Technical University of Munich

Lecture 2: Parallel computing

Informatik elective: GPU Computing

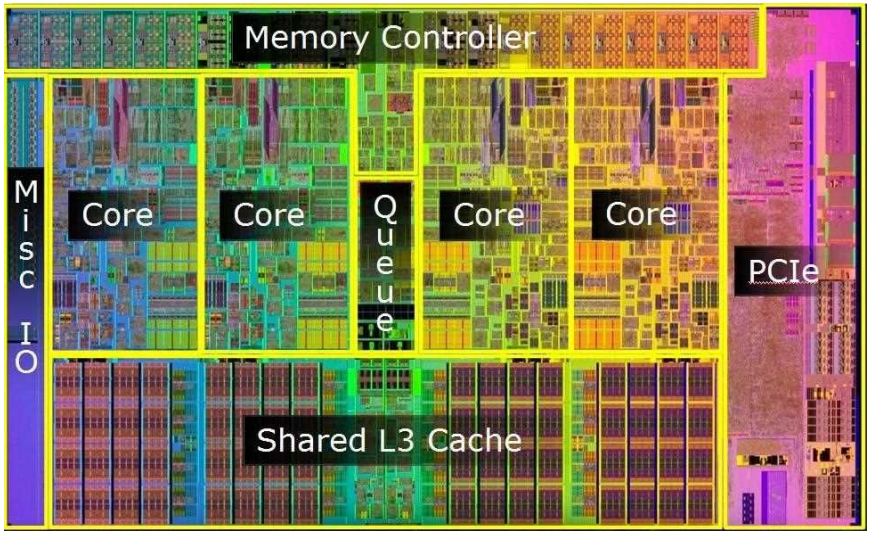
Pratik Nayak

In this session

- Execution strategies
- Task graphs: critical path, concurrency, task decompositions
- Estimating work, memory and cache complexities
- NUMA and cache effects
- Estimating performance
- Roofline model, arithmetic intensity, machine balance
- Scaling: Strong and weak.

Recall terminology: Thread and Core

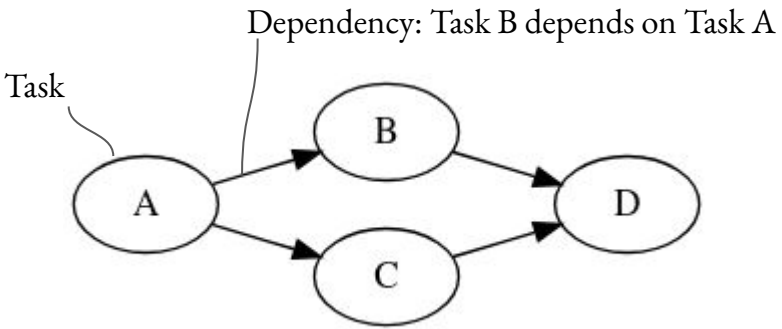
- *Thread (software-level)*: "thread of execution": an ordered sequence of instructions (software)
- *Core (hardware-level)*: One processor within a CPU die (hardware).
- *Multi-core (hardware-level)*: Multiple processors capable of independent execution within one CPU die.



[Intel multi-core CPU architecture, Intel Corp]

Basic definitions

- *Task*: Unit of execution or a unit of work
- *Dependency*: Requirement that enforces a partial ordering of tasks
- *Computing Resource:* A processing unit on which a task can be executed.



A task graph

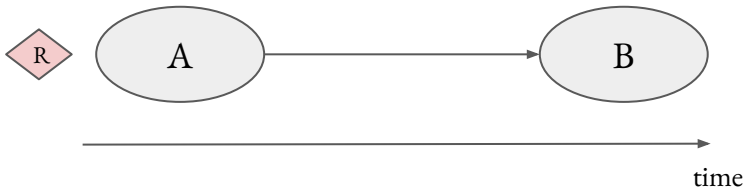
Execution strategies

Two tasks, A and B need to be completed.	
Questions:	1. What are the dependencies ? 2. What execution resources are available ?

Execution strategies

Two tasks, A and B need to be completed.	
Questions:	1. What are the dependencies ? 2. What execution resources are available ?

- *Serial execution:* Execute task A and then execute task B



➔ If B depends on A

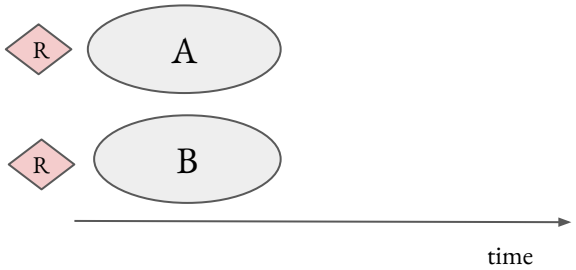
OR

➔ If only one compute resource is available

Execution strategies

Two tasks, A and B need to be completed.	
Questions:	1. What are the dependencies ? 2. What execution resources are available ?

- *Parallel execution:* Execute A and B at the same time on separate resources



➔ A and B are independent

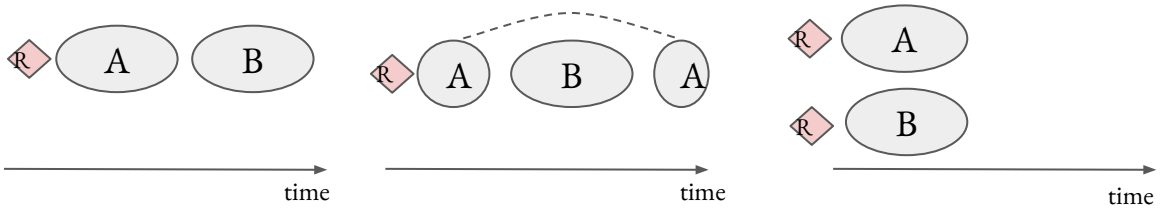
AND

➔ Multiple resources are available

Execution strategies

Two tasks, A and B need to be completed.	
Questions:	1. What are the dependencies ? 2. What execution resources are available ?

- *Concurrent execution:* Execute A and execute B at the same time (no specification on resources)



➔ A and B are independent

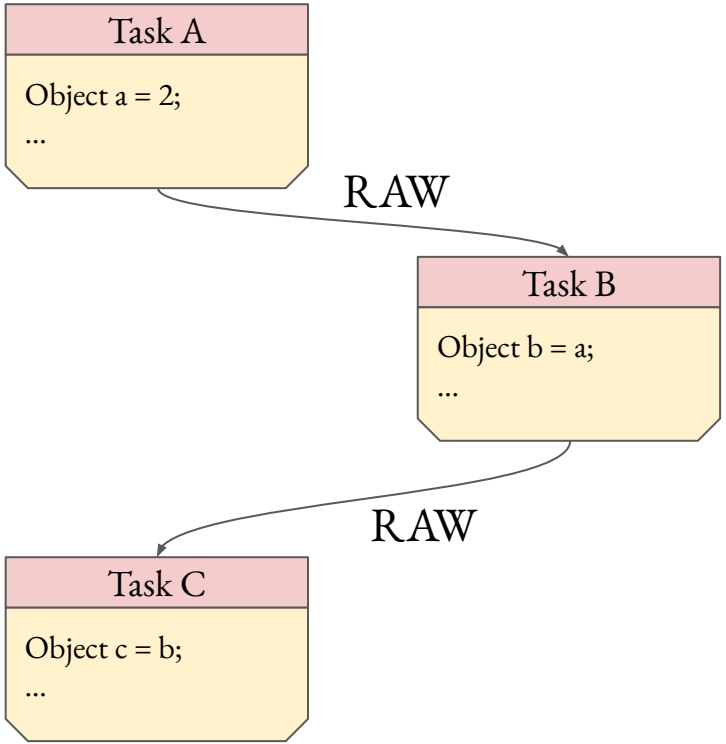
Execution strategies

Two tasks, A and B need to be completed.	
Questions:	1. What are the dependencies ? 2. What execution resources are available ?

