14013204-3 - PARALLEL COMPUTING

14013204-3 - Parallel Computing (3 credits)

- Course Description
  - This course examines the theory and practice of parallel computing.
- Topics covered:
  - Introduction to Parallel computing.
  - Parallel architectures.
  - Designing parallel program algorithms, and managing different kinds of parallel programming overheads, e.g., synchronization, communication, etc. Measuring and Tuning the parallel performance.
  - Programming for shared and distributed parallel architectures.
- Prerequisites
  - 14012203-4 Operating Systems,
  - 14012401-3 Data Structures

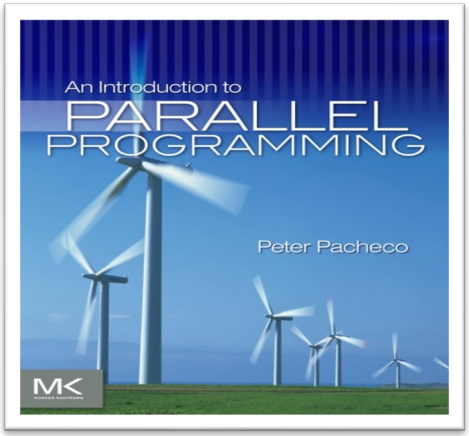
14013204-3 - Parallel Computing (3 credits)

- Course Weekly Hours
  - (2 lec + 2 lab)/week
- Textbook/References
  - Introduction to Parallel Programming, Peter Pacheco - Matthew Malensek, 2022.
  - Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms, Roman Trobec - Boštjan Slivnik - Patricio Bulić - Borut Robič, 2018.

14013204-3 - Parallel Computing (3 credits)

- Assessment Methods
  - Quizzes & Participation: 15 %
  - Lab : 25 %
  - Midterm: 25 %
  - Final : 35 %

- There might be a change in this assessment methods.



Chapter 1

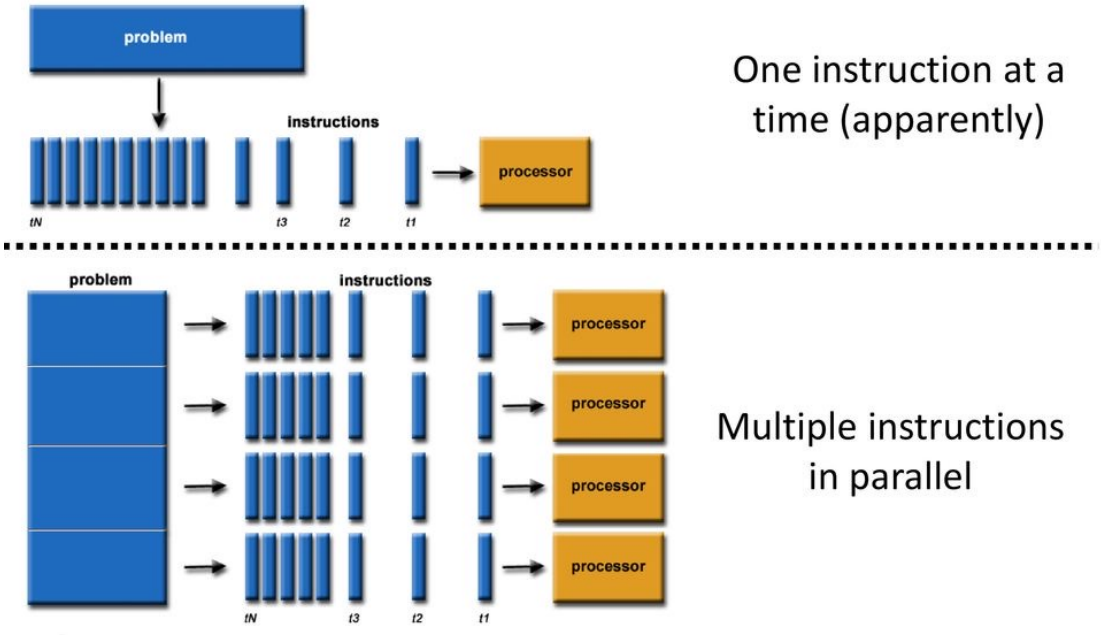
Why Parallel Computing?

Roadmap

- What is parallel Computing?
- How is performance achieved?
- Why we need ever-increasing performance?
- Why we need to write parallel programs?
- How do we write parallel programs?
- Type of Parallel Systems.
- What we'll be doing.
- Concurrent, parallel, distributed!

What is Parallel Programming?

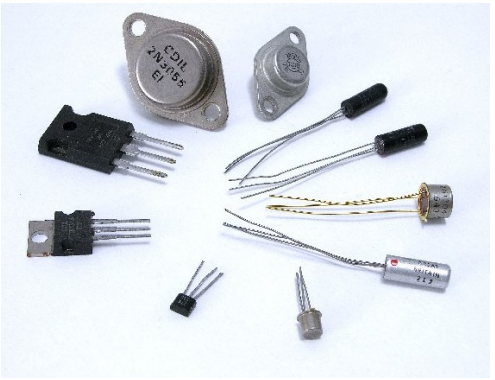
Serial vs. parallel program



- In general, most software will eventually have to make use of **parallelism** as performance matters.

