University of Strasbourg

USAZ FRENCH-AZERBAIJANI UNIVERSITY

Introduction to Parallel Programming

Cédric Bastoul

cedric.bastoul@unistra.fr

University of Strasbourg – French-Azerbaijani University

References

This course mainly refers/reuses/adapts the following materials:

- *Introduction to Parallel Computing* by Doug James, Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License
- *Parallel Computing* course by Vincent Loechner at the university of Strasbourg
- *Introduction to Parallel Computing* by Blaise Barney, Lawrence Livermore National Laboratory

  https://computing.llnl.gov/tutorials/parallel_comp

Parallel Computing

Definition [Wikipedia]

Parallel computing is a type of computation in which several calculations are carried out simultaneously

Objectives:

- Complete computation faster
  - Allocate more computing resources to a given task
- Solve larger problems
  - Overcome single machine limits (speed, memory)
- Address several problems simultaneously
  - Enable concurrency
- Exploit hardware resources
  - Use parallel features of modern architectures

Why Do We Need Performance?

- Simulation
  - Natural phenomena: meteorology, seismology, etc.
  - Economics: market trends, fast trading, etc.
  - Life: biology, medicine, etc.
  - Industrial applications: durability, aerodynamics, etc.
  - Military applications: ballistics, nuclear weapons, etc.
- Human-Computer Interaction
  - Voice recognition, authentication, sensors
  - Artificial intelligence
  - Responsiveness
- Data Processing
  - Signal, photo, video, etc.
  - Big data
- For everything ;-)

Computational Cost Unit: The flop

Definition

The computational cost of a problem is the number of floating point operations (flop) necessary to solve it

- Count them all!
- Some useful patterns:

```
for (i = 0; i < N; i++)
  one_flop();
```

$\rightarrow \sum_{i=1}^{N} 1 \text{ flop} \quad = N \text{ flop}$

```
for (i = 0; i < N; i++)
  for (j = i; j < N; j++)
    one_flop();
```

$\rightsquigarrow \sum_{i=1}^{N} i \text{ flop} \quad \approx N^2/2 \text{ flop}$

```
for (i = 0; i < N; i++)
  for (j = i; j < N; j++)
    for (k = i; k < N; k++)
      one_flop();
```

$\rightsquigarrow \sum_{i=1}^{N} i^2 \text{ flop} \quad \approx N^3/3 \text{ flop}$

Performance Unit: The flop/s

Definition [Wikipedia]

Floating point operations per second (flop/s) is a measure of computer performance

- LINPACK de facto standard to compare computing power (flop/s solving a system of linear equations) see [top500]

Year	Performance		Computer
1947	1 kflop/s	10^3	ENIAC
1984	50 kflop/s	10^4	Intel 8087 (math coprocessor)
1997	23 Mflop/s	10^7	Intel Pentium MMX (200 MHz)
2007	1.5 Gflop/s	10^9	Intel Core 2 Duo (2.4 GHz)
2017	10 Tflop/s	10^{13}	Intel Core i7 + GPU

Performance Unit: The flop/s

Definition [Wikipedia]

Floating point operations per second (flop/s) is a measure of computer performance

- LINPACK de facto standard to compare computing power (flop/s solving a system of linear equations) see [top500]

Year	Performance		Computer
1947	1 kflop/s	10^3	ENIAC
1984	50 kflop/s	10^4	Intel 8087 (math coprocessor)
	800 Mflop/s	10^9	**Cray X-MP/48**
1997	23 Mflop/s	10^7	Intel Pentium MMX (200 MHz)
	1 Tflop/s	10^{12}	**Intel ASCI Red**
2007	1.5 Gflop/s	10^9	Intel Core 2 Duo (2.4 GHz)
	500 Tflop/s	10^{14}	**IBM BlueGene/L**
2017	10 Tflop/s	10^{13}	Intel Core i7 + GPU
	93 Pflop/s	10^{17}	**Sunway TaihuLight**
2019	**148 Pflop/s**	10^{17}	**IBM Summit**

Performance Example

Matrix Multiplication

- $C = A \times B$ with matrix size 10000×10000
- 10000 times computation $c_{i,j} = \sum a_{i,k} \times b_{k,j}$
- Computational cost $= 2 \times 10^4 \times 10^4 \times 10^4 = 10^{12}$ flop

Execution time:

- 1947 – ENIAC: 63 years
- 1997 – PC: 24 minutes
- 1997 – ASCI Red: 2 s
- 2019 – PC: 0.2 s $\rightarrow$ 5 per second
- 2019 – Summit: < 0.02 ms $\rightarrow$ 60000 per second

How Do We Get Performance?

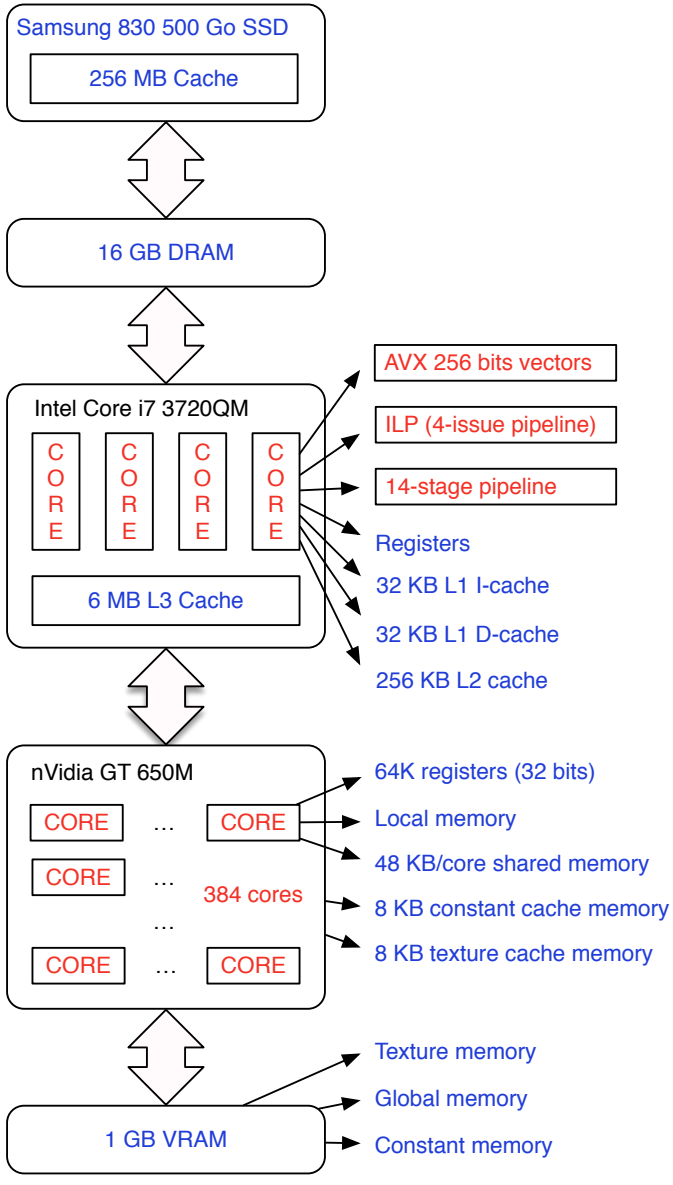
- Understand the problem
  - Chose the best data structures and algorithms
  - Decompose the problem into (independent) tasks
  - Analyze data dependences
- Understand the target architecture
  - Chose the best tools (languages, libraries, compilers)
  - Exploit available resources
  - Optimize
- Parallelization is a key

This course is an introduction to all these aspects

Review of a Modern Architecture

My computer:

- 2.7 billion transistors
- **17 types of memory**
- 5 types of parallelism
- At least 4 programming models + API



Cluster of Modern Architectures

©Carlos Jones/ORNL Wikimedia

IBM Summit Supercomputer

- World's most powerful computer as of November 2019
- 4,608 nodes, 2,414,592 cores, 148 Pflop/s
- Dual-rail Mellanox EDR Infiniband interconnect

Types of Parallelism

Bit-Level Parallelism

Definition [Wikipedia]

Bit-level parallelism performs basic operations on processor words in parallel

- Increase processor word size (e.g., 32 bits to 64 bits)
  - Reduce the number of instructions to process wide data
- Exploit elementary circuits in parallel
  - Use additional logic for more performance
- Example: addition
  - Right-to-left sequential process because of the carry-in!
  - How to achieve a faster addition with several adders?