- *Serial execution:* Execute task A and then execute task B
- *Concurrent execution:* Execute A and execute B at the same time (no restriction on resources)
- *Parallel execution:* Execute A and B at the same time on separate resources

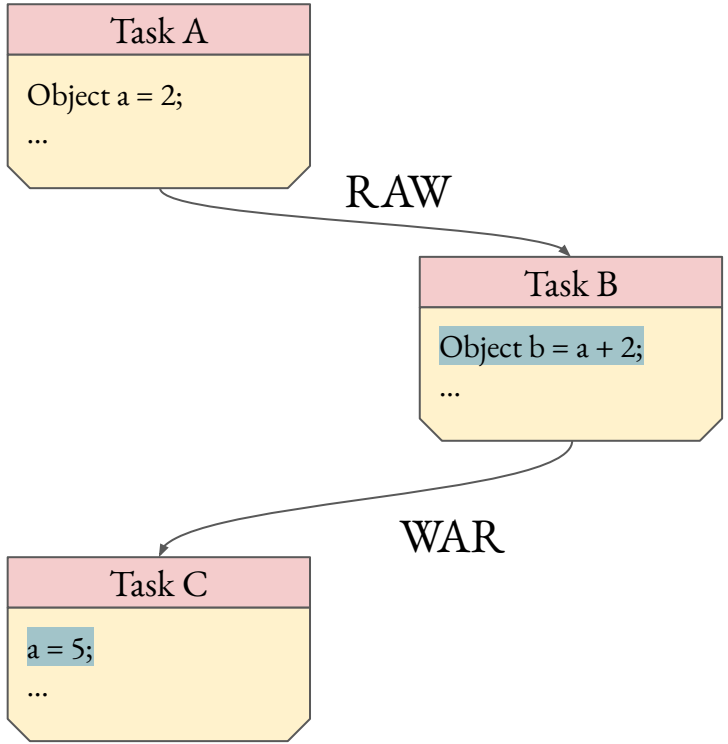
Data dependencies

- (*RAW*) Read-after-write dependency: True dependency



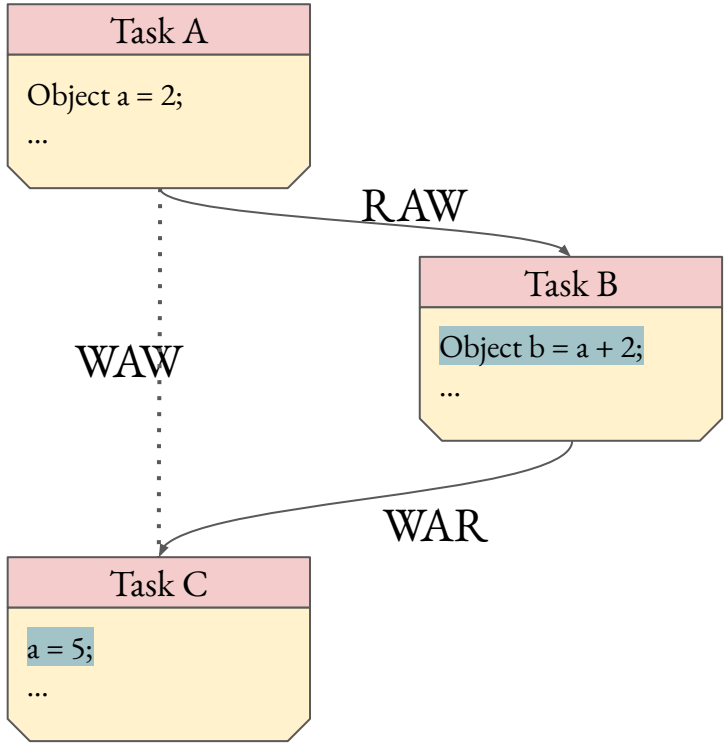
Data dependencies

- (*RAW*) Read-after-write dependency: True dependency
- (*WAR*) Write-after-read dependency: Anti-dependency



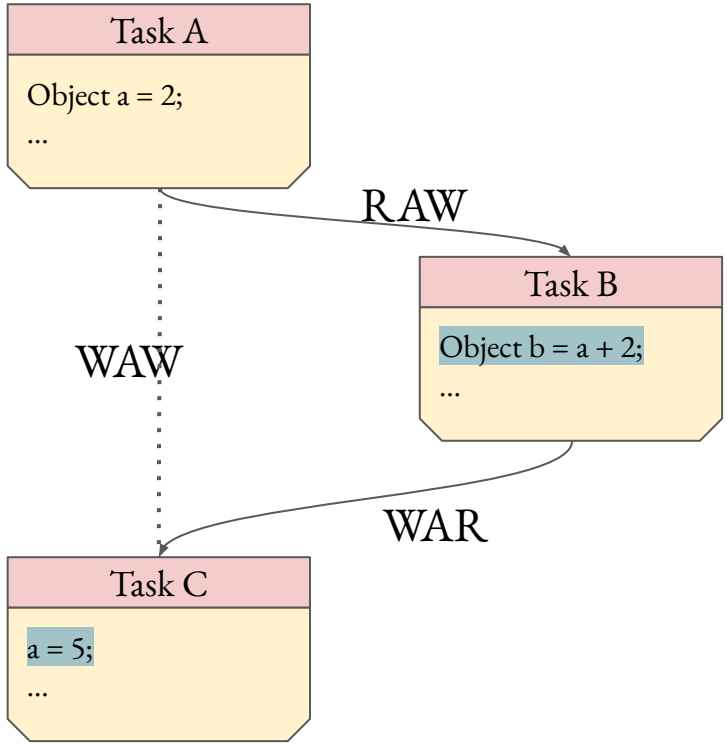
Data dependencies

- (*RAW*) Read-after-write dependency: True dependency
- (*WAR*) Write-after-read dependency: Anti-dependency
- (*WAW*) Write-after-write dependency: Output dependency



Data dependencies

- (*RAW*) Read-after-write dependency: True dependency
- (*WAR*) Write-after-read dependency: Anti-dependency
- (*WAW*) Write-after-write dependency: Output dependency

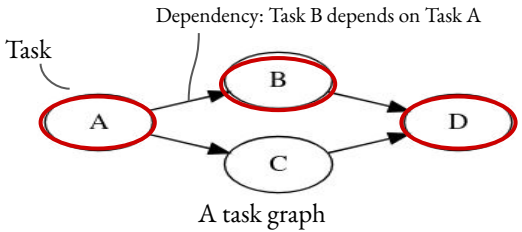


Race conditions and deadlocks

- Race condition: Undefined behaviour based on different thread execution order
  - Data races:
    - Two threads writing to same location (write-write conflict)
    - One thread reads, while other writes to same location (read-write conflict)
  - Incorrect interleaving:
    - Intermediate state exposed to other threads
- Deadlocks:
  - A dependency cycle between concurrent threads:
  - Thread t_1 depends on t_2 and t_2 depends on t_1. No forward progress.

