



CSCI-UA.0480-003

# Parallel Computing

## Lecture 1: Why Parallel Computing?

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>



# Who Am I?



- Mohamed Zahran (aka Z)
- Computer architecture/OS/Compilers Interaction
- <http://www.mzahran.com>
- Office hours: Tuesdays 2-4 pm  
– or by appointment
- Room: WWH 320

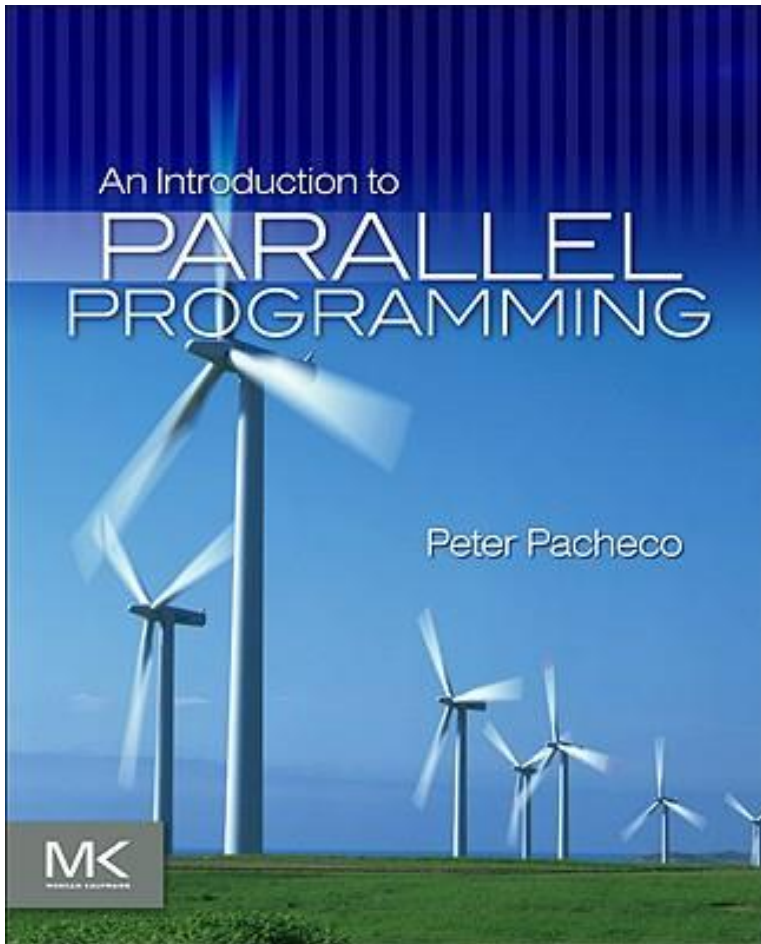
# Main Goals of this Course

- Why parallel computing is the **current** and **next** big thing?
- How does the parallel hardware look like?
- What are the challenges of parallel computing?
- How to write parallel programs and **make the best use of the underlying hardware?**

# *My wish list for this course:*

- Learn to think in parallel
- Make the best choice of hardware configuration and software tools/languages
- Be ready for the competitive market or for your next step in the academic/research ladder
- Learn how to progress way beyond the final exam!
- **Enjoy the course!**

# Textbook



- Author : Peter Pacheco
- Release Date: 2011
- Publisher: Morgan Kaufmann
- Print Book ISBN :  
9780123742605

# Course Components

- Lectures
  - Higher level concepts (slides + reading material)
- Homework assignments (20%)
  - The theoretical part
  - Usually due one week later
- Programming labs (20%)
  - 1-2 weeks each
  - Provide in-depth understanding of some aspect of systems
- One midterm exam (20%)
- One final exam (40%)

# Policies: Assignments

- You must work **alone** on all assignments
  - Post all questions on NYU classes forums,
  - You are encouraged to answer others' questions, but refrain from explicitly giving away solutions.
- Hand-ins
  - Submission through NYU classes
  - (-1) for each day of late submission **up to 3 days then zero** for the corresponding assignment/lab

# Policies:

The following excuses are not accepted and result in penalty:

- I tried to submit one minute after the deadline but I couldn't.
- Asking a question that has been asked and answered before in the forum.
  - Hint: Read Questions and Answers in the forum, even if you don't have any questions. You will learn a lot.
- I spent 100 hrs/week studying for this course, why didn't I get a high grade?
  - Do you really think that your grade is just a function of how much you study?
- What do I concentrate on when studying for the exam?
  - Do you really mean that some parts of the material are not important?