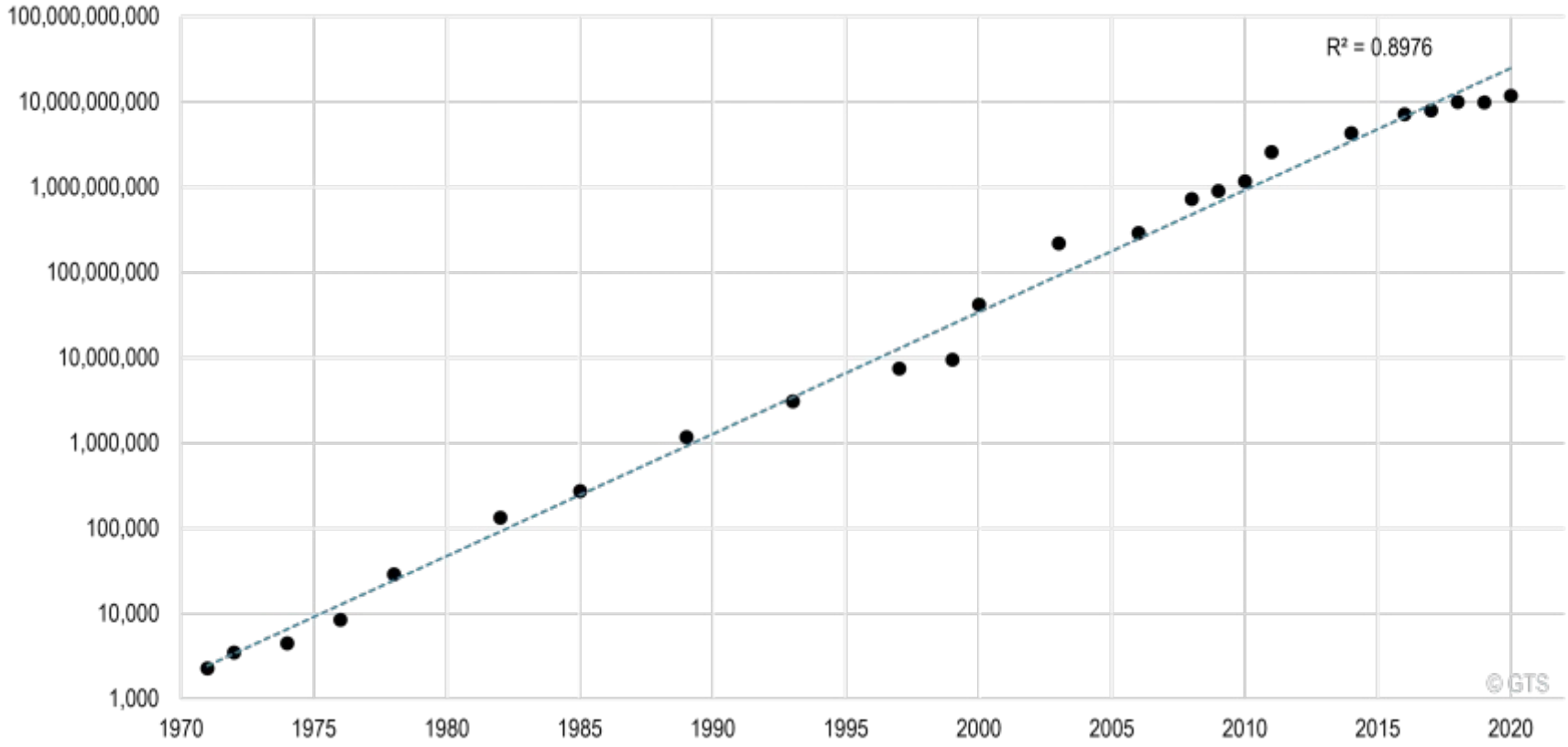
How is performance achieved?

- All processors are made of transistors.
  - The fundamental components of a CPU and play crucial roles in its operation.
  - Smaller transistors change state faster: they enable higher speeds.
  - Added advanced hardware that made your code faster automatically.



- **Moore's Law:** the number of transistors integrated into a single chip will **double** every two years since their invention, leading to an enormous increase in CPU performance and consequently to the application.

How is performance achieved?



How is performance achieved?