Bit-Level Parallelism

Definition [Wikipedia]

Bit-level parallelism performs basic operations on processor words in parallel

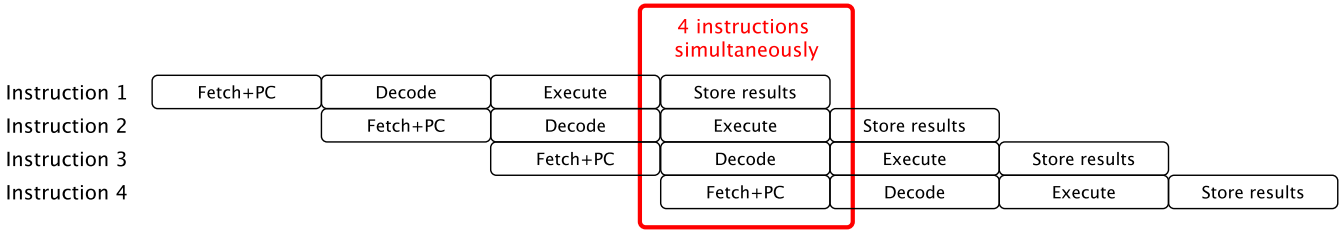
- Increase processor word size (e.g., 32 bits to 64 bits)
  - Reduce the number of instructions to process wide data
- Exploit elementary circuits in parallel
  - Use additional logic for more performance
- Example: addition
  - Right-to-left sequential process because of the carry-in!
  - How to achieve a faster addition with several adders?
  - Carry-select adder: e.g., in parallel one adder for the right part, one adder for the left part with carry-in, one adder for the left part without carry-in, then trivially select the left part

Instruction-Level Parallelism

Definition [Wikipedia]

Instruction-level parallelism performs multiple instructions from the same instruction stream concurrently

- Exploit the multiple functional units of the processor
  - Managed by hardware (superscalar) or compiler (VLIW)
  - Limited by control or data dependences (speculation helps)
- Example: instruction pipeline
  - Decompose the instruction cycle to `n` stages to allow the simultaneous processing of up to `n` instructions
  - Start an instruction cycle before finishing the previous one

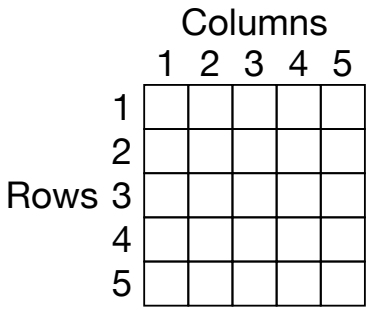


Data-Level Parallelism

Definition [Wikipedia]

Data-level parallelism performs the same processing on mutiple data operands concurrently

- Partition the data to different computation units
  - Applies preferably on regular data structures like arrays
  - Limited by the memory bandwidth
- Execute same instructions synchronously
- Example: graphics processing
  - 4K display at 60 FPS: 500+ million pixels per second!



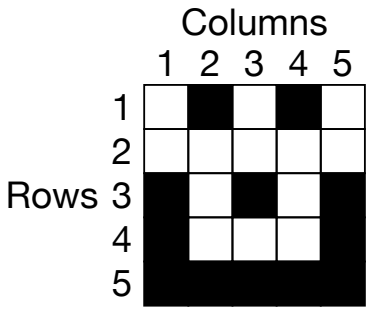

```
// Ideal case: compute each pixel independently
if (row == 1 && is_even(column))
  pixel = BLACK;
else if (row == 3 && is_odd(column))
  pixel = BLACK;
else if (row > 3 && is_on_border(row, column))
  pixel = BLACK;
else
  pixel = WHITE;
```

Data-Level Parallelism

Definition [Wikipedia]

Data-level parallelism performs the same processing on mutiple data operands concurrently

- Partition the data to different computation units
  - Applies preferably on regular data structures like arrays
  - Limited by the memory bandwidth
- Execute same instructions synchronously
- Example: graphics processing
  - 4K display at 60 FPS: 500+ million pixels per second!




```
// Ideal case: compute each pixel independently
if (row == 1 && is_even(column))
  pixel = BLACK;
else if (row == 3 && is_odd(column))
  pixel = BLACK;
else if (row > 3 && is_on_border(row, column))
  pixel = BLACK;
else
  pixel = WHITE;
```

Task/Thread-Level Parallelism

Definition [Wikipedia]

Task/Thread-level parallelism performs multiple instruction sequences from the same application concurrently

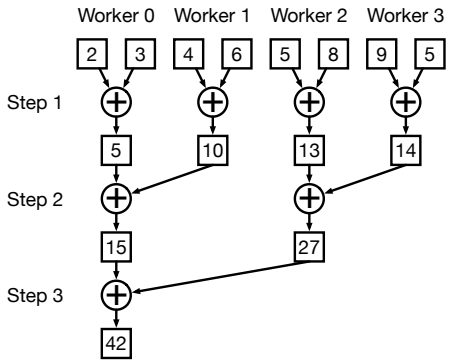
- Partition application into tasks to be run simultaneously
  - Identify tasks manually (user) or automatically (compiler)
  - Limited by communication and/or synchronization costs
- Example: compute the sum of n numbers (reduction)
  - How computing units may share work and communicate?

Task/Thread-Level Parallelism

Definition [Wikipedia]

Task/Thread-level parallelism performs multiple instruction sequences from the same application concurrently

- Partition application into tasks to be run simultaneously
  - Identify tasks manually (user) or automatically (compiler)
  - Limited by communication and/or synchronization costs
- Example: compute the sum of n numbers (reduction)
  - How computing units may share work and communicate?



- Activate $n/2$ workers, each with 2 numbers
- Each worker computes its intermediate result
- Half workers communicate their result and deactivate
- Iterate the two last step until only one worker is active

Accelerator-Level Parallelism

Definition [Wikipedia]

Accelerator-level parallelism performs application-specific processing concurrently on dedicated hardware

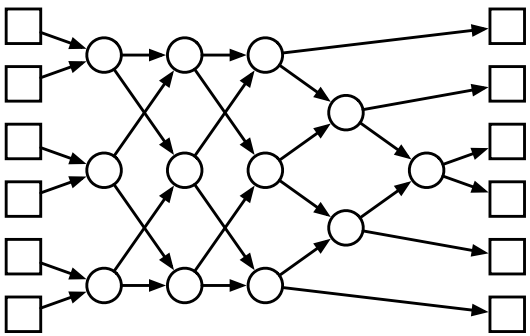
- Exploit application-specific properties
  - Implement specialized functional component
  - Link components using specialized interconnects
- Example: sort strings in alphabetical order
  - How more computation units may be more efficient?

Accelerator-Level Parallelism

Definition [Wikipedia]

Accelerator-level parallelism performs application-specific processing concurrently on dedicated hardware

- Exploit application-specific properties
  - Implement specialized functional component
  - Link components using specialized interconnects
- Example: sort strings in alphabetical order
  - How more computation units may be more efficient?



