COSC 407

Intro to Parallel Computing

Topic 11 - Speedup vs. Efficiency

COSC 407: Intro to Parallel Computing

1

Introduction

- Speed of execution depends on many factors, one of them is good algorithm and code
- Factors that affect the execution time
- Then focus on the factors related to the code (algorithm)
  - Speedup and Efficiency
  - Amdahl's and Gustafson's Laws

2

Computer Performance

How to measure CPU performance?

Response Time (aka **Execution Time**) Our focus today

The **time** the CPU takes to finish a given task

This includes all subtasks (e.g., memory access, I/O activities, CPU execution time, etc.).

Example: How long should I wait for a DB query?

Unit of measurement: seconds

Throughput

The **total work** finished per unit of time We'll talk about this later (CUDA)

Here, we don't care about the amount of time spent on each element, but the total work done per second.

Example: how many client queries can a ***server*** respond to per minute

Unit of measurement: number of jobs finished per seconds

Analogy

You are a store manager and you want to hire employees to work at checkout registers with a $500K/year budget for salaries. Assume you have two options:

1) hire 10 slow employees:

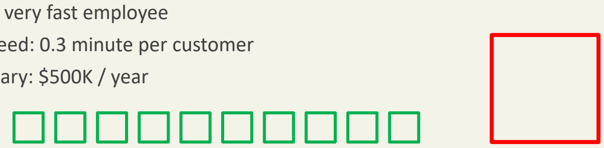
Speed: 2 minutes per customer (i.e. each employee takes 2 min / customer)

Salary: $50K / year (total is $500K/year)

2) hire 1 very fast employee

Speed: 0.3 minute per customer

Salary: $500K / year



Q1: Which option would you choose?

Assume we want to checkout a 100 customer per hour.

Q2: What if you have to make the same decision but you are a CEO for a big company and you want accountants to handle the taxes of very important client?

	Option 1	Option 2
Execution time	2 min	0.5 min
Throughput	20 min	30 min

CPU Performance

Response Time for a program A is split to:

- **CPU time** is the total time the CPU spends **on your program**.
  - **user CPU time**: time CPU spent running A
  - **system CPU time**: time CPU spent executing OS routines issued by A
- **Wait time**: I/O operations and time sharing with other programs.

Metrics

IPS = Instructions Per Second

- Measures approximate number of machine instruction a CPU can execute per second
- Does not consider other factors that might affect execution time (e.g., wait time: memory access delay, I/O speed)
  - ***Disadvantage:*** A machine with higher IPS rating might not run a program any faster than a machine with lower IPS rating

MIPS = Million Instructions Per Second

FLOPS: Floating-point Operations Per Second

- **MFLOPS**: Million FLOPS **GFLOPS**: Billion FLOPS

→ larger FLOPS rates correspond to faster execution times

- ***Disadvantage:*** ignores non-floating-point operations

  → useful for scientific calculations that make heavy use of floating-point calculations

Aside

$$CPU\ Time = \frac{CPI \times InstructionCount}{ClockRate} = \frac{InstructionCount}{IPS}$$

- **CPI** (Clock cycles Per Instruction): is the the average number of CPU cycles used for instructions of program A
- CPI depends on the Architecture (ISA)
- Instruction Count depends on
  - the architecture (ISA) and
  - the code quality (the compiler and the algorithm)

$$IPS = \frac{Clock\ Rate}{CPI}$$

$$Average\ MIPS\ Rating = \frac{Clock\ Rate}{CPI \times 10^6}$$

Benchmarks

- We need **benchmark programs** to objectively evaluate the performance.
- **Benchmarks** are program that **capture the instruction mix** of a variety of applications.
- **Example: SPEC** (System Performance Evaluation Cooperative)
  - SPEC is an organization, established in 1988, that provides a **standardized set of performance benchmarks** for computers.
  - CPU-intensive real-world applications.
  - Provides good indicator of processor performance and compiler technology
  - More information: http://www.spec.org/

spec®

Overhead due to Parallelism

- Parallel runtime includes **computation** and **overhead**
- **Overhead includes**
  - Thread creation / destruction
  - Synchronization
  - Communication (exchange of data)
  - Waiting time due to:
    - Unequal load distribution
    - Mutual exclusion (waiting for a shared resource)
    - Condition synchronization (one thread is waiting for another thread's action that changes a condition)