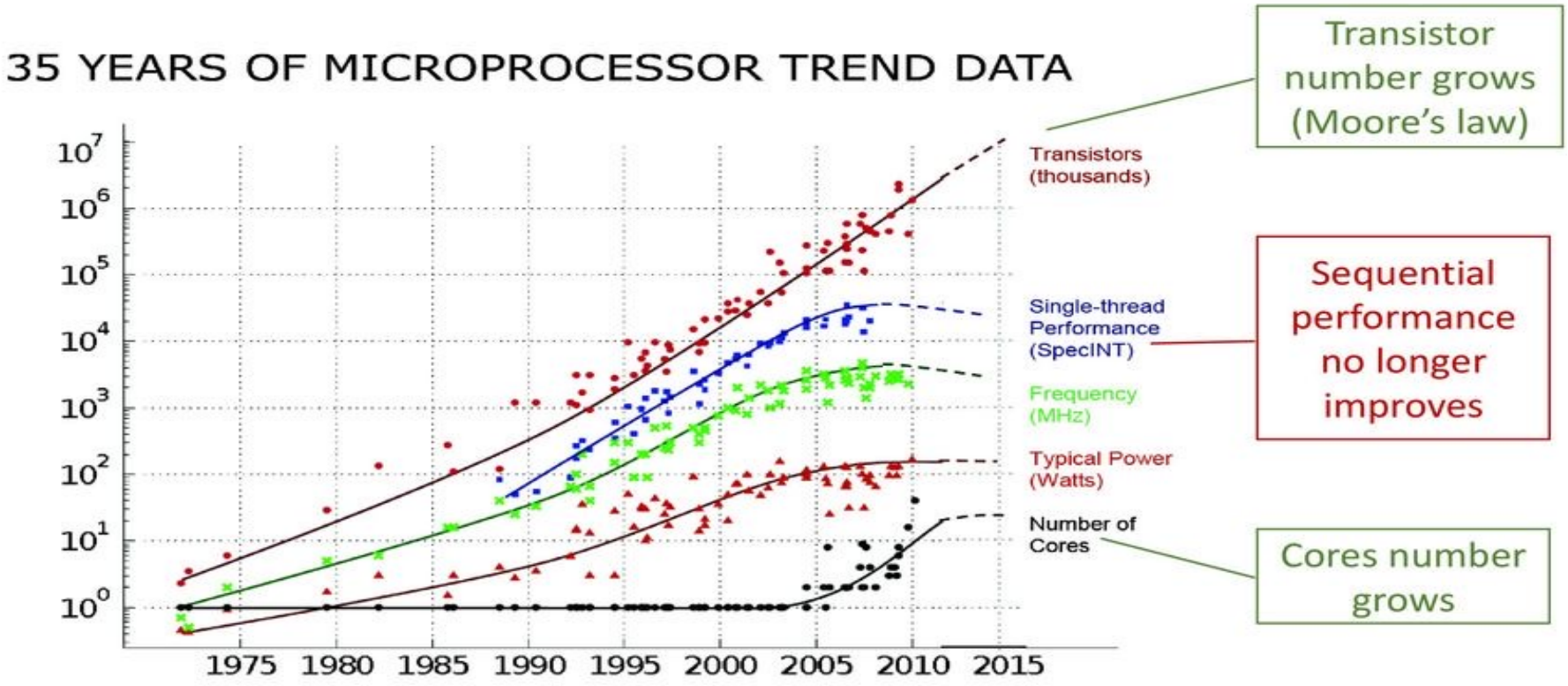
- **Smaller** transistors → **More** transistors on chips (increase in transistor density).
- More transistors on chips → **more computational** power (Faster processors) → **higher applications performance**.
- Each new generation of processors provides **more transistors** and offers **higher speed**.
- From 1986 – 2003, microprocessors were speeding like a rocket, **increasing in performance by an average of 50% per year**.
- This unprecedented increase meant that users and software developers could often simply wait for the next generation of microprocessors to obtain increased performance from their applications.
- BUT, this free performance gain is over around 2003-2004!

Changing times

- Since 2003, however, single-processor performance improvement has slowed to the point that in the period from 2015 to 2017, it increased at less than **4%** per year.
- The power of the conventional processor has reached the point where the processor's performance and speed are **limited** and can not be improved with the increase in transistors.
- Why ??

Changing times

Free lunch – is over ☹



Changing times

- However, as the speed of transistors increases, their **power consumption also increases**.
- Most of this power is dissipated as **heat**, and when an integrated circuit gets too hot, it becomes **unreliable**.
  - Faster processors → increased power consumption.
  - Increased power consumption → increased heat.
  - Increased heat → unreliable processors
- Dissipating (removing) the heat is requiring more and more sophisticated equipment, heat sinks cannot do it anymore.
  - In the first decade of the twenty-first century, air-cooled integrated circuits reached the limits of their ability to dissipate heat. Therefore, it is becoming impossible to continue to increase the speed of integrated circuits. Indeed, in the last few years, the increase in transistor density has slowed dramatically.
- Therefore, it is becoming impossible to continue to increase the speed of integrated circuits.

Changing times

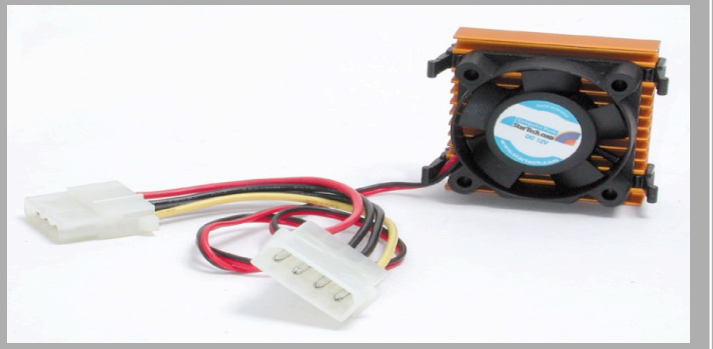
- **Let's look at some heatsinks:**

Intel 386 (25 MHz) Heatsink

- The 386 had no heatsink!
- It did not generate much heat
- Because it has very slow speed