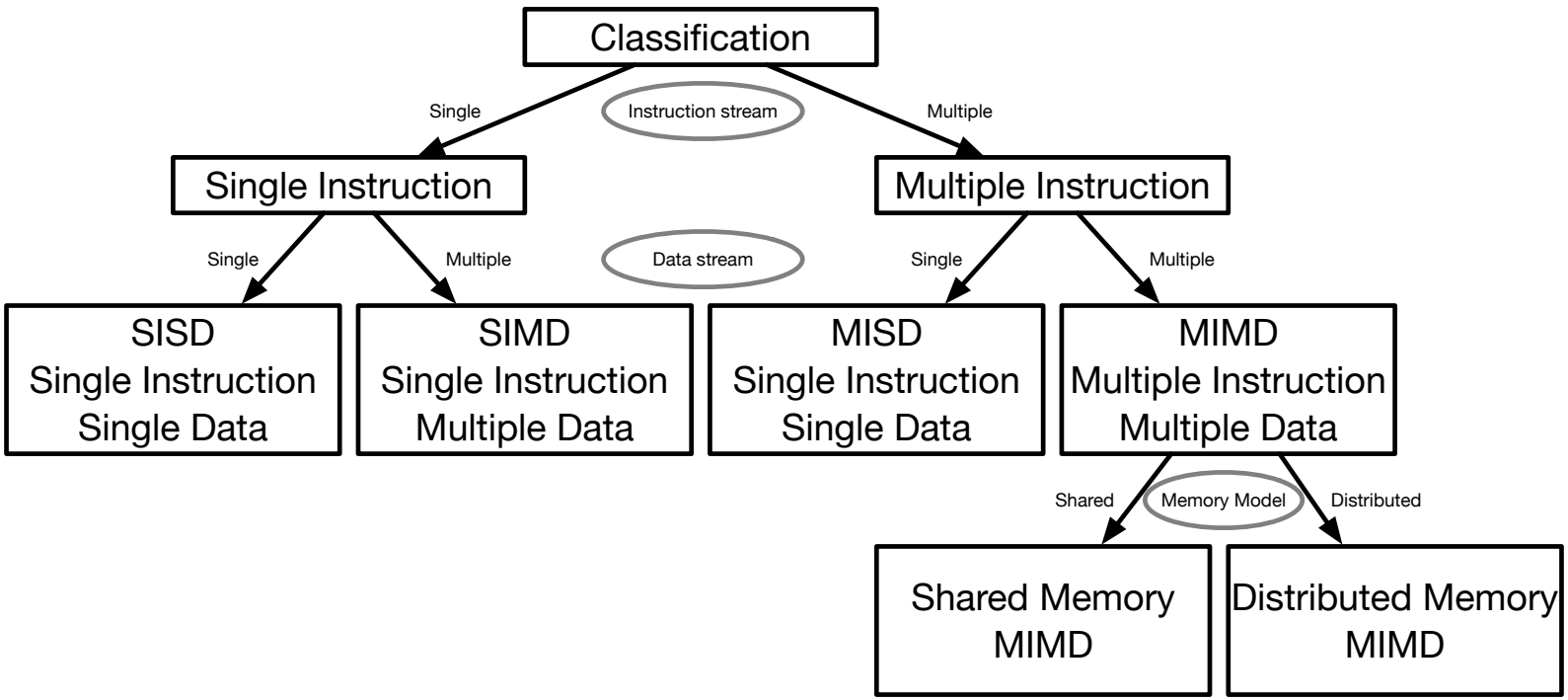
- Specialized comparator units
  - Input: two strings
  - Output: lower string on left, higher on right
- Specialized interconnection
  - Whole accelerator has n input and output
  - Interconnect so that the output is sorted

Flynn's Taxonomy

Flynn's Taxonomy

Definition [Wikipedia]

Flynn's Taxonomy is a classification for parallel and sequential systems and programs proposed by Michael J. Flynn in 1966, it distinguishes single or multiple instruction or data streams

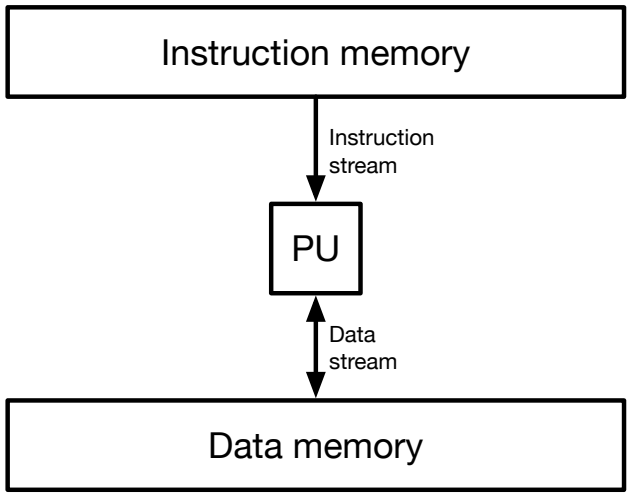


Single Instruction, Single Data (SISD)

Definition [Wikipedia]

SISD is a computer architecture class in which a single processing unit executes a single instruction stream to operate on a single data stream

- Standard uniprocessor
- Sequential processing
- Von Neuman architecture
- E.g., ENIAC, Intel 8088

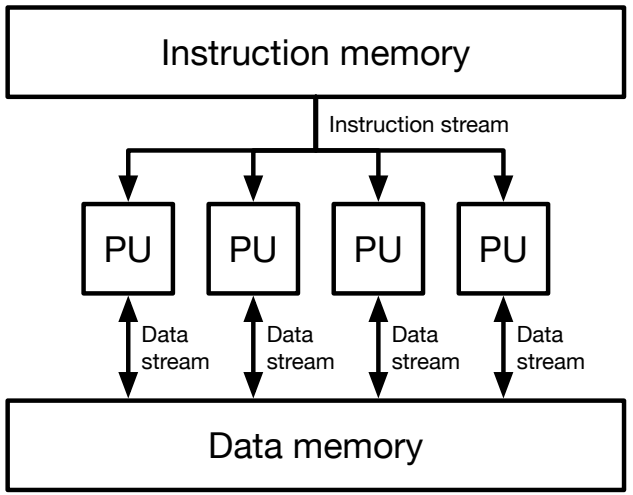


Single Instruction, Multiple Data (SIMD)

Definition [Wikipedia]

SIMD is a computer architecture class in which multiple processing units execute the same instruction stream on multiple data streams simultaneously

- Appropriate for regular problems such as scientific or image processing
- Benefit from loading mechanisms which load several data at once
- E.g., modern processor vector instructions, GPUs

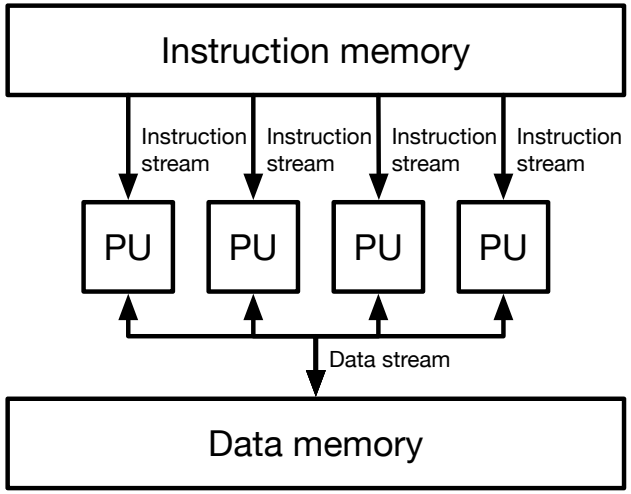


Multiple Instruction, Single Data (MISD)

Definition [Wikipedia]

MISD is a computer architecture class in which multiple processing units perform the same operation on the same data

- Not very useful in practice
- Fault tolerant systems executing the same instruction redundantly, or instruction pipeline may be considered as MISD implementations

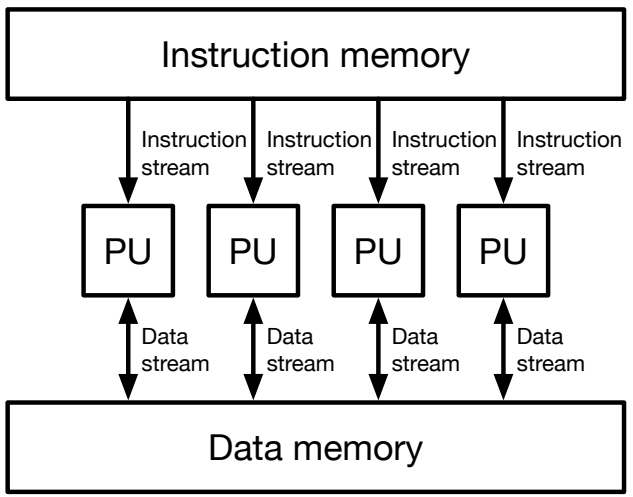


Multiple Instruction, Multiple Data (MIMD)

