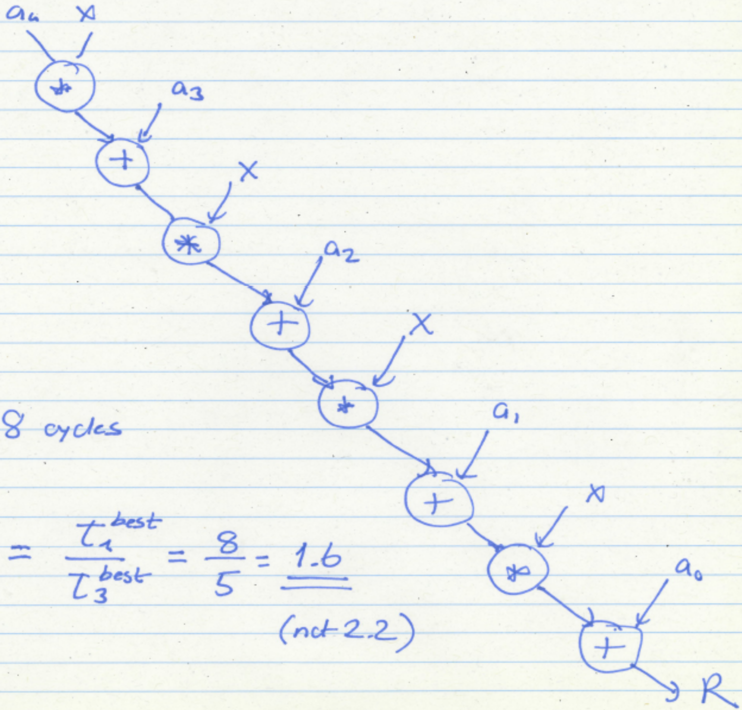
Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

(Horner's method)

Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.



$\tau_1 = 8$ cycles

Speedup with 3 procs. $= \dfrac{\tau_1^{best}}{\tau_3^{best}} = \dfrac{8}{5} = \underline{\underline{1.6}}$

(not 2.2)

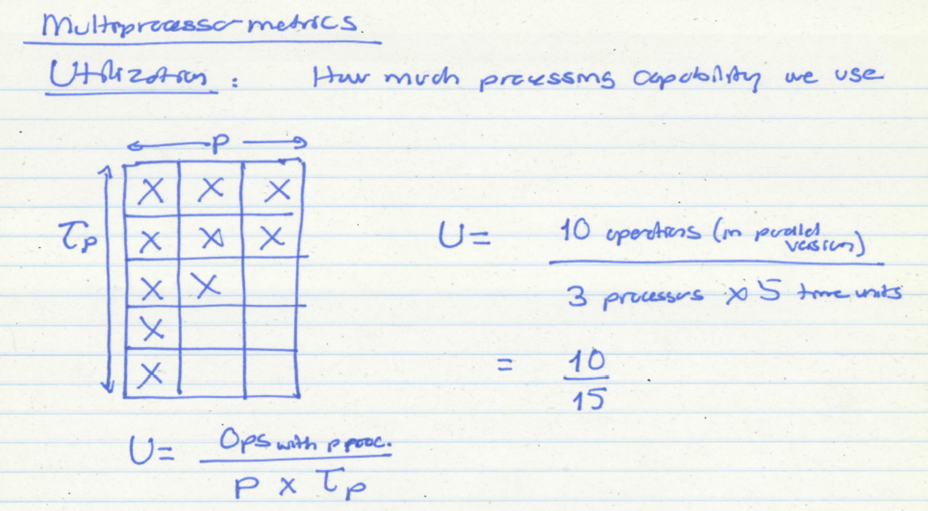
Utilization, Redundancy, Efficiency

- Utilization: How much processing capability is used
  - U = (# Operations in parallel version) / (processors x Time)

- Redundancy: how much extra work is done with parallel processing
  - R = (# of operations in parallel version) / (# operations in best single processor algorithm version)

- Efficiency
  - E = (Time with 1 processor) / (processors x Time with P processors)
  - E = U/R