Changing times

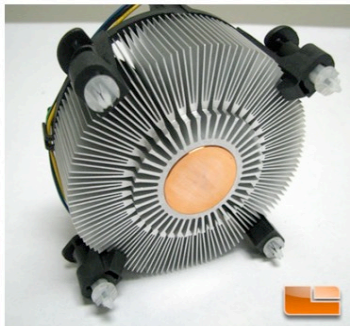
486 (~50Mhz) Heatsink



Pentium 4 (2-3GHz) Heatsink

Pentium 2/3 (233-733MHz) Heatsink

Core i7 (3-3.5GHz) Heatsink



Why we need ever-increasing performance

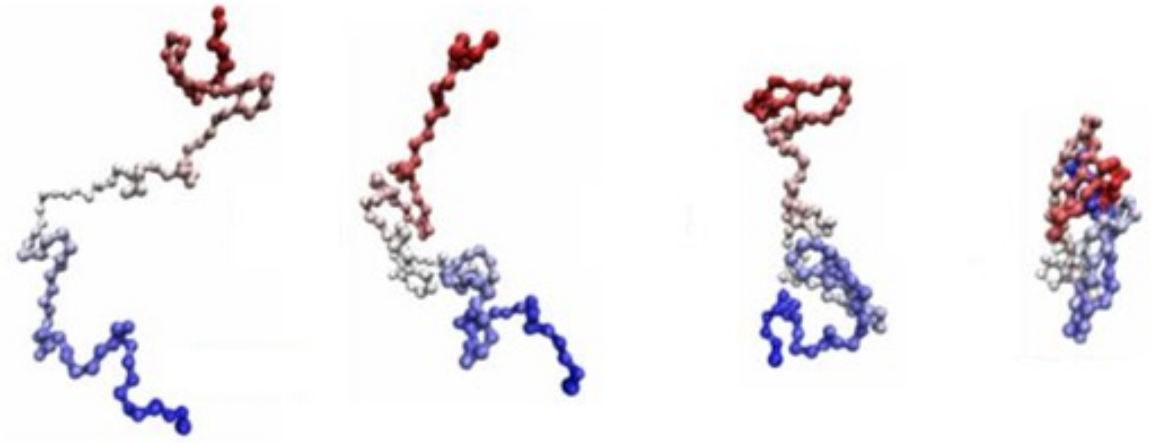
- Computational power is increasing, but so are our computation problems and needs.
- Problems we never dreamed of have been solved because of past increases, such **as decoding the human genome**, ever more **accurate medical imaging**, **astonishingly fast and accurate Web searches**, and ever **more realistic and responsive computer games** would all have been impossible without these increases
- More complex problems are still waiting to be solved.

Climate modeling

- To better understand climate change, we need far **more accurate computer models**, models that include **interactions** between the atmosphere, the oceans, solid land, and the ice caps at the poles. We also need to be able to make **detailed** studies of how various interventions might affect the global climate.

Protein folding

- It's believed that **misfolded proteins** may be involved in diseases such as Huntington's, Parkinson's, and Alzheimer's, but our ability to study configurations of **complex molecules** such as proteins is severely limited by our current computational power.



Drug discovery

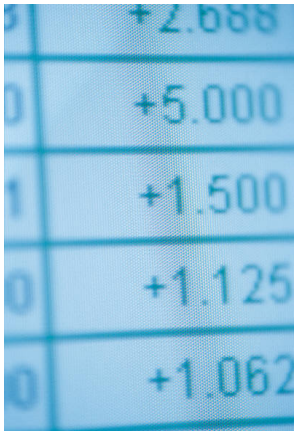
- There are many ways in which increased computational power can be used in **research into new medical treatments**. For example, there are many drugs that are effective in treating a relatively small fraction of those suffering from some disease. It's possible that we can **devise alternative treatments** by careful **analysis of the genomes** of the individuals for whom the known treatment is ineffective. This, however, will involve extensive computational analysis of genomes.

Energy research

- Increased computational power will make it possible to **program much more detailed models of technologies**, such as wind turbines, solar cells, and batteries. These programs may provide the information needed to construct far more efficient clean energy sources.

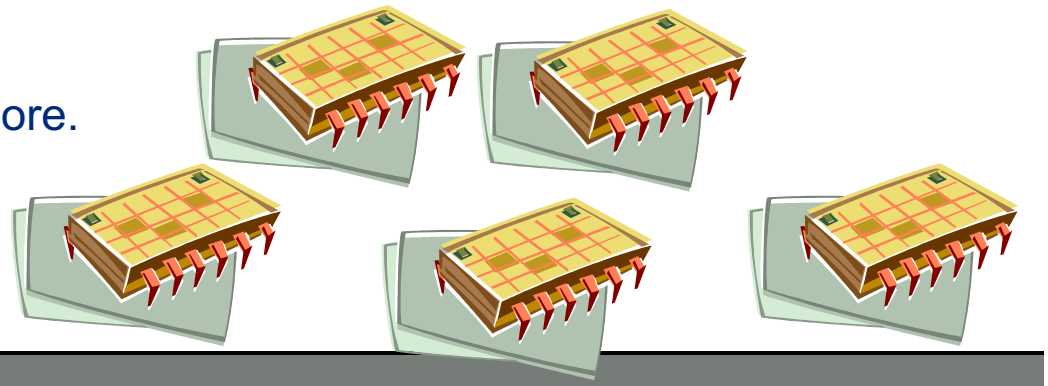
Data analysis

- We generate **tremendous amounts of data**. By some estimates, the quantity of data stored worldwide **doubles every two years**, but most of it is **largely useless unless it's analyzed**.
  - As an example, knowing the sequence of nucleotides in human DNA is, by itself, of little use. Understanding how this sequence affects development and how it can cause disease requires extensive analysis.
  - In addition to genomics, huge quantities of data are medical imaging, astronomical research, and Web search engines—to name a few.