# Policies:

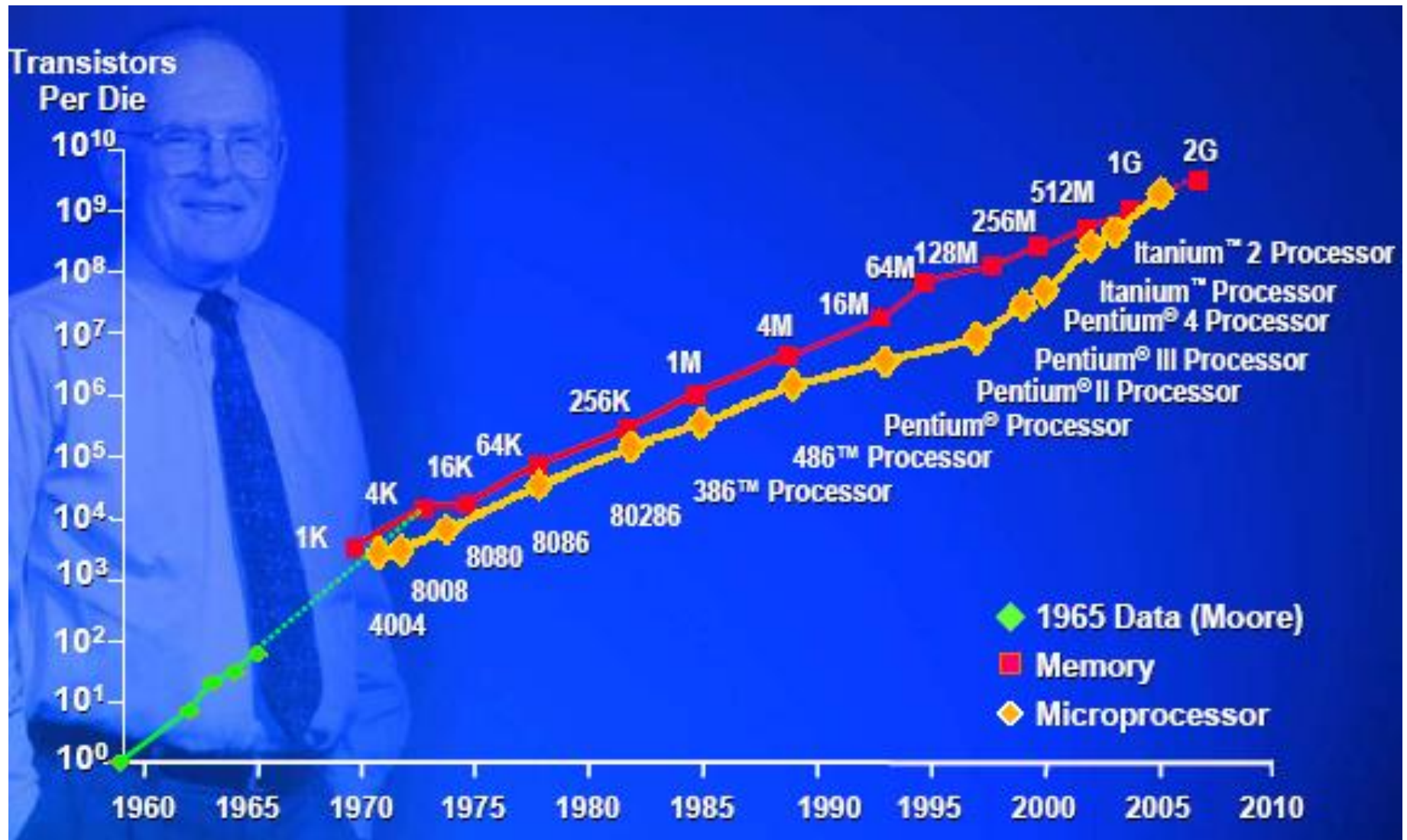
Arguing a grade of an assignment, lab, or exam.

You have **one week** from the time of receiving your grade to argue.

After that, no arguments are allowed.

Now, what is this story of  
parallel computing, multicore,  
multiprocessing, multi-this and  
multi-that?

# The Famous Moore's Law



It was implicitly assumed that more transistors per chip = more performance. **BUT ...**

# Effect of Moore's law

- ~1986 - 2002 → 50% performance increase
- Since 2002 → ~20% performance increase

Hmmm ...

- Why do we care? 20%/year is still nice.
- What happened at around 2002?
- Can't we have auto-parallelizing programs?