Definition [Wikipedia]

MIMD is a computer architecture class in which multiple processing units can perform different operations on different data

- Very general and flexible
- Easier to build and to program
- Either shared memory or distributed memory
- E.g., multi-core processors, computer clusters

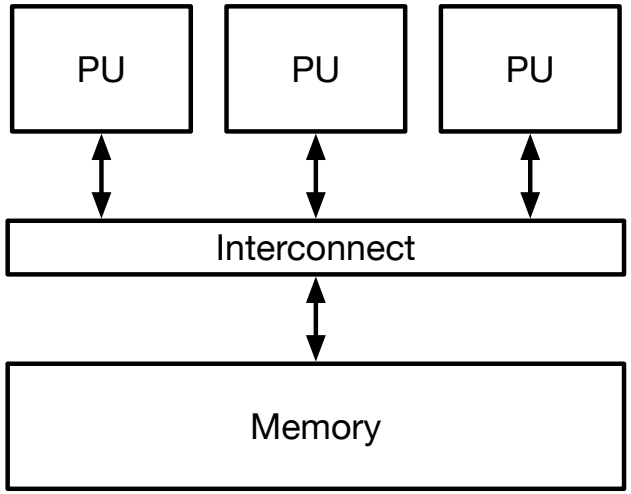


Shared Memory MIMD

Definition [Wikipedia]

Shared memory is a subclass of MIMD where processors share a common memory space and they all have access to it

- All processes can use the same virtual address space
- Communication through shared variables
- Hardware and system support for synchronization
- Easier to program (but beware of race conditions)
- E.g., multi-core processors

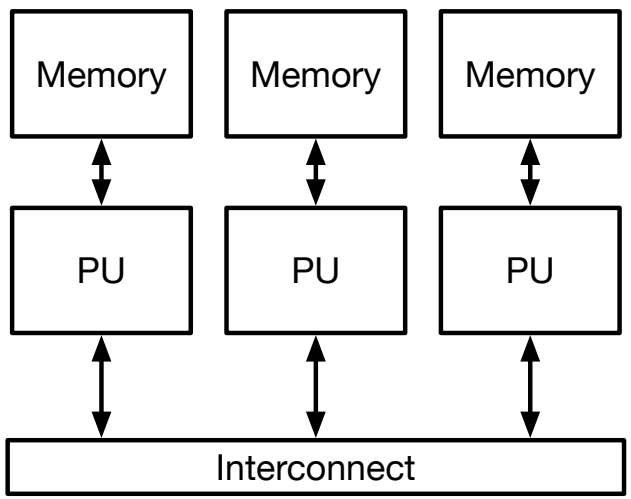


Distributed Memory MIMD

Definition [Wikipedia]

Distributed memory is a subclass of MIMD where each processor has its own private memory and is connected to other processors through an interconnection network

- All processes use different virtual address spaces
- Explicit messages for communication and synchronization
- Harder to program, high communication latency
- E.g., computer clusters



Performance Limits

Switching to Doug James's document

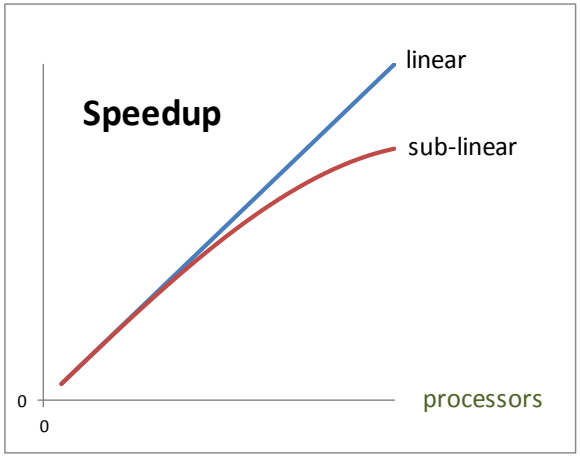
Measuring Scalability (1/3)

Two basic metrics

Speedup on p processors*:

$$S_p = \frac{\text{time on one processor}}{\text{time on } p \text{ processors}}$$

($S_p = p$ is perfect)



*here, "processor" (or "process") could mean many things depending on context: OpenMP thread, MPI task, core, node…

Measuring Scalability (2/3)

Two basic metrics

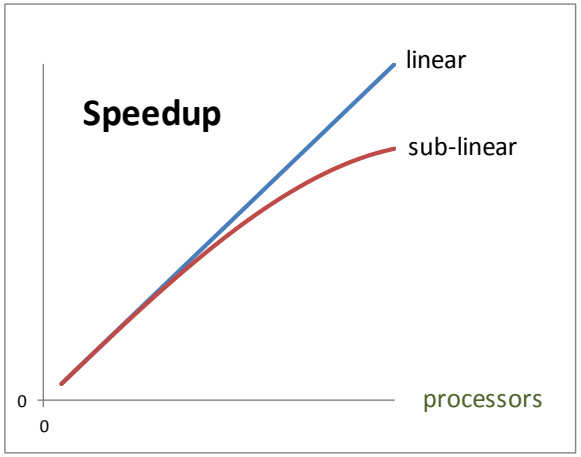
Speedup on p processors*:

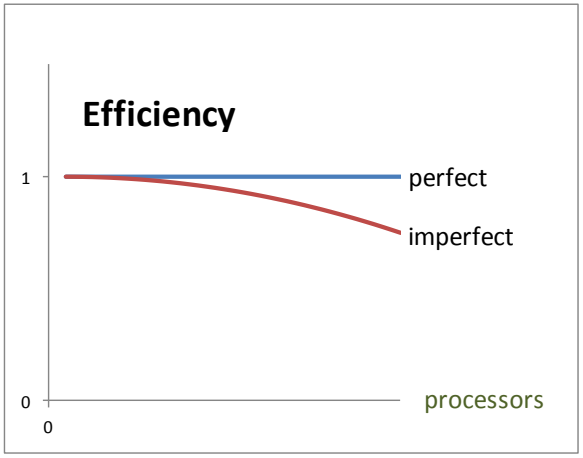
$$S_p = \frac{\text{time on one processor}}{\text{time on } p \text{ processors}}$$

($S_p = p$ is perfect)

Efficiency on p processors:

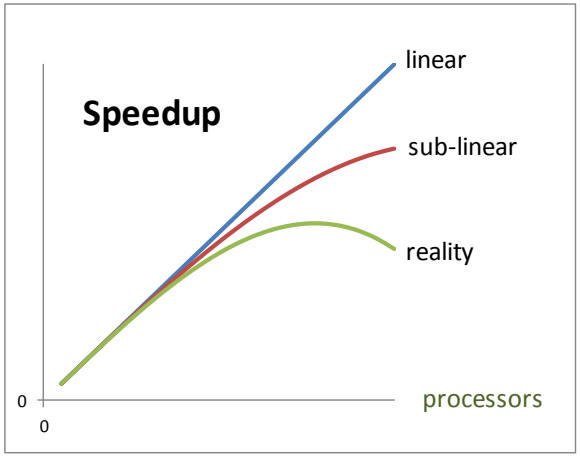
$$E_p = \frac{S_p}{p} \qquad (E_p = 1 \text{ is perfect})$$

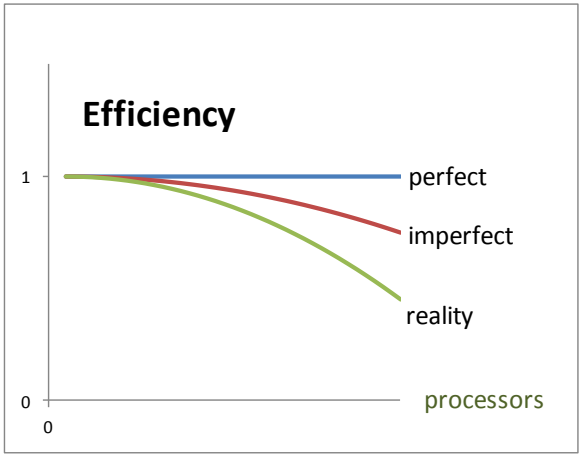




Measuring Scalability (3/3)