Solution

- This difference in **performance increase** has been associated with a **dramatic change in processor design**.
- By 2005, most of the major manufacturers of microprocessors had decided that the road to rapidly increasing performance lay in the direction of **parallelism.**
- Instead of designing and building faster processors, put **multiple complete processors on a single integrated circuit**.
  - Move away from single-core systems to multicore processors.
  - "core" = central processing unit (CPU)
    - desktop and laptop 4 to 16 cores, even more.
    - Servers exceeding 64 cores



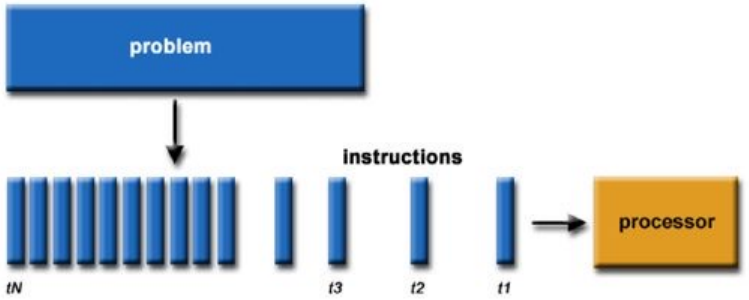
Why we need to write parallel programs

- This change has a very important consequence for software developers: Adding more processors doesn't help much if programmers aren't aware of them… or don't know how to use them.

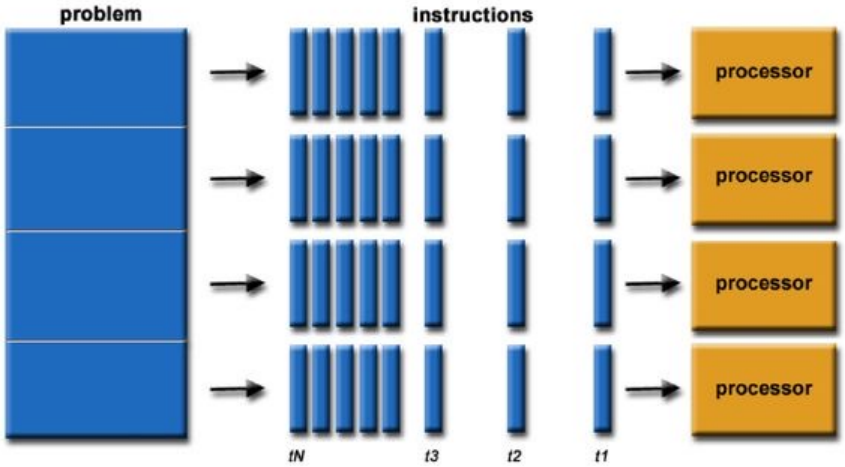
- **Serial programs (programs that were written to run on a single processor)**don't benefit from this approach (in most cases).
  - Such programs are **unaware of the existence of multiple processors**, and the performance of such a program on a system with multiple processors will be effectively the same as its performance on a single processor of the multiprocessor system.

Why we need to write parallel programs

Serial vs. parallel program



One instruction at a time (apparently)



Multiple instructions in parallel

Approaches to the serial problem

- Rewrite serial programs so that they're parallel, so that they can make use of multiple cores
- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.
- Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

Example

- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example (cont.)

- After each core completes execution of the code, is a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.

- Ex., 8 cores, n = 24, then the calls to Compute_next_value return: 1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9
- Then the values stores in my_sum will be:

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Example (cont.)

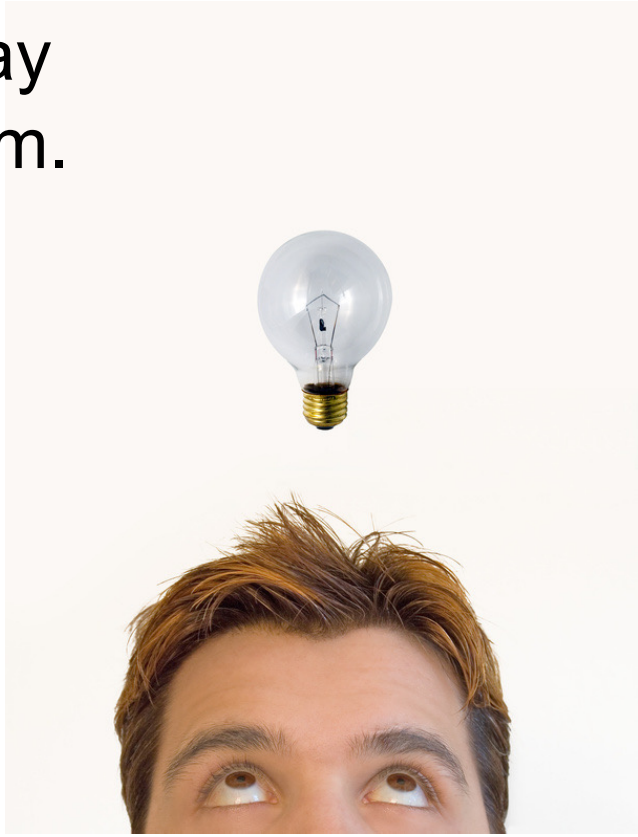
Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Global sum

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

But wait!