Decomposition and tasking

- Given some computation, to enable parallel processing, we need to:
  - Decompose the computations into smaller tasks
  - Assign the tasks to available resources.
- Decomposition into tasks is generally represented in terms of a task-dependency graph
  - A directed graph with nodes representing the tasks, and edges representing the dependencies.
  - Directed graph → Represents a sequence of tasks
  - Critical path: Longest directed path between a start node and a finish node of a task dependency graph



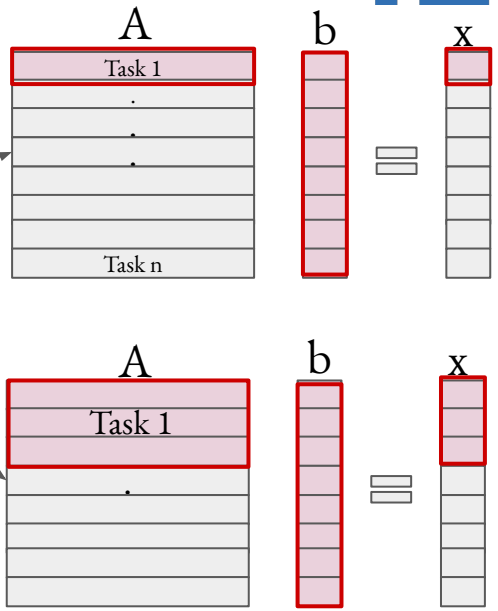
A task graph

Critical path: A→ B → D

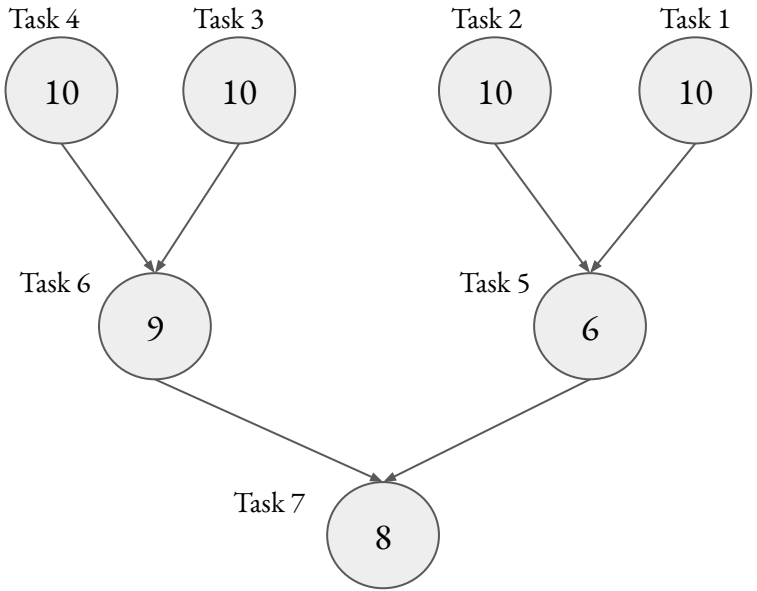
Decomposition and tasking

- Choice of decompositions → Task granularity (Gr):
  - Large number of small tasks (fine-grained decomposition)
  - Small number of large tasks (coarse-grained decomposition)
- Degree of concurrency (#C) → Number of tasks that can execute in parallel
  - Maximum #C → Largest number of tasks at any point of execution
  - Average #C → Average number of tasks that can be executed concurrently → total work/critical path length

$$\#C \propto \frac{1}{Gr}$$

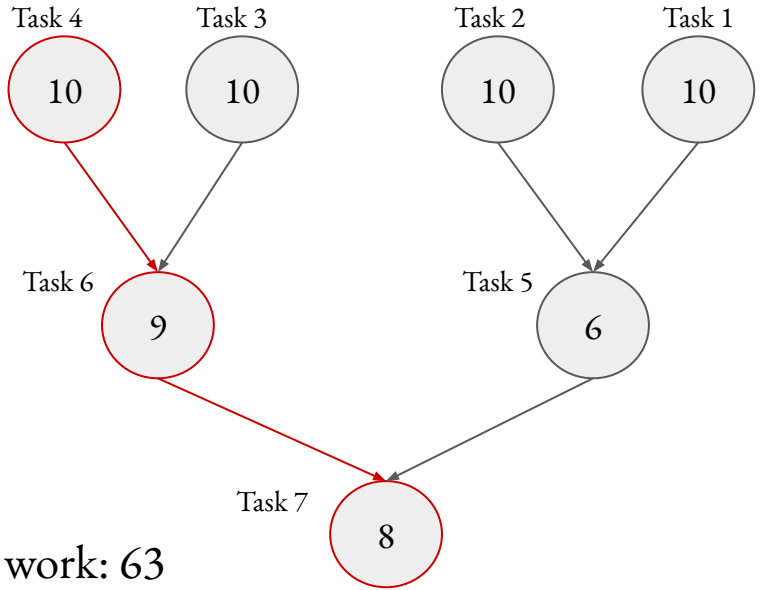


Critical path: Examples



[Introduction to Parallel Computing, Grama, Gupta, Karypis]