- Reality sets in
  - Expect to lag behind perfection
  - Expect a sweet spot beyond which adding processors will make things worse
- There are several reasons
  - Various kinds of overhead
  - Communication costs
  - Load imbalances
  - But one particular issue that deserves its own discussion…





How Fast Can We Bake A Cake? (1/4)

- Assume (with no claim of realism)…
  - Two hours to prepare the cake for baking
  - Another half hour to bake the cake
- So total time is…

```
Prep: 120 minutes
Bake:  30 minutes
Total: 150 minutes
```

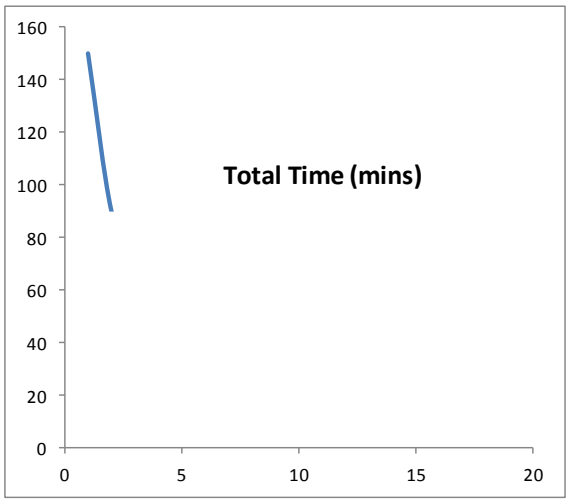
How Fast Can We Bake A Cake? (2/4)

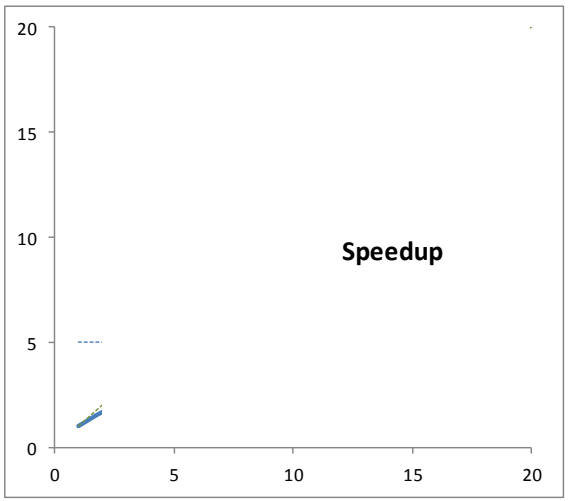
- But what if we have two cooks?
  - Assume they work perfectly together
  - No wasted time, no overhead
- Then total time is…

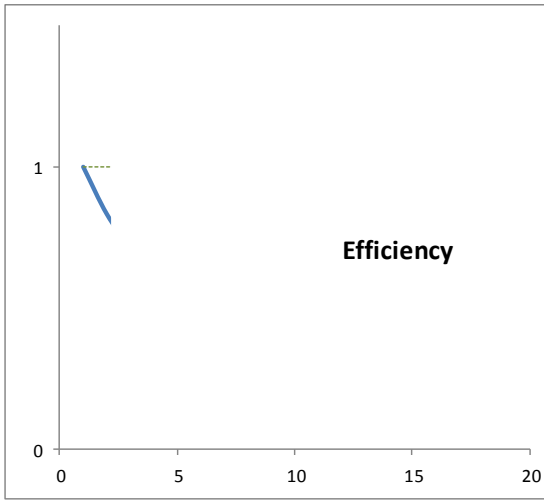
```
Prep: 120/2 = 60 minutes
Bake:         30 minutes
Total:        90 minutes
```

How Fast Can We Bake A Cake? (3/4)

cooks	prep*	bake	total	speedup	efficiency
1	120/1 = 120.0	30.0	150.0	1.0	1.0
2	120/2 = 60.0	30.0	90.0	1.7	0.8

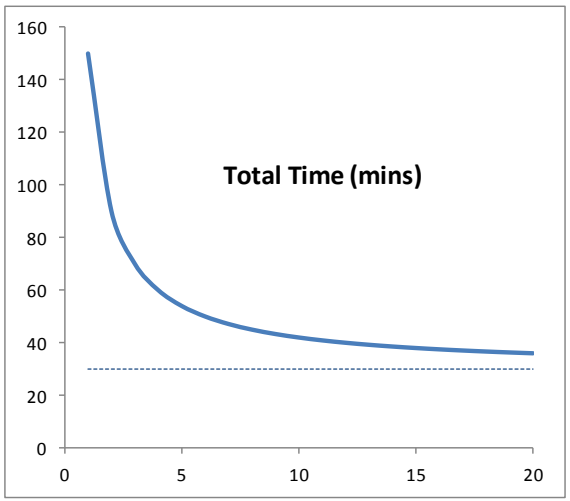


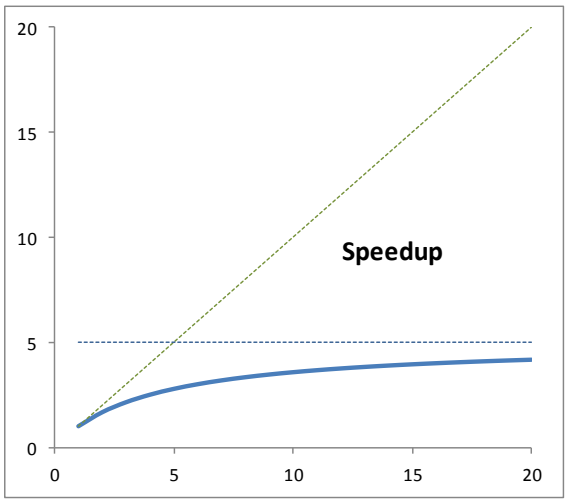


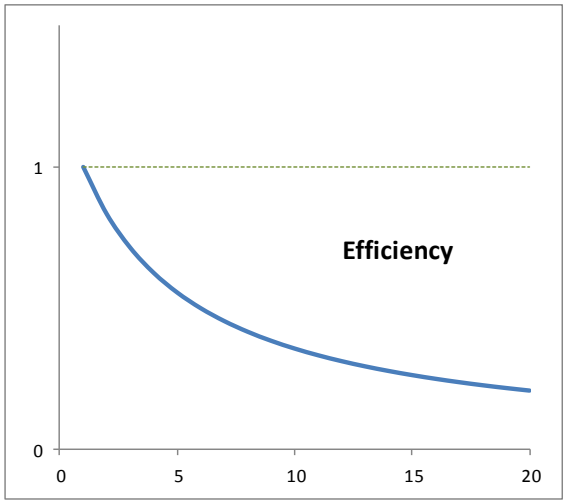


How Fast Can We Bake A Cake? (4/4)

cooks	prep*	bake	total	speedup	efficiency
1	120/1 = 120.0	30.0	150.0	1.0	1.0
2	120/2 = 60.0	30.0	90.0	1.7	0.8
3	120/3 = 40.0	30.0	70.0	2.1	0.7
4	120/4 = 30.0	30.0	60.0	2.5	0.6
5	120/5 = 24.0	30.0	54.0	2.8	0.6
19	120/19 = 6.3	30.0	36.3	4.1	0.2
20	120/20 = 6.0	30.0	36.0	4.2	0.2