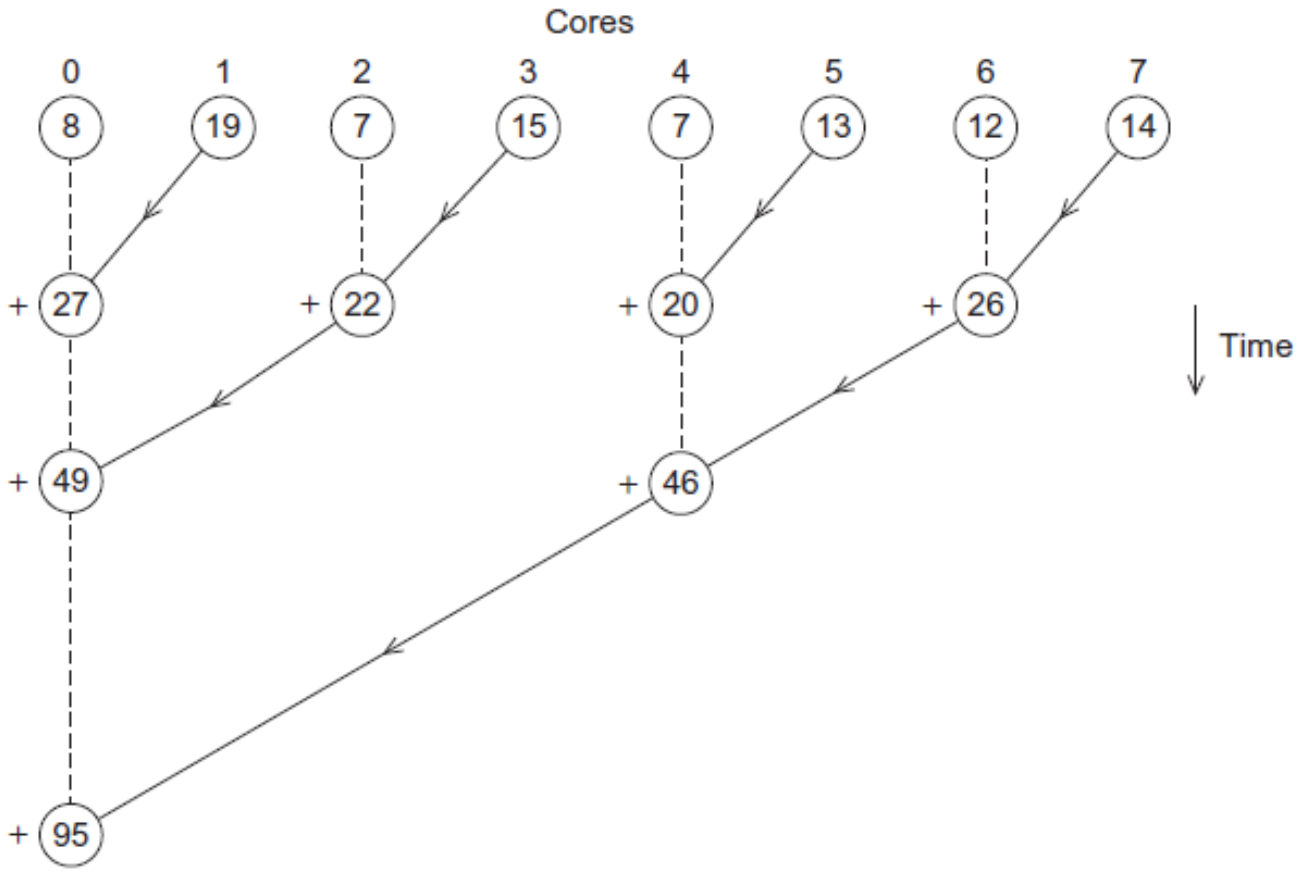
There's a much better way to compute the global sum.



Better parallel algorithm

- Don’t make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that :
  - Core 0 adds its result with core 1’s result.
  - Core 2 adds its result with core 3’s result, etc.
- Work with odd and even-numbered pairs of cores.
- Then we can repeat the process with only the even-ranked cores:
  - 0 adds in the result of 2,
  - 4 adds in the result of 6, and so on.
- Now cores divisible by 4 repeat the process, and so forth until core 0 has the final result.

Multiple cores forming a global sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
- The improvement is more than a factor of 2!

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.
- That's an improvement of almost a factor of 100!
- The first global sum is a fairly obvious **generalization of the serial global sum**: divide the work of adding among the cores, and after each core has computed its part of the sum, the master core simply repeats the basic serial addition—if there are p cores, then it needs to add p values.
- The second global sum bears little relation to the original serial addition.
- The point here is that it's **unlikely that a translation program would "discover" the second global sum.**

How do we write parallel programs?

- There are a number of possible answers to this question, but most of them depend on the basic idea of **partitioning the work** to be done among the cores.

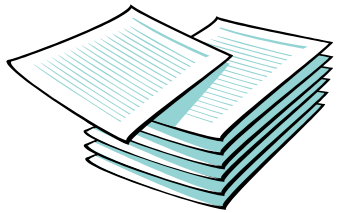
1. Task parallelism
   - Partition various tasks carried out solving the problem among the cores.

2. Data parallelism
   - Partition the data used in solving the problem among the cores.
   - Each core carries out similar operations on it's part of the data.