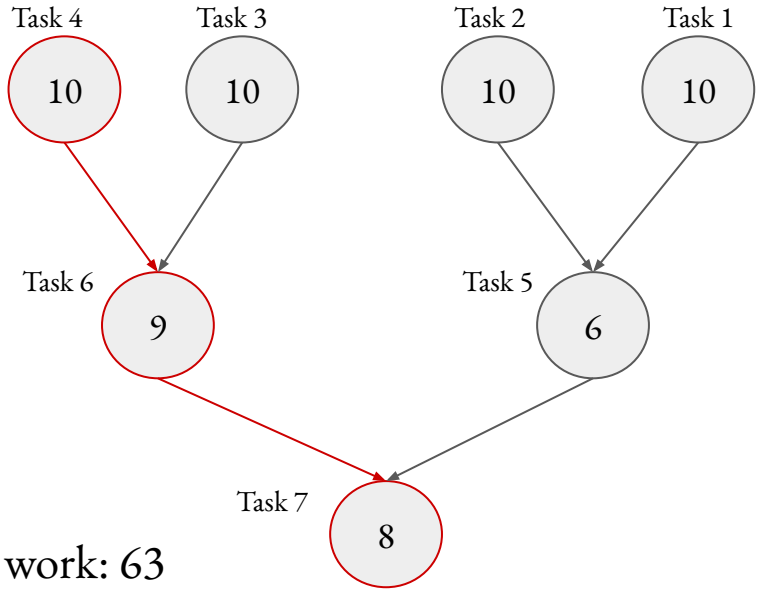
Critical path: Examples



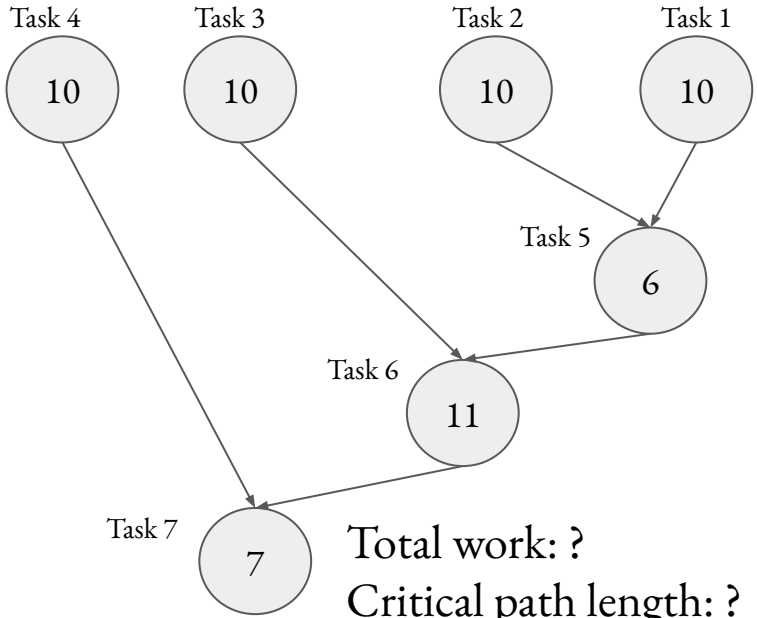
Total work: 63
Critical path length: 27
Average concurrency ~ 2.3

[Introduction to Parallel Computing, Grama, Gupta, Karypis]

Critical path: Examples



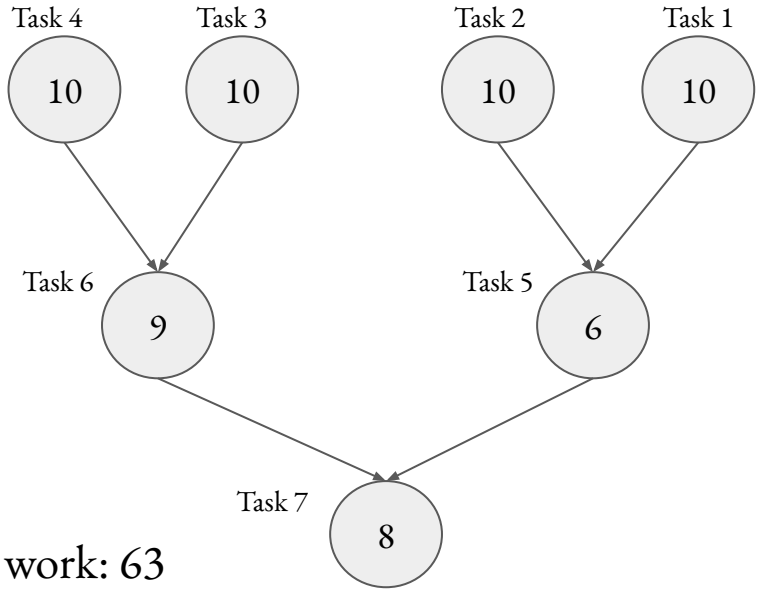
Total work: 63
Critical path length: 27
Average concurrency ~ 2.3



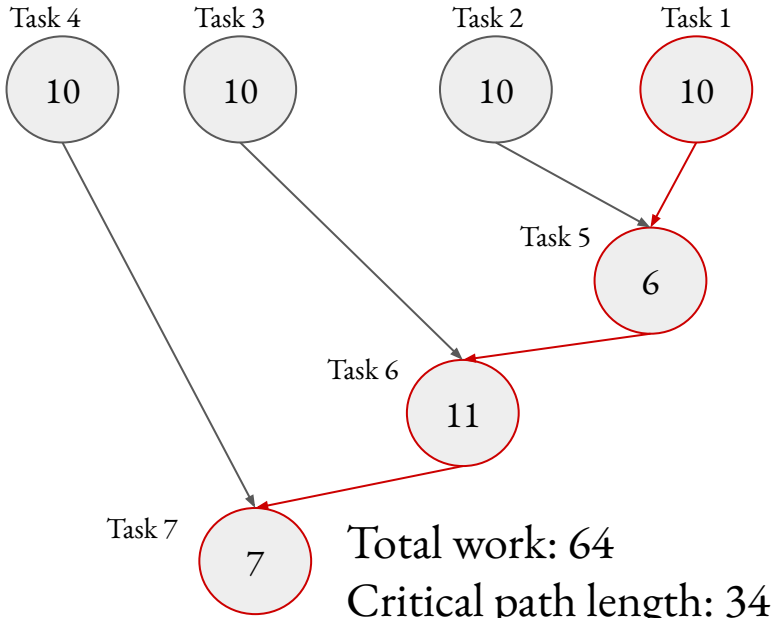
Total work: ?
Critical path length: ?
Average concurrency: ?

[Introduction to Parallel Computing, Grama, Gupta, Karypis]

Critical path: Examples



Total work: 63
Critical path length: 27
Average concurrency ~ 2.3



Total work: 64
Critical path length: 34
Average concurrency ~ 1.9

[Introduction to Parallel Computing, Grama, Gupta, Karypis]

Algorithmic complexity

- Assume T_1 to be the amount of work and p be the number of available parallel resources
- Given a critical path length (span) of T_∞, the lower bound for algorithmic complexity is given by

$$T_p \geq max\left(\frac{T_1}{p}, T_\infty\right)$$

- A greedy scheduler (no unenforced idleness), for example achieves (upper-bound)

$$T_{p,greedy} \leq \frac{T_1}{p} + T_\infty$$

[Brent 1975 and Introduction to Parallel Computing, Grama, Gupta, Karypis]

Work and memory complexities

- (Work complexity): Given some computation, parameterized by an input size n, the work complexity is defined by $W(n)$: The total number of basic operations (add, mul, divide etc) required.
  - Example (vector addition): Addition of two vectors: u and v, each of size n has a work complexity of $W(n) = n$
  - Example (dot product): Dot product of two vectors: u and v, each of size n has a work complexity of $W(n) = ?$

Work and memory complexities

- (Work complexity): Given some computation, parameterized by an input size n, the work complexity is defined by $W(n)$: The total number of basic operations (add, mul, divide etc) required.
  - Example (vector addition): Addition of two vectors: u and v, each of size n has a work complexity of $W(n) = n$
  - Example (dot product): Dot product of two vectors: u and v, each of size n has a work complexity of $W(n) = 2n$