Utilization of a Multiprocessor



Multiprocessor metrics

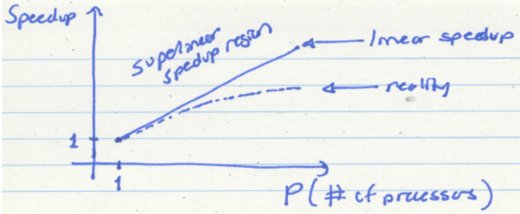
Utilization : How much processing capability we use

$$\begin{array}{c|ccc} & \longleftarrow & p & \longrightarrow \\ \hline & \times & \times & \times \\ & \times & \times & \times \\ \tau_p & \times & \times & \\ & \times & & \\ & \times & & \end{array}$$

$$U = \frac{\text{10 operations (in parallel version)}}{\text{3 processors} \times \text{5 time units}} = \frac{10}{15}$$

$$U = \frac{\text{Ops with } p \text{ proc.}}{p \times \tau_p}$$

Caveats of Parallelism (I)



Why the reality? (diminishing returns)

$$\tau_p = \underbrace{\alpha \cdot \frac{\tau_1}{p}}_{\substack{\text{parallelizable part/fraction}\\ \text{of the single-processor}\\ \text{program}}} + \underbrace{(1-\alpha)\cdot\tau_1}_{\text{non-parallelizable part}}$$

Redundancy: How much extra work due to multiprocessing

$$R = \frac{\text{Ops with } p \text{ proc.}^{\text{best}}}{\text{Ops with } 1 \text{ proc.}^{\text{best}}} = \frac{10}{8}$$

R is always $\geqslant 1$

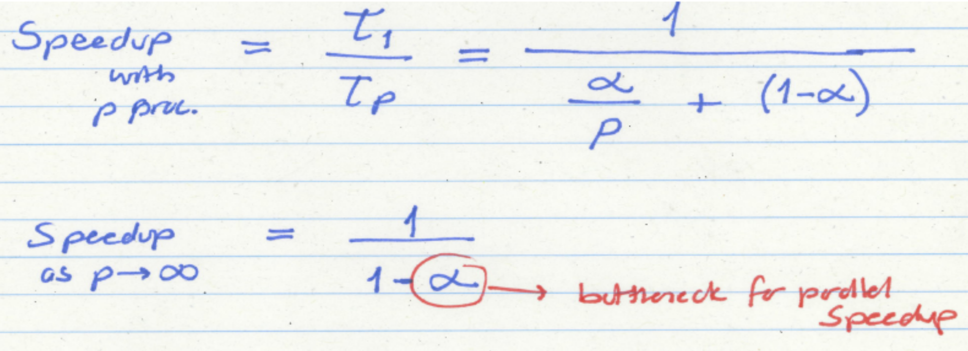
Efficiency: How much resource we use compared to how much resource we can get away with

$$E = \frac{1 \cdot T_1^{\text{best}}}{p \cdot T_p^{\text{best}}}$$ (tying up 1 proc for T_p time units)

(tying up p proc. for T_p time units)

$$= \frac{8}{15} \qquad \left(E = \frac{U}{R} \right)$$

Amdahl's Law

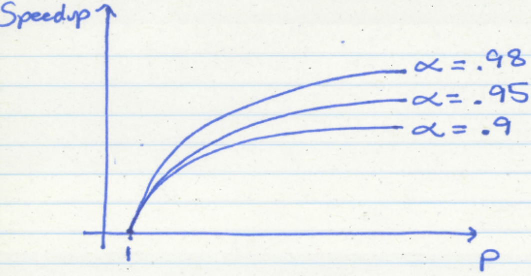


$$\text{Speedup with } p \text{ proc.} = \frac{t_1}{T_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

$$\text{Speedup as } p \rightarrow \infty = \frac{1}{1-\boxed{\alpha}} \longrightarrow \text{bottleneck for parallel speedup}$$