Speedup and Efficiency of Parallel Programs

***Speedup* S** : how much faster is our parallel algorithm compared to the serial algorithm

$$S = \frac{T_{serial}}{T_{parallel}}$$

- Theoretically, **maximum speedup** S **= number of cores (p)**
  - When the speed up is equal to *p,* the algorithm is said to have ***linear speedup (no overhead)*** → very rare!
  - In practice, we have overhead: $T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$

Parallel Efficiency E : how effectively you're using your multiple cores:

$$E = \frac{actual\ speedup}{maximum\ possible\ speedup} = \frac{S}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

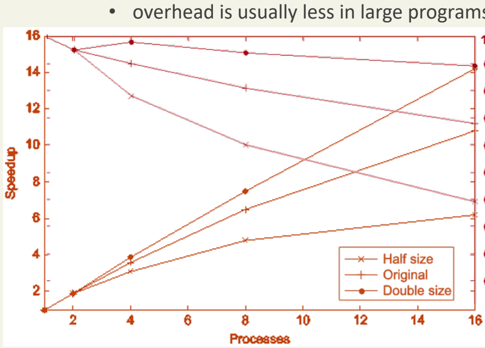
Question: how much does the ***# cores p*** and ***problem size*** affect ***S*** and ***E***?

Experiment: Effect of p and problem size on E and S

- Observations:
  1. As the number of cores p increases,
     - The speedup S increases
       - *non-linearly*
     - The efficiency E decreases
       - *due to the increased overhead*

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

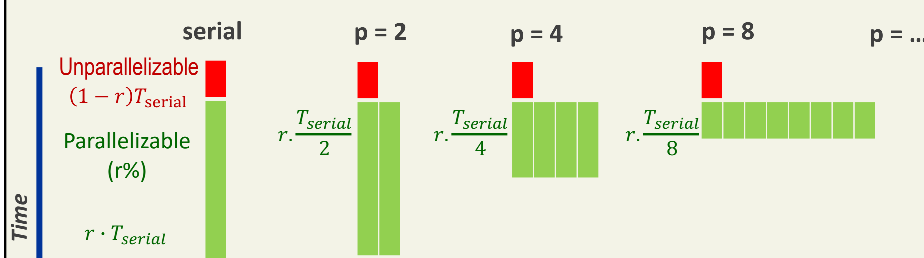
  2. As the problem size increases, both E and S increase.
     - overhead is usually less in large programs compared to computation.



	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Speedup Formula

- *Does adding more and more cores always mean better speedup?*
- To answer this question, let's find the speed up for any given program. Note that there will always be **part of your program that cannot be parallelized**.



Execution Time = $(1-r)T_{\text{serial}} + r.\frac{T_{serial}}{p}$

$$S = \frac{T_{serial}}{(1-r) \cdot T_{serial} + r \cdot \frac{T_{serial}}{p}}$$

$$\boxed{S = \frac{1}{1-r+\frac{r}{p}}}$$

Amdahl's Law

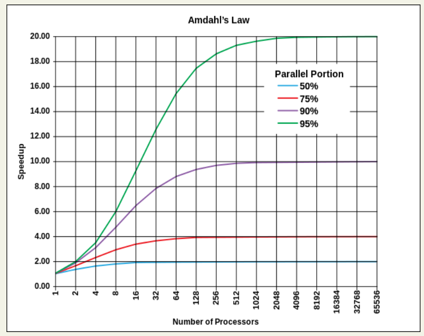
Does adding more and more cores always mean better speedup?

- Speedup is ***limited*** by the portion of unparallelizable code regardless of number of cores available

$$S = \frac{1}{1 - r + \frac{r}{p}}$$

As p → ∞,

maximum speedup is $\frac{1}{1-r}$



Amdahl's Law: *Example*

Let's say we can parallelize 90% of a serial program (r = 0.9)

Applying speedup formula:

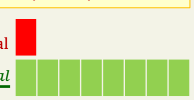
$$S = \frac{1}{1-0.9+\frac{0.9}{p}}$$

As p → ∞, **s→ 10** (maximum speedup)!!

Q: without math formulas, can you tell what the max speedup is?

$(1-r)T_{serial}$

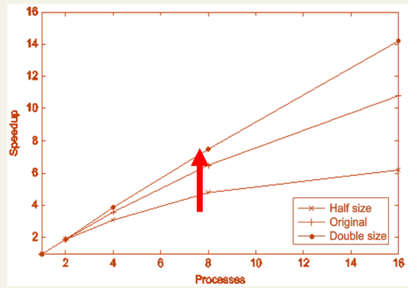
$r.\frac{T_{serial}}{P}$



Should we give up on parallelism?