# Why do we care?

- More realistic games
- Decoding the human genome
- More accurate medical applications

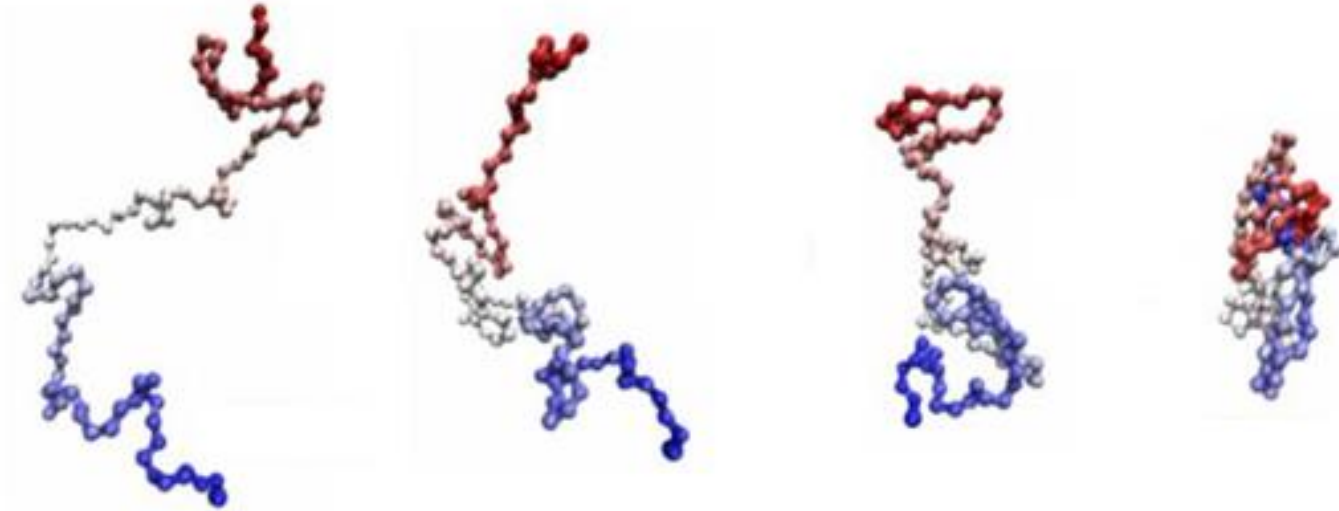
The list goes on and on ....

As our computational power increases → the number of problems we can seriously consider also increases.