Work and memory complexities

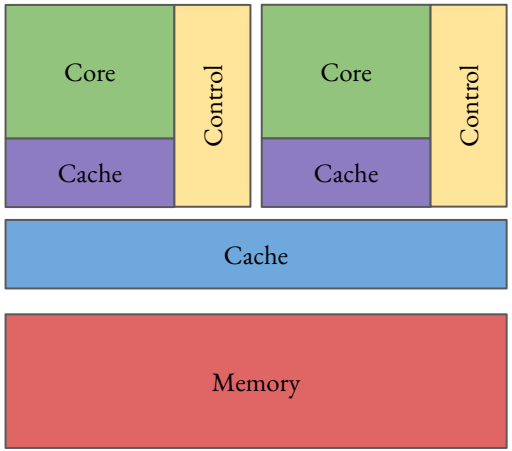
- (Work complexity): Given some computation, parameterized by an input size n, the work complexity is defined by $W(n)$: The total number of basic operations (add, mul, divide etc) required.
- (Memory complexity): Given some computation, parameterized by an input size n, the memory complexity is defined by $M(n)$: The total number of memory operations (reads and writes) required.
  - Ex (vector addition): Addition of two vectors: u and v, each of size n has a memory complexity of $M(n) = 3n$
  - Ex (dot product): Dot product of two vectors: u and v, each of size n has a memory complexity of $M(n) = ?$

Work and memory complexities

- (Work complexity): Given some computation, parameterized by an input size n, the work complexity is defined by $W(n)$: The total number of basic operations (add, mul, divide etc) required.
- (Memory complexity): Given some computation, parameterized by an input size n, the memory complexity is defined by $M(n)$: The total number of memory operations (reads and writes) required.
  - Ex (vector addition): Addition of two vectors: u and v, each of size n has a memory complexity of $M(n) = 3n$
  - Ex (dot product): Dot product of two vectors: u and v, each of size n has a memory complexity of $M(n) = 2n + 1$

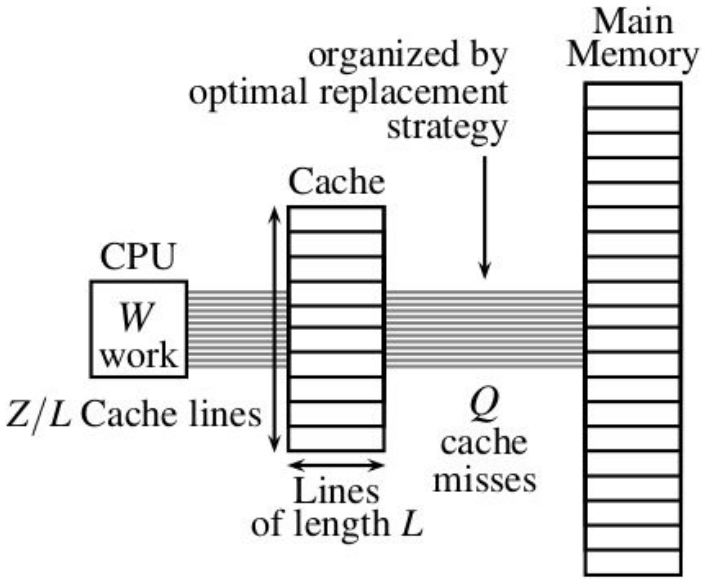
Caching

- Intermediate memory storage between main memory and compute
- Compute unit looks for data in the closest cache:
  - If data is found → cache hit. load data from cache
  - Inability to find data → cache miss, (#cache misses → Q)
  - Recursively look for data in higher-level caches
- Fetches from cache always in blocks called cache lines (L)
- Fetches are hierarchical:
  - Data must be hierarchically fetched into each level of cache, if it needs to be fetched from main memory.



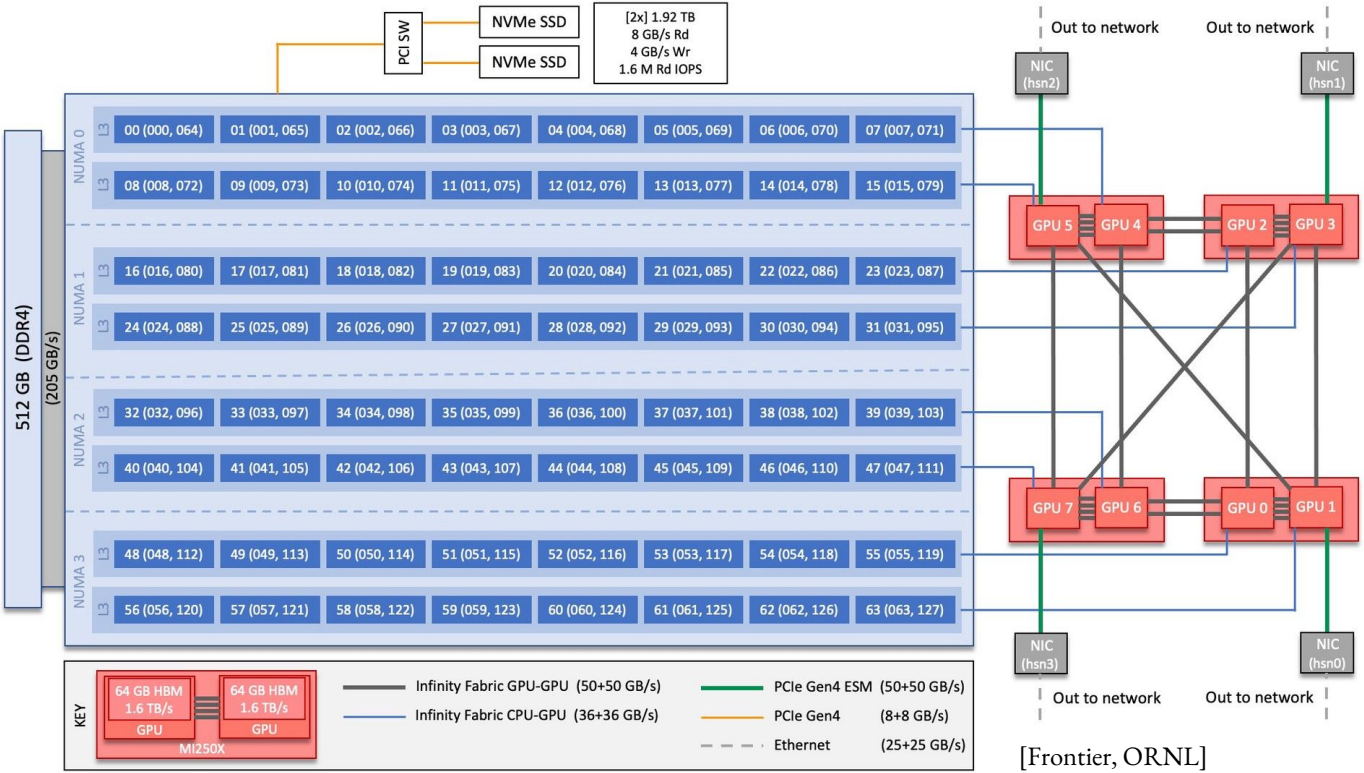
Cache complexity

- Ideal cache model:
  - L: Length of Cache line
  - $Z = \Omega(L^2)$: Cache size (tall cache)
- Cache complexity of an algorithm:
  - Input size n
  - Number of cache misses: $Q(n; Z, L)$
- Cache aware v/s cache oblivious algorithms
  - Cache aware: Q parameterized on compile-time/runtime parameters
  - Cache-oblivious: Q independent of compile-time/runtime parameters