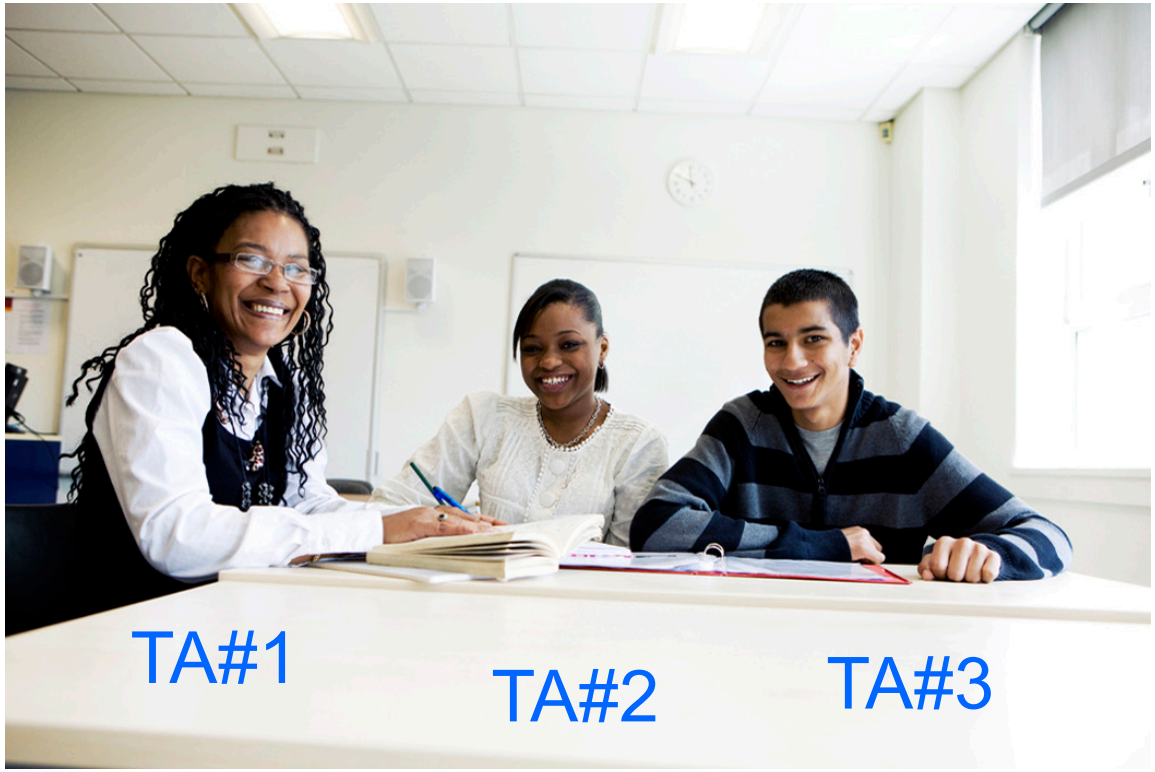
Professor P

15 questions

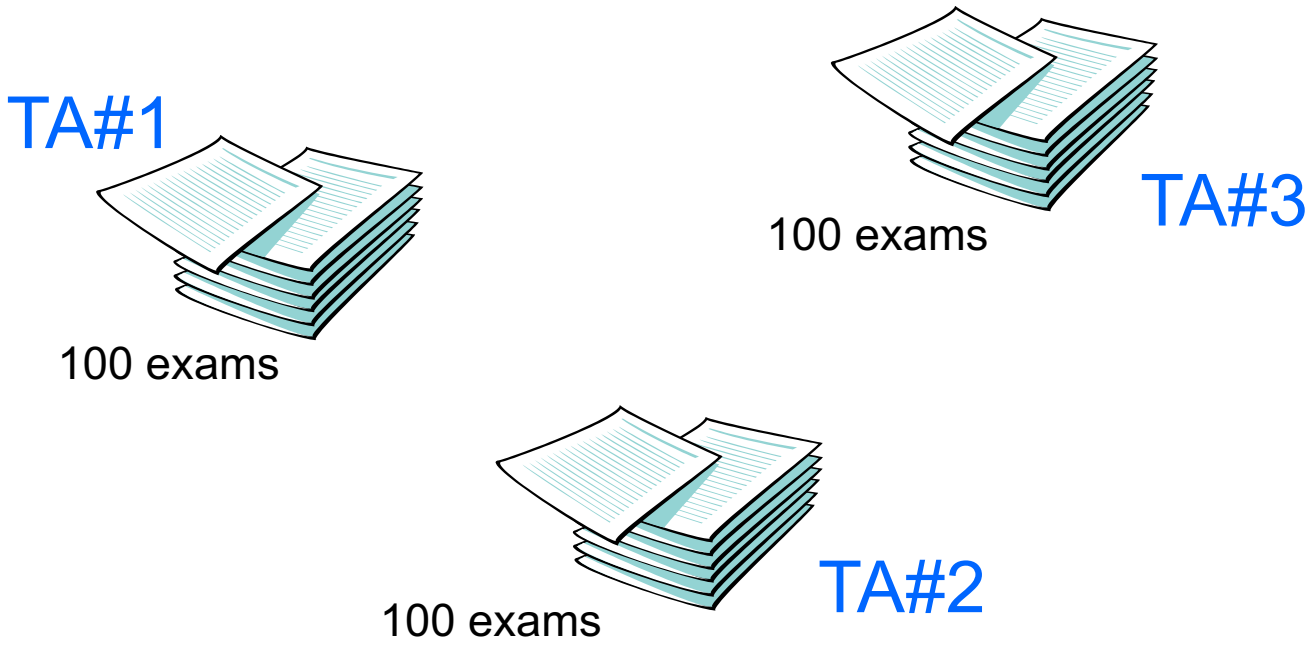
300 exams



Professor P's grading assistants



Division of work – data parallelism



Division of work – task parallelism

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

Division of work – data parallelism

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Division of work – task parallelism

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Tasks

1) Receiving
2) Addition

Coordination

- Cores usually need to coordinate their work.
  - Communication – one or more cores send their current partial sums to another core.
  - Load balancing – share the work evenly among the cores so that one is not heavily loaded.
  - Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