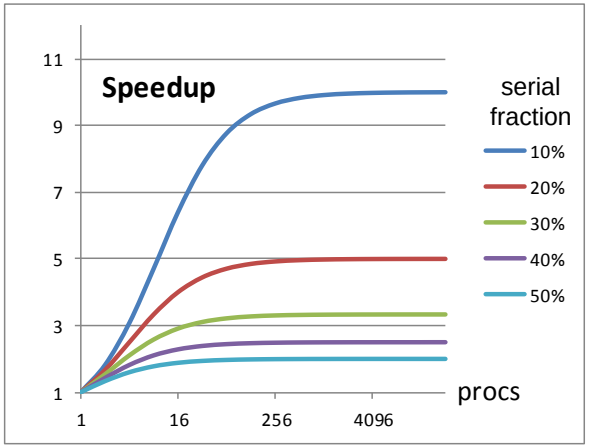


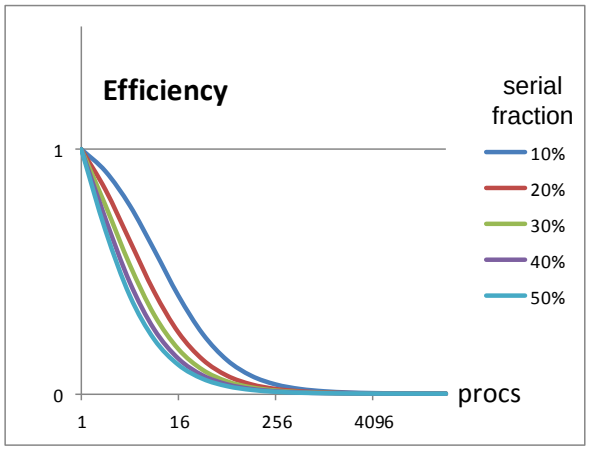


Amdahl's Law: Strong Scaling (1/3)

- Back-of-napkin scalability bounds

$$S_p = \frac{T_1}{T_p} \leq \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \rightarrow \frac{1}{\alpha}$$





Amdahl's Law: Strong Scaling (2/3)

- Back-of-napkin scalability bounds

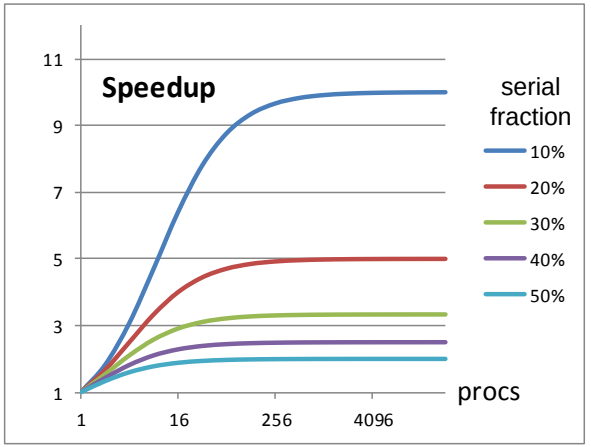
$$S_p = \frac{T_1}{T_p} \leq \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \rightarrow \frac{1}{\alpha}$$

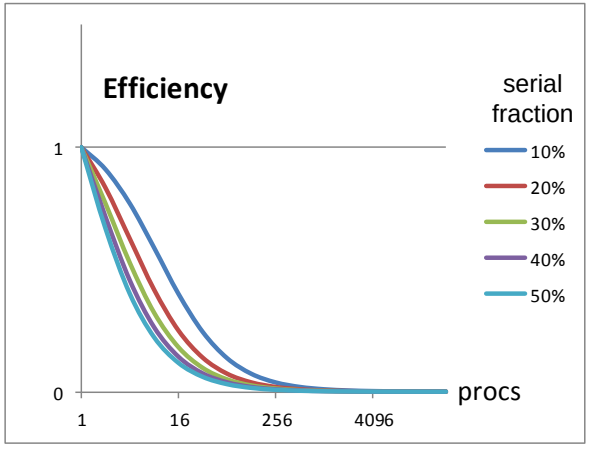
actual speedup

perfect on parallel section

infinitely fast on parallel section

reciprocal of serial fraction



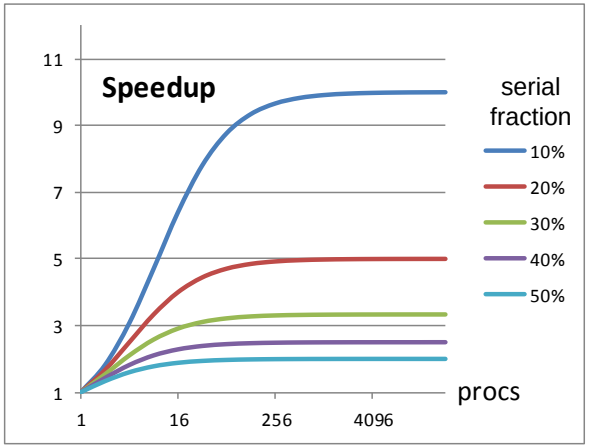


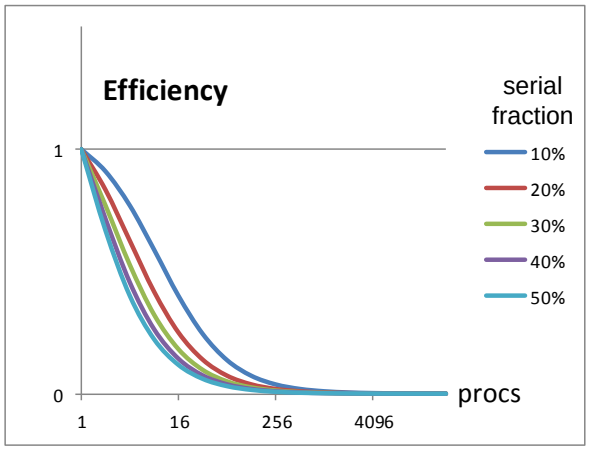
Amdahl's Law: Strong Scaling (3/3)

- Back-of-napkin scalability bounds

$$S_p = \frac{T_1}{T_p} \leq \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \rightarrow \frac{1}{\alpha}$$

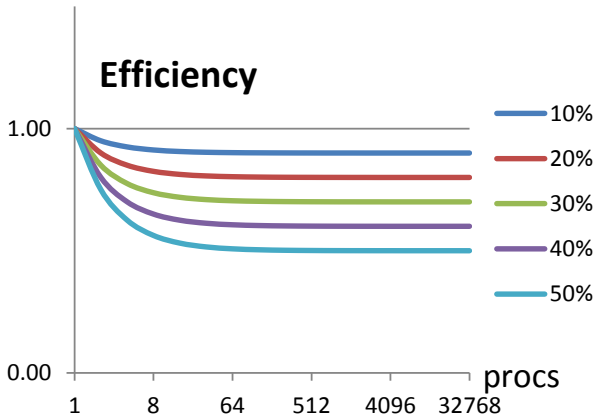
- Under reasonable assumptions…
  - Serial fraction is a severe constraint
  - As you add processors, you reach a point of diminishing returns
  - Can't run faster than serial time
  - Speedup bounded by the reciprocal of the serial fraction





Gustafson's Law: Weak Scaling (1/2)

- Increase problem size in proportion to processors
- How big a problem can we solve in the same time as the serial problem?
- Generalizing meaning of speedup...

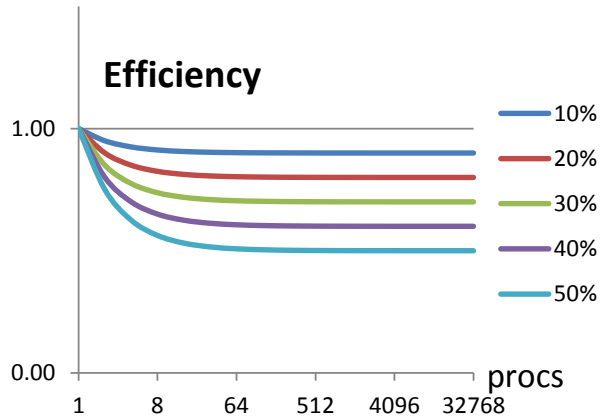


S_p bounded by $\approx (1 - \alpha)p$

E_p bounded by $\approx 1 - \alpha$

Gustafson's Law: Weak Scaling (2/2)

- Picture is a bit more optimistic
- Serial fraction is still a constraint
- But incremental benefit of additional processors can continue indefinitely (at least in theory)