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

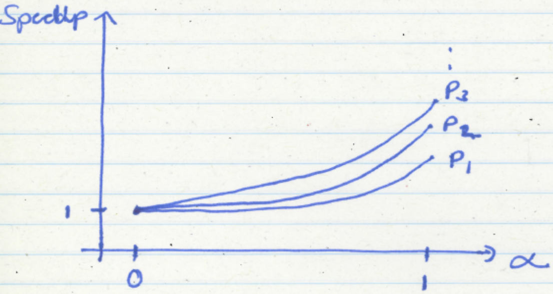
Amdahl's Law Implication 1



Amdahl's Law illustrated

Adding more and more processors gives less & less benefit if $\alpha < 1$

Amdahl's Law Implication 2



The benefit (speedup) is small until $\alpha \approx 1$

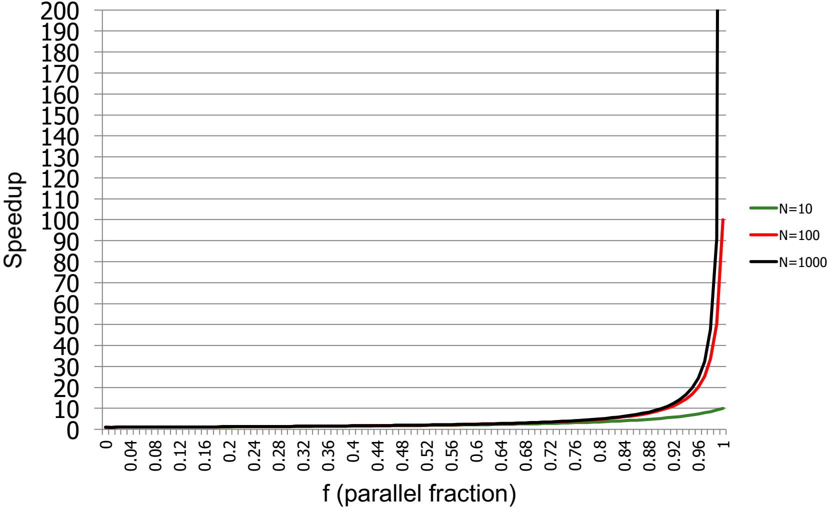
Caveats of Parallelism

- Amdahl’s Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

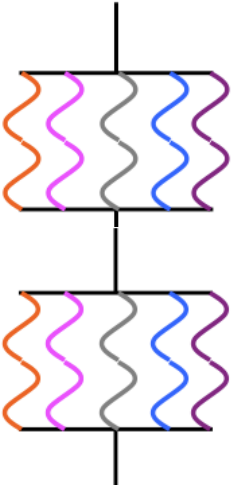
$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

  - Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.
- Maximum speedup limited by serial portion: Serial bottleneck
- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among N processors)

Sequential Bottleneck



Why the Sequential Bottleneck?

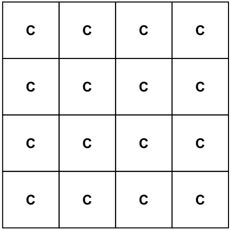


- Parallel machines have the sequential bottleneck
- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