[Frigo et.al , Cache-Oblivious Algorithms, 2012]

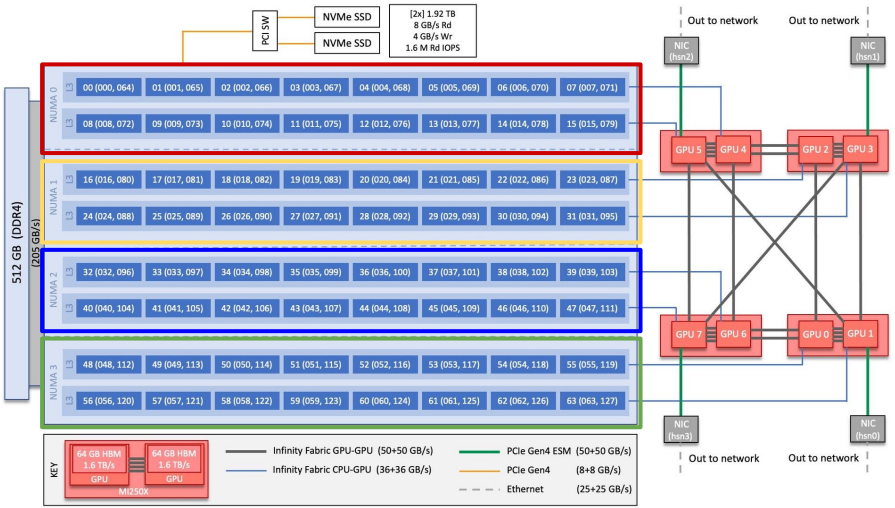
Multi-core systems and NUMA



[Frontier, ORNL]

Multi-core systems and NUMA

- 4 NUMA (Non-Uniform Memory Access) domains per node
  - 2 L3 caches per NUMA domain
- Mapping compute region to appropriate memory region crucial.
- GPUs only connected to some NUMA domains



[Frontier, ORNL]

Measuring compute performance: FLOP/s

$$FLOP/s = racks \times \frac{nodes}{rack} \times \frac{sockets}{node} \times \frac{cores}{socket} \times \frac{cycles}{second} \times \frac{FLOP}{cycle}$$

- My computer: 32 FLOP/cycle, 6 GHz, 8 cores/socket, 1 socket: 1536 GFLOP/s = 1.5 TFLOP/s

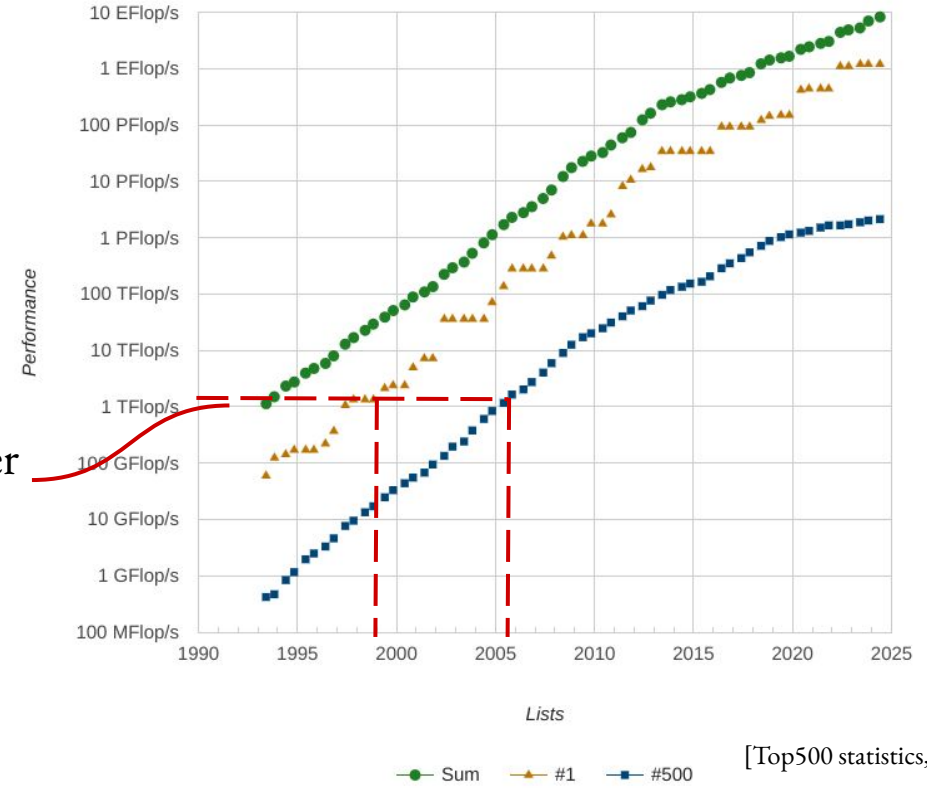
$$\text{GFLOP} \rightarrow \quad \times 10^9$$

- Frontier supercomputer: 1.7 EFLOP/s

$$\text{EFLOP} \rightarrow \quad \times 10^{18}$$

Measuring compute performance: FLOP/s

My computer



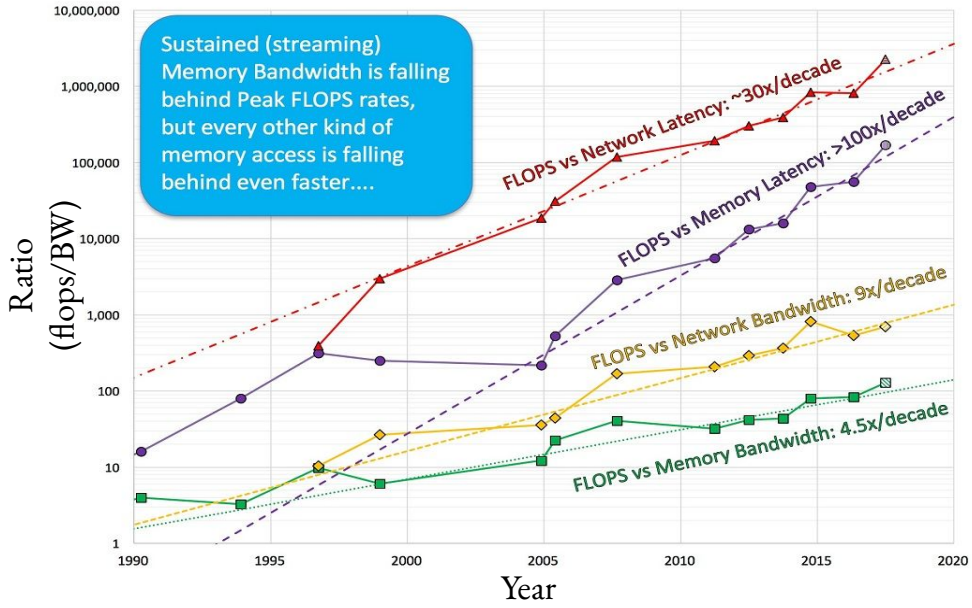
[Top500 statistics, top500.org, 2024]

Machine balance

- Machine balance

$$(\text{MB}) = \frac{\text{FLOP/s}}{\text{BW}}$$

- Much more expensive to move data than to compute on existing data
- Ratio is worsening over the years



[McCalpin , HPC system machine balance trends, 2016]

Application FLOP/s and Bandwidth

- Application performance generally measured also with achieved FLOP/s.
- Can be computed from work complexity, $W_{app}(n)$, memory complexity $M_{app}(n)$, and application runtime, $t_{app}(n)$

$$\mathcal{P}_{app}(n) = \frac{W_{app}(n)}{t_{app}(n)}$$

Achieved performance estimate

$$\mathcal{BW}_{app}(n) = \frac{M_{app}(n)}{t_{app}(n)}$$

Achieved bandwidth estimate

- Can be misleading:
  - Ignores cache effects
  - High $\mathcal{P}_{app}(n)$ does not necessarily indicate optimal algorithm. Why ?