S_p bounded by $\approx (1 - \alpha)p$

E_p bounded by $\approx 1 - \alpha$

Parallelization

Program Parallelization

Definition

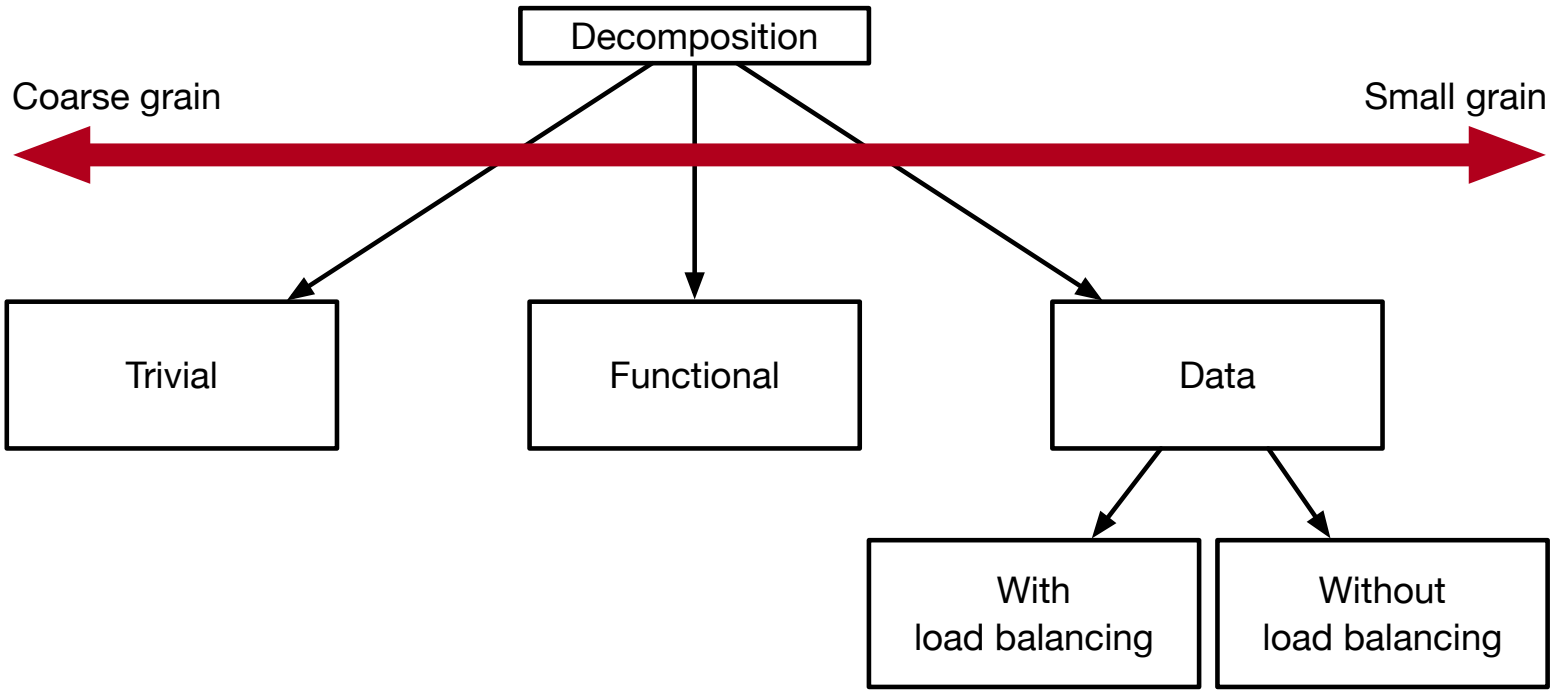
Parallelization is the process of identifying parallelism, extracting parallelism and implementing parallelism in a sequential code

- Tools like compilers may parallelize a code automatically but they are still rather limited
  - May be good for simple loops
  - Do not optimize if the code is too complex to analyze
- Manual work is often necessary for efficient parallelization
  - Identify hotspots, i.e., where the time is spent (90/10 rule)
  - Find bottlenecks, i.e., what limits performance (e.g., I/O)
  - Consider better algorithms or existing parallel libraries
  - Determine whether the processing may be parallelized
  - Decompose the workload to expose parallelism

Workload Decomposition

Definition

Workload decomposition is finding program parts that may be run concurrently, they may include many operation (coarse grain) or only one to few operations (small grain)



Trivial Decomposition

Definition [Wikipedia]

Trivial decomposition is possible for the "embarrassingly parallel" class of programs where no/little manipulation is necessary to separate the processing into concurrent tasks

- Process similar, independent tasks
- Trivial parallelization: launch the tasks concurrently!
- Typical cases: parameter sweep, brute-force searches, genetic algorithms, 3D video rendering, etc.

Functional Decomposition

Definition [Wikipedia]

Functional decomposition is the process of breaking a long/complex process into smaller/simpler tasks corresponding to basic operations

- Build a task dependency graph
  - Analyze the input/output data of each task
  - A task depends on another if it requires its output
  - Independent tasks may be run concurrently
- Assign tasks to computation units carefully
  - Minimize communications
  - Maximize concurrency

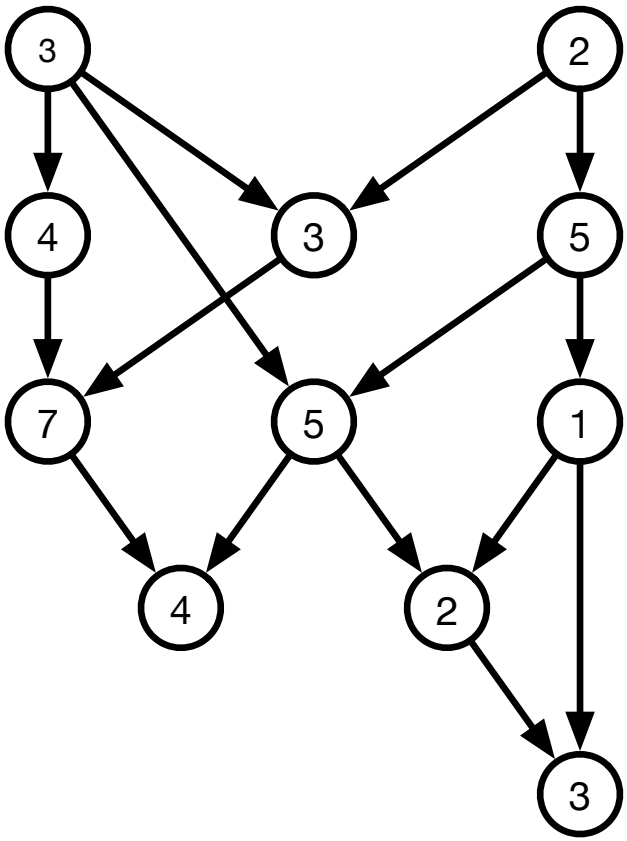
Task Dependency Graph

Definition [Wikipedia]

A task dependency graph is a directed acyclic graph (DAG) representing dependencies (edges) between tasks (nodes, which may have an associated cost)

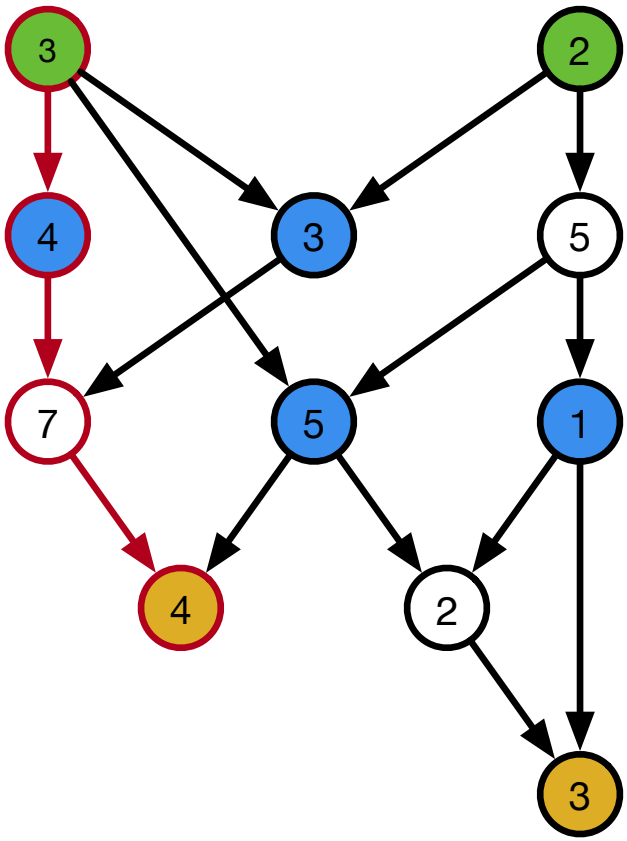
- Represent valid relative execution orders
- Important notions:
  - **Path length**: sum of costs of tasks along the path
  - **Critical path**: longest path between a start task (no incoming edge) and finish task (no outgoing edge)
  - **Maximum degree of concurrency**: largest number of tasks that can execute concurrently
  - **Average degree of concurrency** (ADG): average number of tasks that can execute concurrently
    ADG = total amount of costs / critical path length