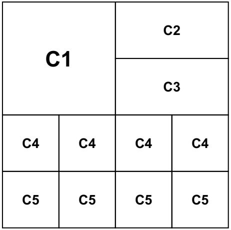
```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- Single thread prepares data and spawns parallel tasks (usually sequential)

Asymmetry Enables Customization



Symmetric



Asymmetric

- Symmetric: One size fits all
  - Energy and performance suboptimal for different “workload” behaviors
- Asymmetric: Enables customization and adaptation
  - Processing requirements vary across workloads (applications and phases)
  - Execute code on best-fit resources (minimal energy, adequate perf.)

Aside: Examples from Life

- Heterogeneity is abundant in life
  - both in nature and human-made components

- Humans are heterogeneous
- Cells are heterogeneous → specialized for different tasks
- Organs are heterogeneous
- Cars are heterogeneous
- Buildings are heterogeneous
- Rooms are heterogeneous
- …

General-Purpose vs. Special-Purpose

- Asymmetry is a way of enabling specialization

- It bridges the gap between purely general purpose and purely special purpose
  - Purely general purpose: Single design for every workload or metric
  - Purely special purpose: Single design per workload or metric
  - Asymmetric: Multiple sub-designs optimized for sets of workloads/metrics and glued together

- The goal of a good asymmetric design is to get the best of both general purpose and special purpose

Asymmetry Advantages and Disadvantages

- Advantages over Symmetric Design

  + Can enable optimization of multiple metrics

  + Can enable better adaptation to workload behavior

  + Can provide special-purpose benefits with general-purpose usability/flexibility

- Disadvantages over Symmetric Design

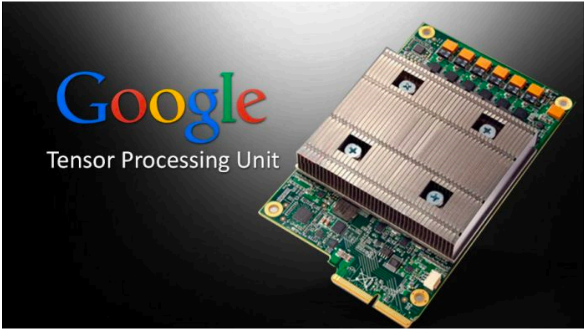
  - Higher overhead and more complexity in design, verification

  - Higher overhead in management: scheduling onto asymmetric components

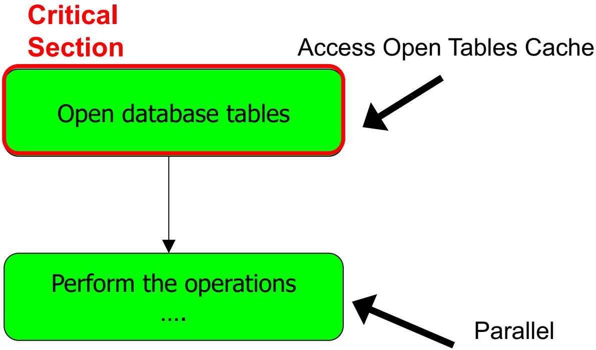
  - Overhead in switching between multiple components can lead to degradation

Yet Another Example

- Modern processors integrate general purpose cores and GPUs
  - CPU-GPU systems
  - Heterogeneity in execution models



Example from MySQL



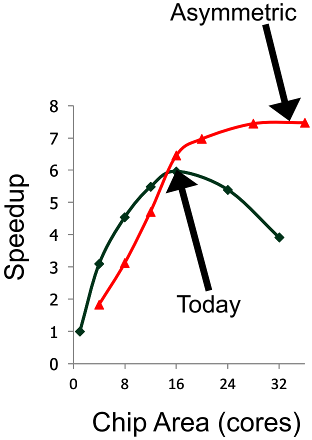
Critical Section

Access Open Tables Cache

Open database tables

Perform the operations ....

Parallel



Demands in Different Code Sections

- What we want:

- In a serialized code section → one powerful “large” core

- In a parallel code section → many wimpy “small” cores

- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core

“Large” vs. “Small” Cores

Large Core

- *Out-of-order*
- *Wide fetch e.g. 4-wide*
- *Deeper pipeline*
- *Aggressive branch predictor (e.g. hybrid)*