# Climate modeling



# Protein folding



# Drug discovery



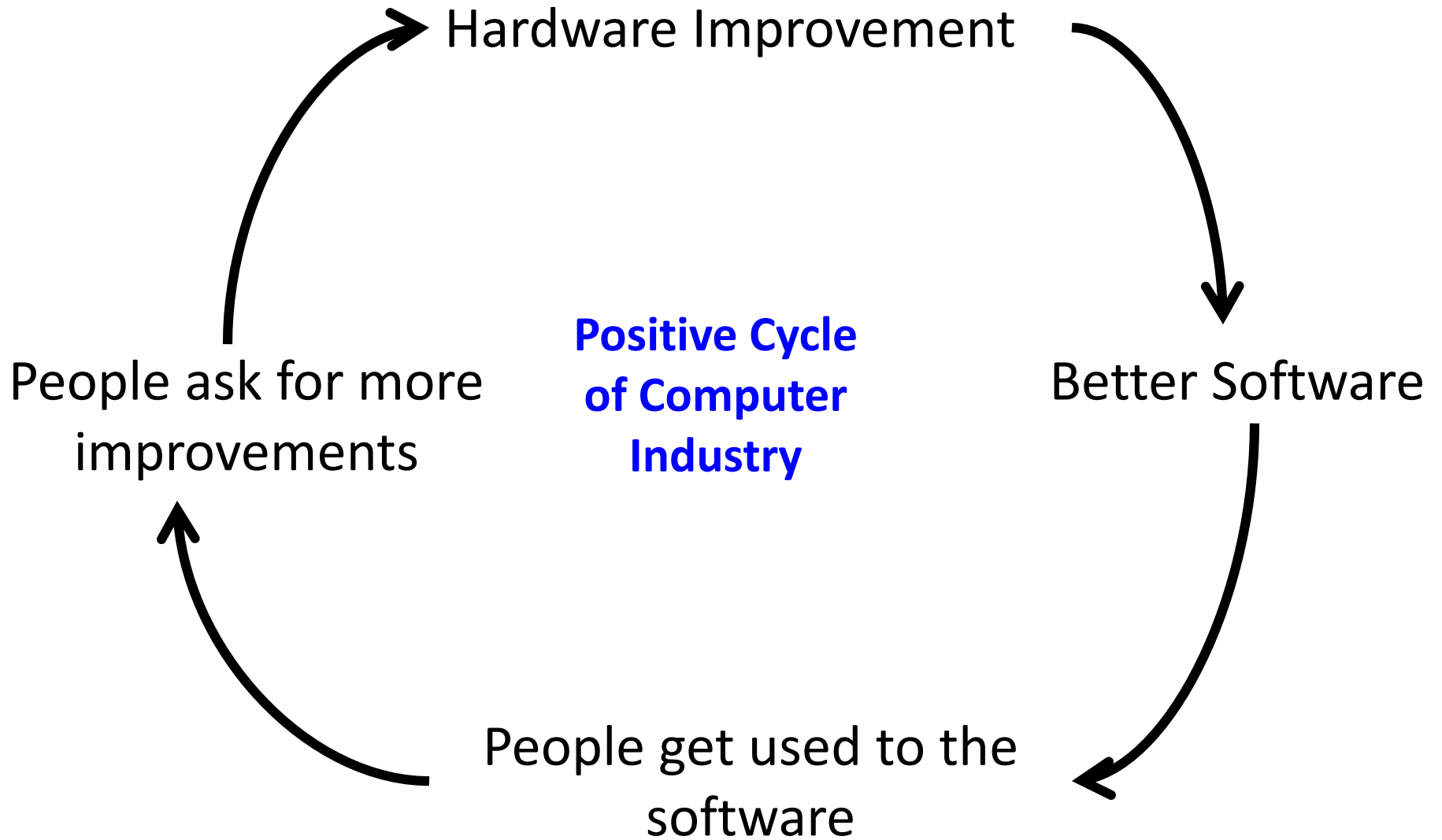


# Energy research



# Data analysis



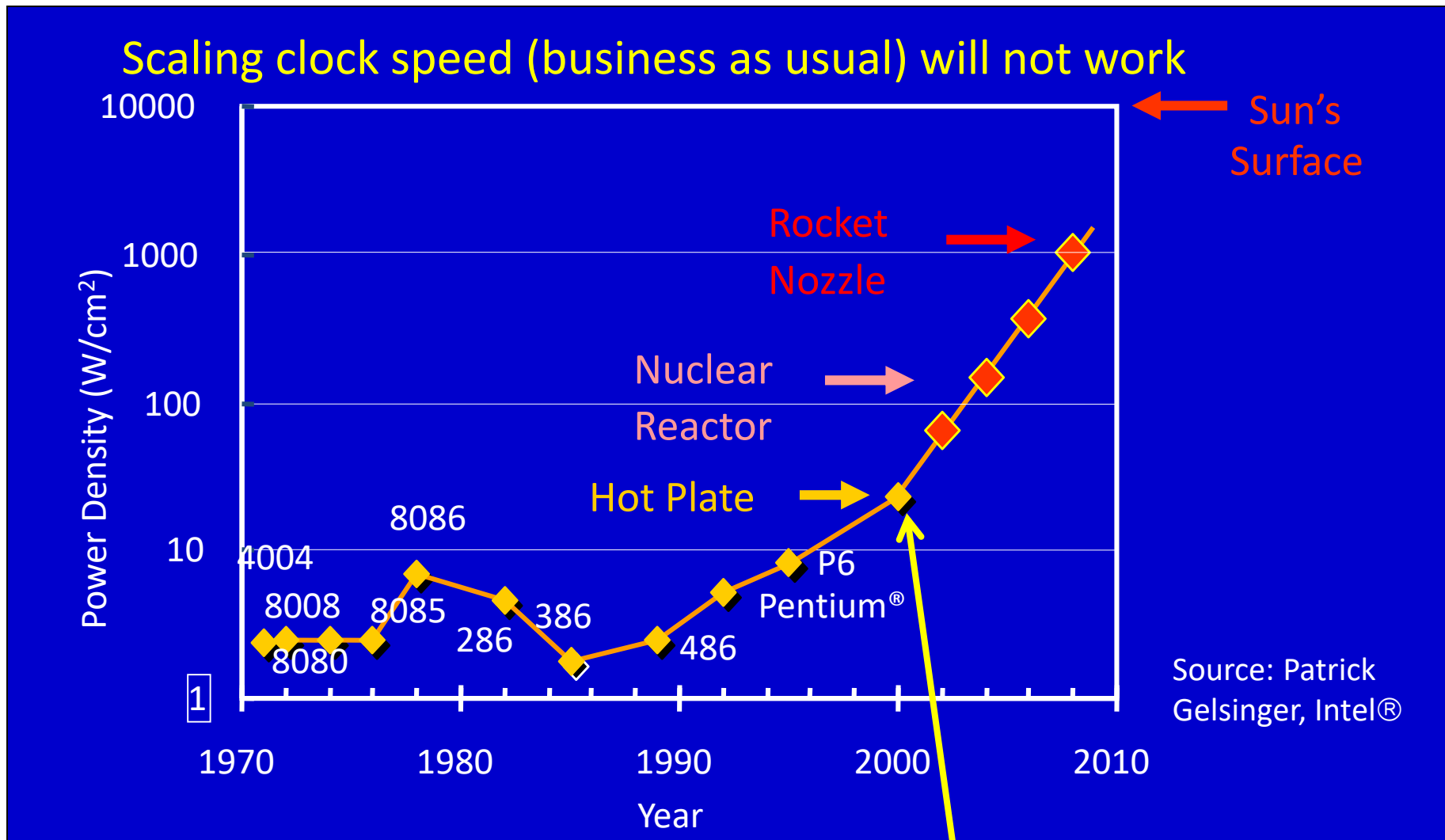


Why did we build parallel machines  
(and continue to do so)?

(multicore, multiprocessors,  
multi-anything!)

# Power Density

Moore's law is giving us more transistors than we can afford!



This is what happened at around 2002!

# Multicore Processors Save Power

$$\text{Power} = C * V^2 * F$$

$$\text{Performance} = \text{Cores} * F$$

Let's have two cores

$$\text{Power} = 2 * C * V^2 * F$$