Application FLOP/s and Bandwidth

- Application performance generally measured also with achieved FLOP/s.
- Can be computed from work complexity, $W_{app}(n)$, memory complexity $M_{app}(n)$, and application runtime, $t_{app}(n)$

$$\mathcal{P}_{app}(n) = \frac{W_{app}(n)}{t_{app}(n)}$$

Achieved performance estimate

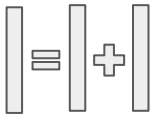
$$\mathcal{BW}_{app}(n) = \frac{M_{app}(n)}{t_{app}(n)}$$

Achieved bandwidth estimate

- Can be misleading:
  - Ignores cache effects
  - High $\mathcal{P}_{app}(n)$ does not necessarily indicate optimal algorithm. Why ? (No indication of useful FLOPs)
- Nevertheless, a good starting point to analyze performance of your application.

Arithmetic Intensity

- A rule of thumb for measuring ratio of compute intensity to memory intensity
- Example: (Vector addition): $u + v$



$$AI = \frac{FLOP\ count}{Number\ of\ byte\ accesses}$$

$$AI = \frac{n}{3n} = \frac{1}{3} = \mathcal{O}(1)$$

Arithmetic Intensity

- A rule of thumb for measuring ratio of compute intensity to memory intensity
- Example: (Vector addition): $u + v$
- Example: (Matrix-vector product): $x = Ab$

$$AI = \frac{FLOP\ count}{Number\ of\ byte\ accesses}$$

$$AI = \frac{n}{3n} = \frac{1}{3} = \mathcal{O}(1)$$

$AI =$?

Arithmetic Intensity

- A rule of thumb for measuring ratio of compute intensity to memory intensity
- Example: (Vector addition): $u + v$
- Example: (Matrix-vector product): $x = Ab$

$$AI = \frac{FLOP\ count}{Number\ of\ byte\ accesses}$$

$$AI = \frac{n}{3n} = \frac{1}{3} = \mathcal{O}(1)$$

$$AI = \frac{n^2 + n}{2n^2 + n} \cong \frac{1}{2} = \mathcal{O}(1)$$

Arithmetic Intensity

- A rule of thumb for measuring ratio of compute intensity to memory intensity
- Example: (Vector addition): $u + v$
- Example: (Matrix-vector product): $x = Ab$
- Example: (Matrix-matrix product): $C = AB$

$$AI = \frac{FLOP\ count}{Number\ of\ byte\ accesses}$$

$$AI = \frac{n}{3n} = \frac{1}{3} = \mathcal{O}(1)$$

$$AI = \frac{n^2 + n}{2n^2 + n} \cong \frac{1}{2} = \mathcal{O}(1)$$

$AI =$?

Arithmetic Intensity

- A rule of thumb for measuring ratio of compute intensity to memory intensity
- Example: (Vector addition): $u + v$
- Example: (Matrix-vector product): $x = Ab$
- Example: (Matrix-matrix product): $C = AB$

$$AI = \frac{FLOP\ count}{Number\ of\ byte\ accesses}$$

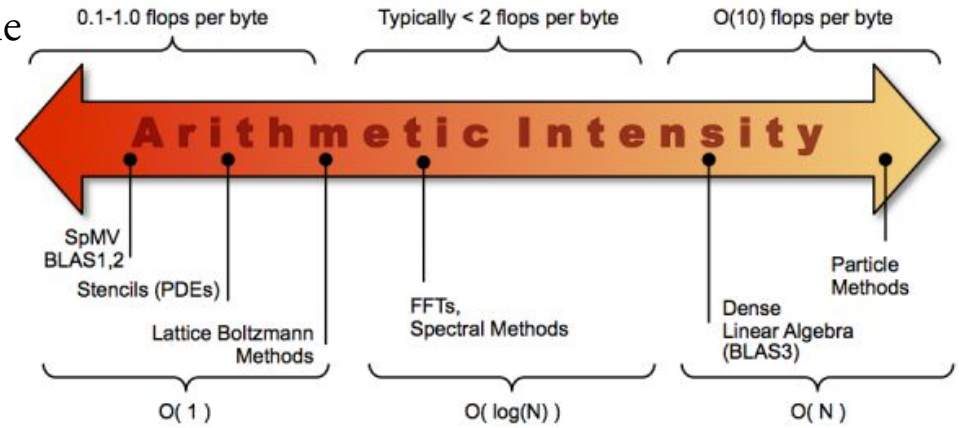
$$AI = \frac{n}{3n} = \frac{1}{3} = \mathcal{O}(1)$$

$$AI = \frac{n^2 + n}{2n^2 + n} \cong \frac{1}{2} = \mathcal{O}(1)$$

$$AI = \frac{2n^3}{3n^2} = \frac{2n}{3} = \mathcal{O}(n)$$

Arithmetic Intensity

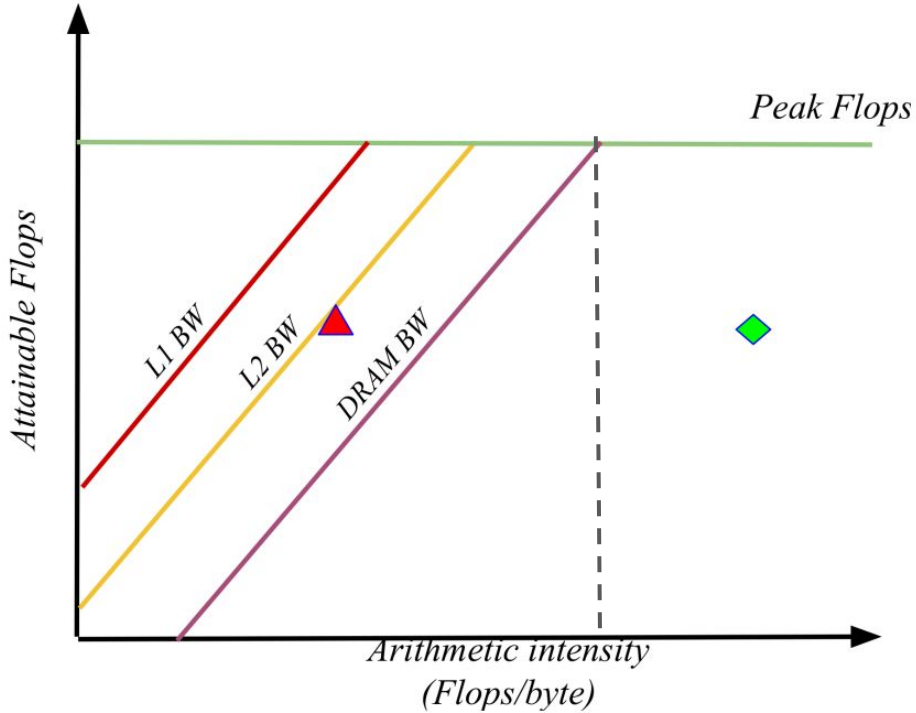
- Due to the imbalance in the machine, higher the arithmetic intensity of an operation, the better.
- Reframe algorithms into building blocks consisting of higher arithmetic intensity operations
- Particle methods, BLAS 3 operations can be very performant.