- *Multiple functional units*
- *Trace cache*
- *Memory dependence*

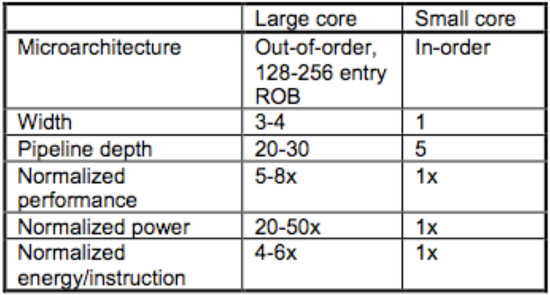
Small Core

- *In-order*
- *Narrow Fetch e.g. 2-wide*
- *Shallow pipeline*
- *Simple branch predictor (e.g. Gshare)*
- *Few functional units*

Large Cores are power inefficient:
e.g., 2x performance for 4x area (power)

Large vs. Small Cores

- Grochowski et al., “Best of both Latency and Throughput,” ICCD 2004.



	Large core	Small core
Microarchitecture	Out-of-order, 128-256 entry ROB	In-order
Width	3-4	1
Pipeline depth	20-30	5
Normalized performance	5-8x	1x
Normalized power	20-50x	1x
Normalized energy/instruction	4-6x	1x

Remember the Demands

- What we want:

- In a serialized code section → one powerful "large" core

- In a parallel code section → many wimpy "small" cores

- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core

- Can we get the best of both worlds?

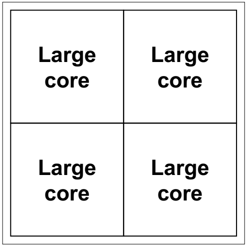
Performance vs. Parallelism

Assumptions:

1. *Small cores takes an area budget of 1 and has performance of 1*

2. *Large core takes an area budget of 4 and has performance of 2*

Tile-Large Approach



Large core

Large core

Large core

Large core

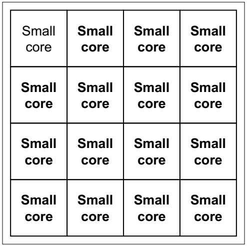
“Tile-Large”

- Tile a few large cores
- IBM Power 5, AMD Barcelona, Intel Core2Quad, Intel Nehalem

+ High performance on single thread, serial code sections (2 units)

- Low throughput on parallel program portions (8 units)

Tile-Small Approach



Small core	**Small core**	**Small core**	**Small core**
Small core	**Small core**	**Small core**	**Small core**
Small core	**Small core**	**Small core**	**Small core**
Small core	**Small core**	**Small core**	**Small core**

"Tile-Small"

- Tile many small cores
- Sun Niagara, Intel Larrabee, Tilera TILE (tile ultra-small)

+ High throughput on the parallel part (16 units)

- Low performance on the serial part, single thread (1 unit)

Can we get the best of both worlds?

- Tile Large

  + High performance on single thread, serial code sections (2 units)

  - Low throughput on parallel program portions (8 units)

- Tile Small

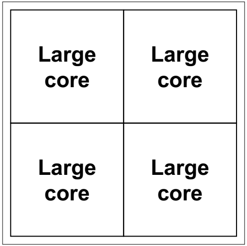
  + High throughput on the parallel part (16 units)

  - Low performance on the serial part, single thread (1 unit), reduced single-thread performance compared to existing single thread processors

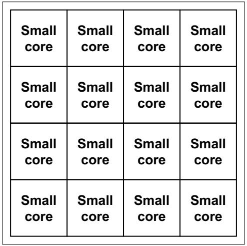
- Idea: Have both large and small on the same chip → Performance asymmetry

Asymmetric Multi-Core

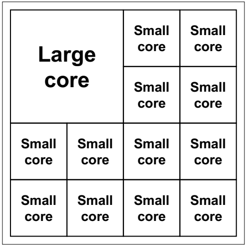
Asymmetric Chip Multiprocessor (ACMP)



"Tile-Large"



"Tile-Small"



ACMP

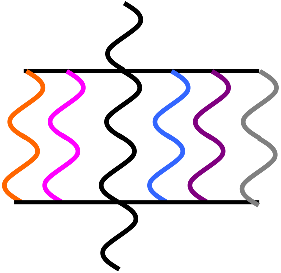