$$\text{Performance} = 2 * \text{Cores} * F$$

But decrease frequency by 50%

$$\text{Power} = 2 * C * V^2 / 4 * F / 2$$

$$\text{Performance} = 2 * \text{Cores} * F / 2$$



$$\text{Power} = C * V^2 / 4 * F$$

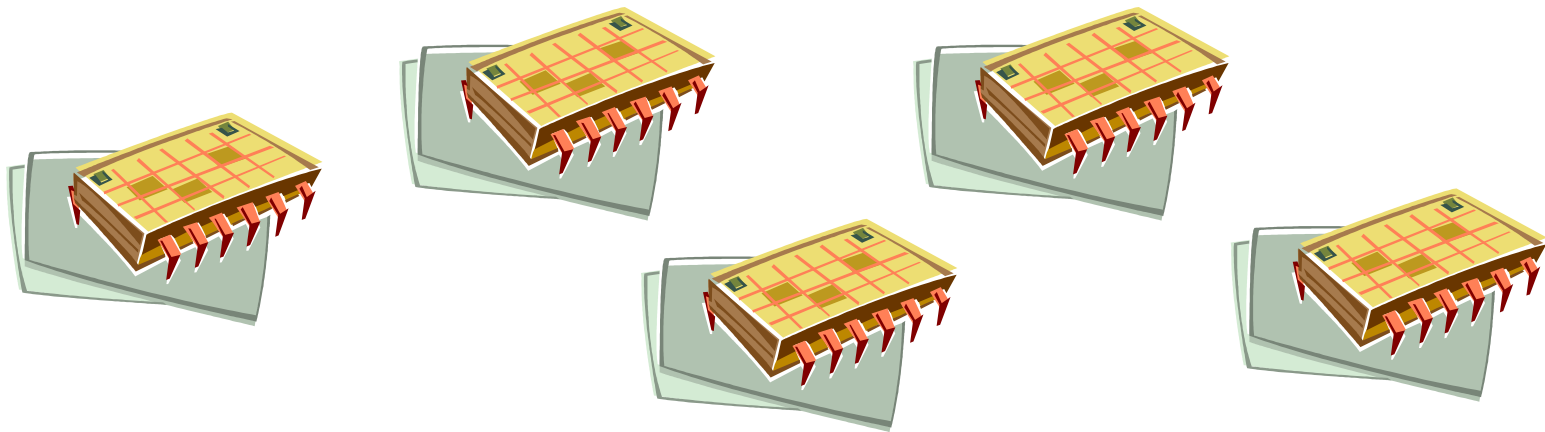
$$\text{Performance} = \text{Cores} * F$$

# A Case for Multiple Processors

- Can exploit different types of parallelism
- Reduces power
- An effective way to hide memory latency
- Simpler cores
  - = easier to design and test
  - = higher yield
  - = lower cost

# An intelligent solution

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.