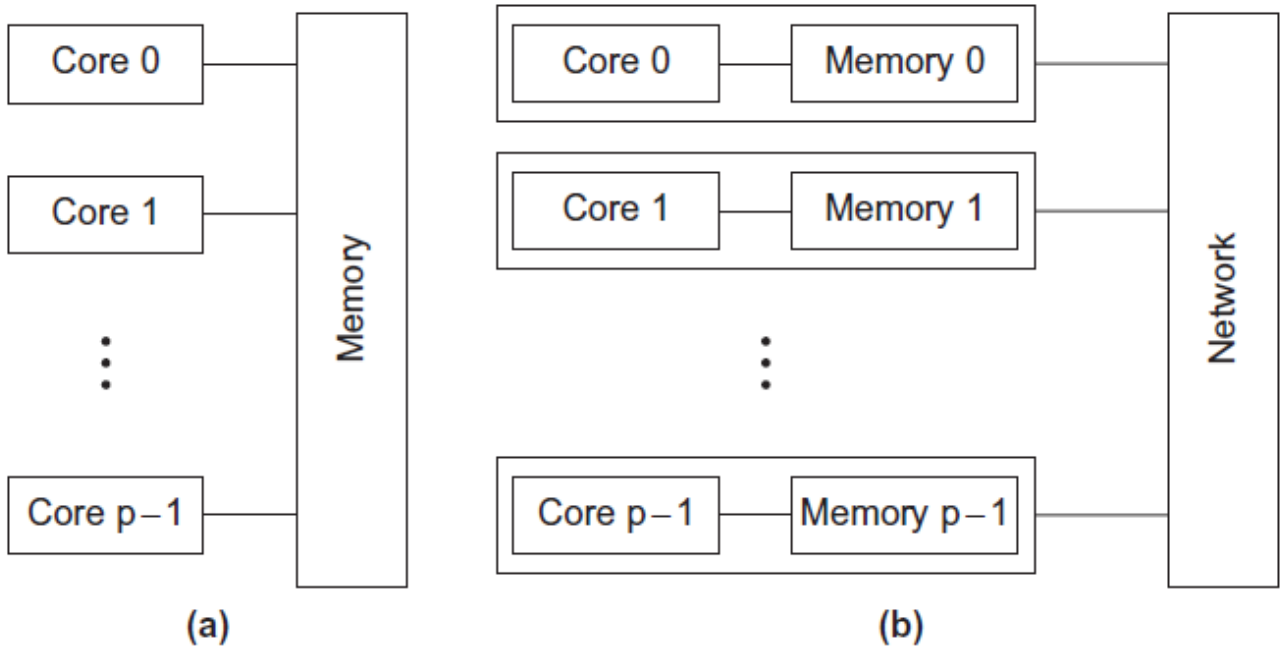
Explicit Vs. Implicit Parallelism

- **Explicit Parallelism**: Currently, the most powerful parallel programs are written using **explicit parallel constructs.**
  - that is, they are written using extensions to languages such as C, C++, and Java.
  - These programs include explicit instructions for parallelism:
  - E.g. core 0 executes task 0, core 1 executes task 1, . . . , all cores synchronize, . . . , and so on.
  - so such programs are often extremely complex.
- **Implicit Parallelism**: There are other options for writing parallel programs—for example, higher level languages.
  - They tend to sacrifice performance to make program development somewhat easier.

Type of parallel systems

- Shared-memory
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

Type of parallel systems



Shared-memory Distributed-memory

What we'll be doing

- Learning Parallel HW and SW.
  - Parallel architectures.
  - Parallel algorithms and coordination details.
  - Measuring the performance of parallel algorithms.
  - Etc.
- Learn to write programs that are explicitly parallel.
  - Will be using the C language.
  - Using three different extensions to C.
    - Message-Passing Interface (MPI) – Distributed Memory
    - OpenMP
    - CUDA

Terminology

- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.

Terminology

- So parallel and distributed programs are concurrent, but a program such as a multitasking operating system is also concurrent, even when it is run on a machine with only one core since multiple tasks can be in progress at any instant.
- There isn't a clear-cut distinction between parallel and distributed programs:
  - a parallel program usually runs **multiple tasks simultaneously on cores that are physically close to each other** and that either share the same memory or are connected by a very highspeed network.
  - On the other hand, distributed programs tend to be more "**loosely coupled**." The tasks may be executed by multiple computers that are separated by relatively large distances, and the tasks themselves are often executed by programs that were created independently.
  - As examples, our two concurrent addition programs would be considered parallel by most authors, while a Web search program would be considered distributed.

Concluding Remarks (1)

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically don’t benefit from multiple cores.
- Automatic parallel program generation from serial program code isn’t the most efficient approach to get high performance from multicore computers.

Concluding Remarks (2)

- How Performance achieved?
  - Before
    - Write a sequential (non-parallel) program.
    - It becomes faster with newer processors.
    - Higher speed, more advanced.
  - Now
    - New processor has more cores, but each is slower
    - Sequential programs will run slower on a new processor
    - They can only use one core
    - What will run a faster → Parallel program that can use all the cores!!!

Concluding Remarks (3)

- Learning to write parallel programs involves learning how to coordinate the cores.
- Parallel programs are usually very complex and therefore, require sound program techniques and development.
  - Many factors affect performance
  - Not easy to find the source of bad performance
  - Usually requires a deeper understanding of processor architectures
  - This is why there is a whole course for it