- Provide one large core and many small cores

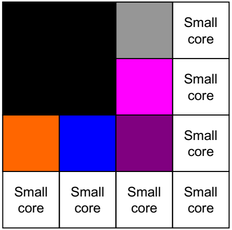
+ Accelerate serial part using the large core (2 units)

+ Execute parallel part on small cores and large core for high throughput (12+2 units)

Accelerating Serial Bottlenecks

Single thread → Large core





Small core

Small core

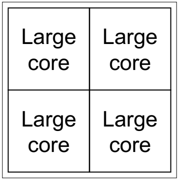
Small core

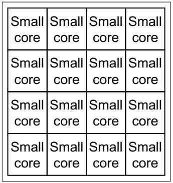
Small core Small core Small core Small core

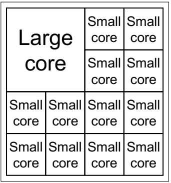
ACMP Approach

ACMP Performance vs. Parallelism

Area-budget = 16 small cores







	"Tile-Large"	"Tile-Small"	ACMP
Large Cores	4	0	1
Small Cores	0	16	12
Serial Performance	2	1	2
Parallel Throughput	2 x 4 = 8	1 x 16 = 16	1x2 + 1x12 = 14

Amdahl's Law Modified

- Simplified Amdahl's Law for an Asymmetric Multiprocessor
- Assumptions:
  - Serial portion executed on the large core
  - Parallel portion executed on both small cores and large cores
  - f: Parallelizable fraction of a program
  - L: Number of large processors
  - S: Number of small processors
  - X: Speedup of a large processor over a small one

$$\text{Speedup} = \frac{1}{\frac{1 - f}{X} + \frac{f}{S + X*L}}$$

Accelerating Parallel Bottlenecks

- Serialized or imbalanced execution in the parallel portion can also benefit from a large core

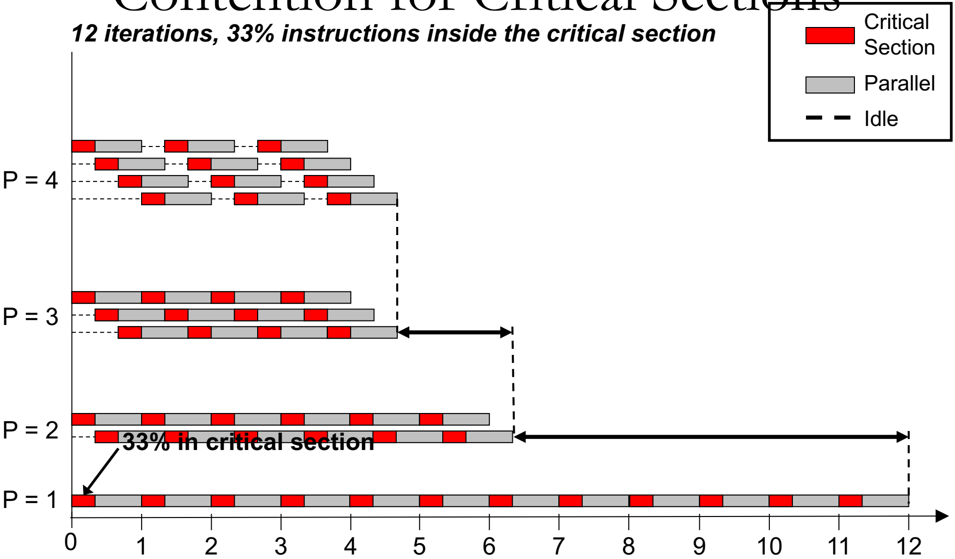
- Examples:
  - Critical sections that are contended
  - Parallel stages that take longer than others to execute

- Idea: Dynamically identify these code portions that cause serialization and execute them on a large core

Accelerated Critical Sections

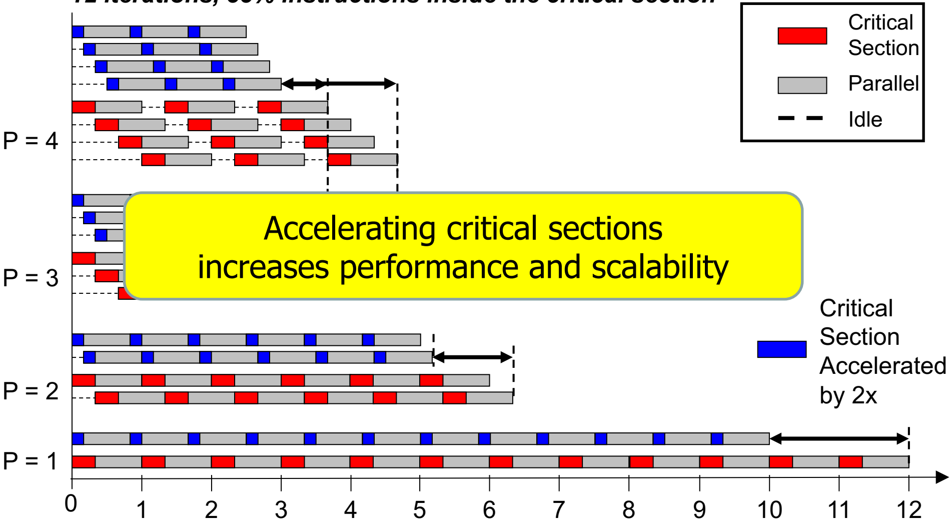
M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt,
"Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures"
*Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**), 2009*

Contention for Critical Sections



Contention for Critical Sections

12 iterations, 33% instructions inside the critical section



Accelerating critical sections increases performance and scalability

An Example: Accelerated Critical Sections

- Idea: HW/SW ships critical sections to a large, powerful core in an asymmetric multi-core architecture

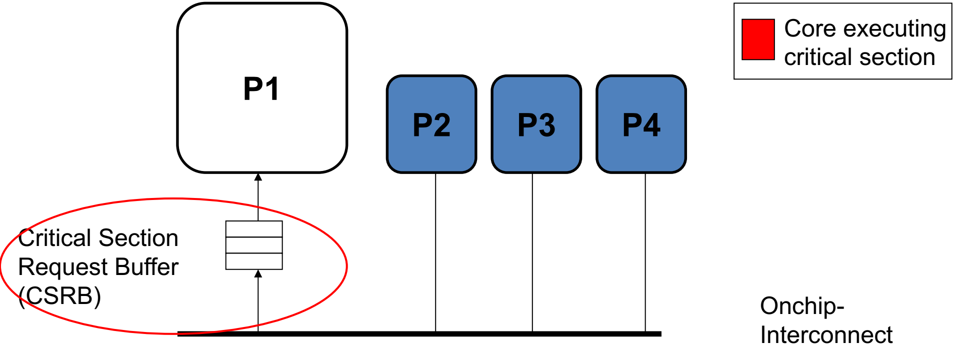
- Benefit:
  - Reduces serialization due to contended locks
  - Reduces the performance impact of hard-to-parallelize sections
  - Programmer does not need to (heavily) optimize parallel code → fewer bugs, improved productivity

- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.
- Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010, IEEE Micro Top Picks 2011.

Accelerated Critical Sections