[Roofline model, crd.lbl.gov]

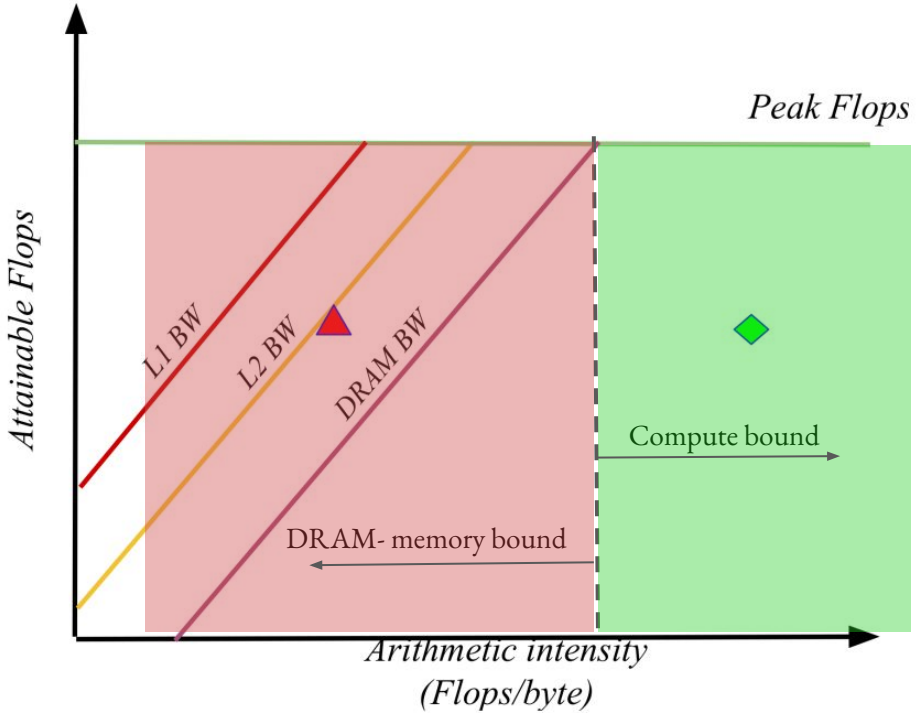
Measuring algorithm performance: Roofline

- A throughput oriented performance model
- Independent of ISA and architecture
- Components:
  - Machine parameters
  - Theoretical application bounds
  - Actual application execution
- Estimate whether an application is *memory-bound* or *compute-bound*



Measuring algorithm performance: Roofline

- A throughput oriented performance model
- Independent of ISA and architecture
- Components:
  - Machine parameters
  - Theoretical application bounds
  - Actual application execution
- Estimate whether an application is *memory-bound* or *compute-bound*

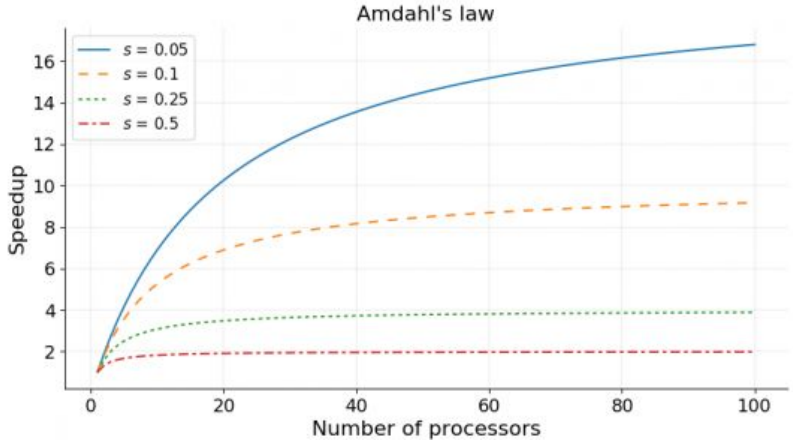


Measuring performance scaling: Strong scaling

- Speedup of some application: $S = \frac{t_1}{t_N}$
- Consider an application with p parallelizable component and s serial component, $s = 1 - p$
- Effective speedup:

$$S_{strong} = \frac{1}{s + \frac{p}{N}}$$

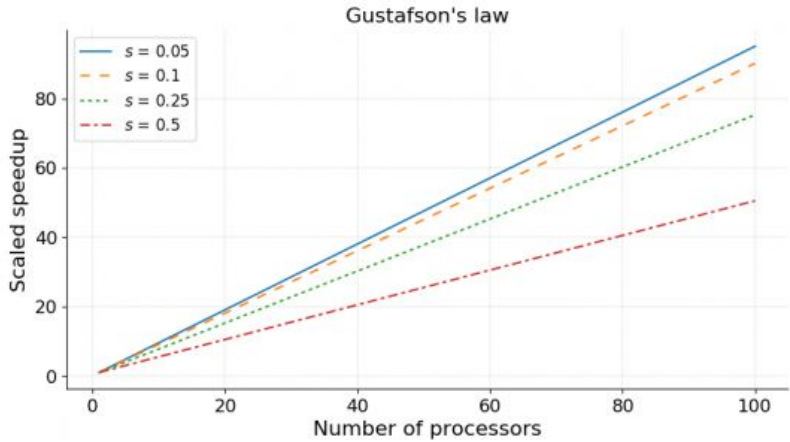
- Serial components kill performance



Measuring performance scaling: Weak scaling

- Strong scaling keeps problem size same, increase number of available resources
- More realistic:
  - Increase problem size with available resources.
  - Aim to saturate each available resource.

$$S_{weak} = s + p \times N$$



Summary and takeaways

- To parallelize your code, look at:
  - Dependencies
  - Available resources
- Decompose your problem to maximize parallelism
- Reframe your algorithm to have building blocks of high arithmetic intensity
- Be aware of cache effects and NUMA effects
- Code → measure performance → re-code → measure

Next week

- Parallel programming models
  - Shared and distributed parallel programming models
  - CUDA (NVIDIA), HIP (AMD), SYCL (Intel), Metal (Apple)
- GPU hardware architecture
  - Important differences to CPUs
  - Compute and Memory hierarchies
  - Brief look at ISA