- No, ***problem size*** is another factor that should be considered (as we saw before). And there are 'sweet values' that we should consider to achieve maximum possible efficiency vs. speedup.

Gustafson's Law



- Remember when we said that with as increased problem size we can achieve better speedup and efficiency?
- **Gustafson's** law shows that speedup can be increased with **larger problem sizes**
  – Or, with more cores, larger problem sizes can be solved within the same time.
- **Amdahl's** law **assumes fixed problem size** and shows how limited the speed up is considering the sequential part of the problem.

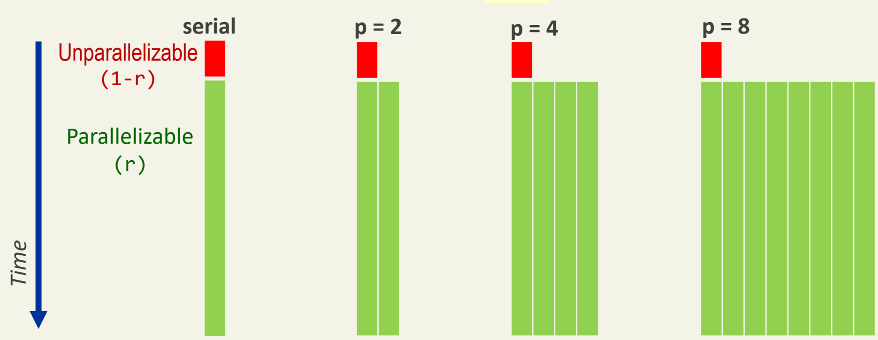
Gustafson's Law

Problems with **large repetitive DATA SETS** can be efficiently parallelized.

- As the program processes more and more data, the portion of the serial workload becomes smaller and smaller

$$S = \frac{1}{1 - r + \frac{r}{p}}$$

  – As r → 100%, the max speedup is S → p



Analogy

Suppose a car is traveling between two cities, and has already spent one hour traveling at 30 km/h (*this is the unparallelizable part*)

Amdahl's Law:

- Assume the two cities are 60 km apart (the car already travelled half of them). No matter how fast you drive the second half, it is impossible to achieve an average speed higher than 60 km/h before reaching the second city.
  - Since it has already taken you 1 hour and you only have a distance of 60 km total; going infinitely fast you would only achieve 60 km/h.

Gustafson's Law :

- Given enough time and distance to travel (the cities are far far away from each other), the car's average speed can always eventually reach more than 60 km/h, no matter how long or how slowly it has already traveled.
  - e.g., the could achieve an average speed of 90km/h by driving at 150 km/h for one more hour.

Estimation

- Based on the formula $S = \frac{1}{1-r+\frac{r}{p}}$, we can estimate the required portion of parallelized code (r) given s and p:

$$r = \frac{1-\frac{1}{s}}{1-\frac{1}{p}}$$

- Example:
  - Assume the required speed up is 10, and we have 20 cores. How much of the program should be parallelizable?

$$r = \frac{1-\frac{1}{s}}{1-\frac{1}{p}} = \frac{1-\frac{1}{10}}{1-\frac{1}{20}} = \frac{0.90}{0.95} = 94.7\%$$

Scalability

- In general, a problem is scalable if it can handle ever increasing problem sizes.
- ***strongly scalable:*** problems where we keep E fixed after increasing *p* and without increasing problem size.
  - *p* ↑ + problem_size ***fixed*** = E fixed
  - We are achieving higher speedup with more cores, with a fixed E and fixed problem size.
- ***weakly scalable:*** problems where we keep E fixed after increasing p and ***with*** increasing the problem size.
  - *p* ↑ + problem_size ↑ = E fixed
  - Problem size is increased at the same rate as we increase p.

Remember!

- Speedup and efficiency are different
- Using more cores doesn't necessarily mean significant speedup
- Sometimes serial code is the best choice
  - Especially for serial algorithms that do not parallelize well
  - You need to test on your system
- As the problem size increases, both efficiency and speedup increase
  - Parallel programs are better used for larger problems

Conclusion/Up Next

- What we covered today (review key concepts):
  - Speedup and Efficiency
  - Amdahl’s and Gustafson’s Laws

- Next:
  - Intro to GPU programming
    - CPU vs GPU programming
    - Latency vs. Throughput
  - CUDA basics: the hardware layout