```
EnterCS()
        PriorityQ.insert(…)
LeaveCS()
```

1. **P2 encounters a critical section (CSCALL)**
2. **P2 sends CSCALL Request to CSRB**
3. **P1 executes Critical Section**
4. **P1 sends CSDONE signal**


P1
P2
P3
P4
🟥 Core executing critical section
Critical Section Request Buffer (CSRB)
Onchip-Interconnect

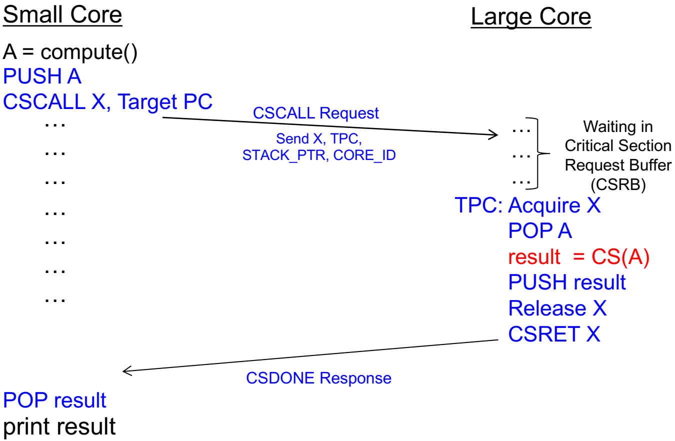
Accelerated Critical Sections (ACS)

Small Core

A = compute()

LOCK X
    result = CS(A)
UNLOCK X

print result



Small Core

A = compute()
PUSH A
CSCALL X, Target PC
…
…
…
…
…
…
…

CSCALL Request

Send X, TPC,
STACK_PTR, CORE_ID

Large Core

…
…
…

Waiting in
Critical Section
Request Buffer
(CSRB)

TPC: Acquire X
POP A
result = CS(A)
PUSH result
Release X
CSRET X

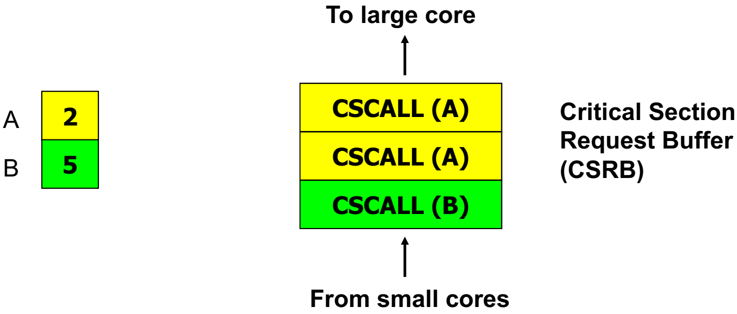
CSDONE Response

POP result
print result

- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.

False Serialization

- ACS can serialize independent critical sections
- Selective Acceleration of Critical Sections (SEL)
  - Saturating counters to track false serialization



A	2
B	5

To large core

CSCALL (A)

CSCALL (A)

CSCALL (B)

From small cores

Critical Section Request Buffer (CSRB)