Task Dependency Graph: Example



- Start tasks:
- Finish tasks:
- Critical path length:
- Maximum concurrency degree:
- Average Concurrency degree:

Task Dependency Graph: Example



- Start tasks:
  See green nodes
- Finish tasks:
  See orange nodes
- Critical path length:
  18, see red path
- Maximum concurrency degree:
  4, see blue nodes
- Average Concurrency degree:
  $39/18 = 2.2$

Data Decomposition

Definition [Wikipedia]

Data decomposition is the process of partitioning the data space on which the computation is performed and to partition the problem into tasks accordingly

- Same operations are performed on different data subsets
- Amount of parallelism is proportional to the dataset size
- Ideal for array/matrix/tensor computation
- Beware of data dependences!

Data Dependencies

Definition [Wikipedia]

Two operations are data-dependent on each other if they access the same memory location, and at least one access is a write

- Bernstein conditions: two operations S_1 and S_2 such that S_1 executes before S_2 sequentially are independent if:

$$\begin{cases} \mathcal{W}(S_1) \cap \mathcal{R}(S_2) = \emptyset & (\text{if} \neq \emptyset \rightarrow \text{flow dependencies}) \\ \mathcal{R}(S_1) \cap \mathcal{W}(S_2) = \emptyset & (\text{if} \neq \emptyset \rightarrow \text{anti dependencies}) \\ \mathcal{W}(S_1) \cap \mathcal{W}(S_2) = \emptyset & (\text{if} \neq \emptyset \rightarrow \text{output dependencies}) \end{cases}$$

  $\mathcal{W}(S)/\mathcal{R}(S)$: sets of memory locations written/read by S

- Data-dependent operations can't be executed concurrently
- But modifying the code may help :)

Data Dependencies: Example

Code

```
int a, b, c;
S1: a = 2;
S2: b = a + 40;
S3: c = a + 775;
S4: a = 0;
```

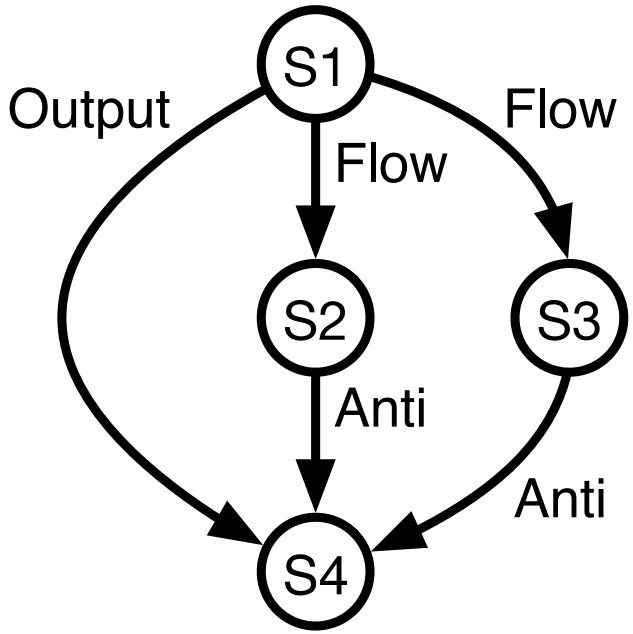
Dependence Graph

Data Dependencies: Example

Code

```
int a, b, c;
S1: a = 2;
S2: b = a + 40;
S3: c = a + 775;
S4: a = 0;
```

Dependence Graph



Data Dependencies in Loops

Definition [Wikipedia]

- A **loop-independent dependency** exists when a statement in an iteration of a loop depends on a statement in the same iteration of that loop
- A **loop-carried dependency** exists when a statement in an iteration of a loop depends on a statement in another iteration of that loop

- Loop-independent dependencies do not prevent running iterations of the loop concurrently
- Loop-carried dependencres do prevent running iterations of the loop concurrently (if possible, loop transformations may improve the situation)

Data Dependencies in Loops: Example

- Draw the dependency graph between the first five iterations

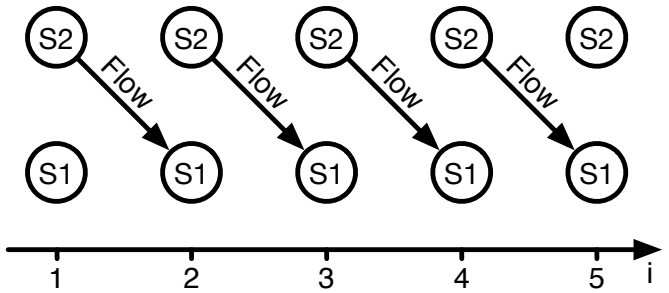
```
    for (i = 1; i < N; i++) {
S1:   b[i] = b[i] + a[i-1];
S2:   a[i] = a[i] + c[i];
    }
```

- Transform the code to allow concurrent iterations

Data Dependencies in Loops: Example

- Draw the dependency graph between the first five iterations

```
    for (i = 1; i < N; i++) {
S1:   b[i] = b[i] + a[i-1];
S2:   a[i] = a[i] + c[i];
    }
```

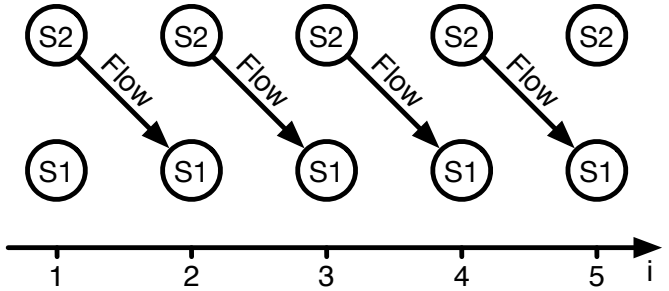


- Transform the code to allow concurrent iterations

Data Dependencies in Loops: Example

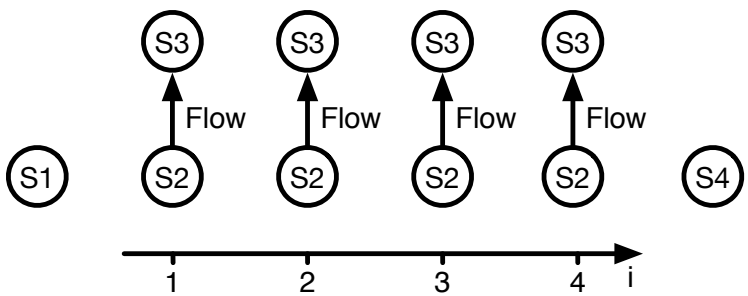
- Draw the dependency graph between the first five iterations

```
    for (i = 1; i < N; i++) {
S1:   b[i] = b[i] + a[i-1];
S2:   a[i] = a[i] + c[i];
    }
```



- Transform the code to allow concurrent iterations

```
S1: b[1] = b[1] + a[0];
    for (i = 1; i < N-1; i++) {
S2:   a[i] = a[i] + c[i];
S3:   b[i+1] = b[i+1] + a[i];
    }
S4: a[N-1] = a[N-1] + c[N-1];
```



Take Home Messages

- More computing power comes only from more parallelism
  - Parallelism is a major key to performance
  - But it has limits (Amdhal's law)
- Many types of parallelism coexist
  - They are addressed differently in hardware
  - The appropriate programming model should be used
- Analyzing the problem is necessary to expose parallelism
  - This is a complex task (dependencies!)
  - Converting parallelism to performance is complex too