# Now it's up to the programmers

- Adding more processors doesn't help much if programmers aren't aware of them...
- ... or don't know how to use them.
- Serial programs don't benefit from this approach (in most cases).



# The Need for Parallel Programming

**Parallel computing:** using multiple processors in parallel to solve problems more quickly than with a single processor

Examples of parallel machines:

- A cluster computer** that contains multiple PCs combined together with a high speed network
- A shared memory multiprocessor (SMP)** by connecting multiple processors to a single memory system
- A Chip Multi-Processor (i.e. multicore) (CMP)** contains multiple processors (called cores) on a single chip

# Attempts to Make Multicore Programming Easy

- **1<sup>st</sup> idea:** The right computer language would make parallel programming straightforward
  - **Result so far:** Some languages made parallel programming easier, but none has made it as fast, efficient, and flexible as traditional sequential programming.

# Attempts to Make Multicore Programming Easy

- **2<sup>nd</sup> idea:** If you just design the hardware properly, parallel programming would become easy.
  - **Result so far:** no one has yet succeeded!

# Attempts to Make Multicore Programming Easy

- **3<sup>rd</sup> idea:** Write software that will automatically parallelize existing sequential programs.
  - **Result so far:** Success here is inversely proportional to the number of cores!

# Parallelizing a sequential program is not very easy!

- It is not about parallelizing every step of the sequential program.
- Maybe we need a totally new algorithm.
- Our parallelization strategy also depends on the software!

# 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 its own private variables and executes this block of code independently of the other cores.



## Example (cont.)

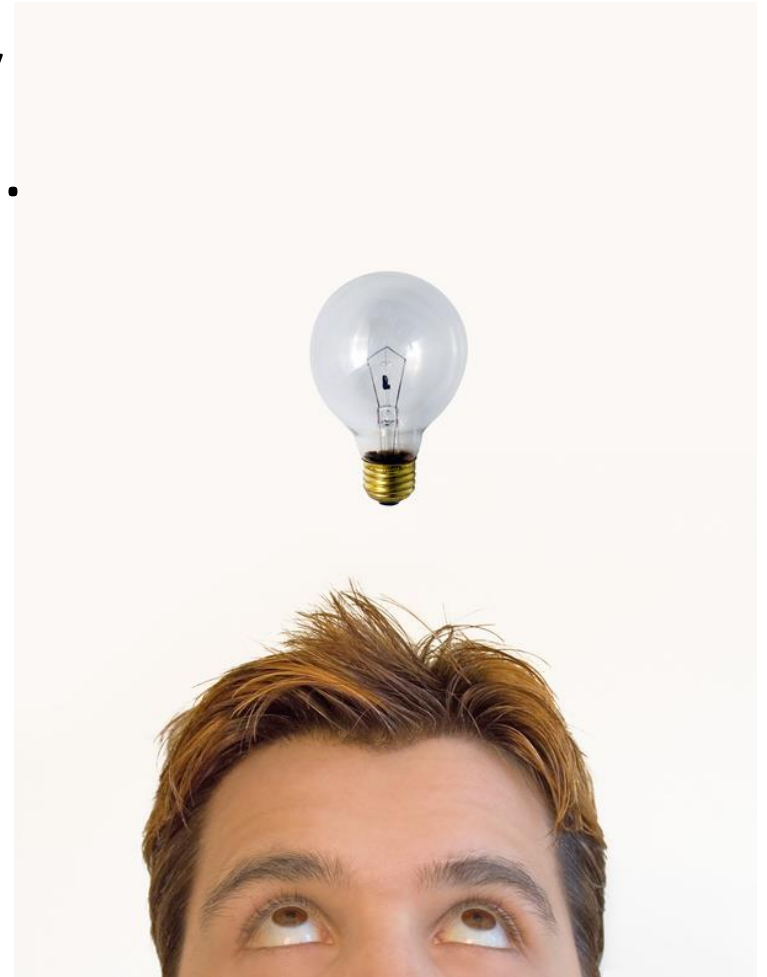
- Once all the cores are done computing their private `my_sum`, they form a global sum by sending results to a designated "master" core which adds the final result.

# Example (cont.)

```
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;  
}
```

But wait!

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



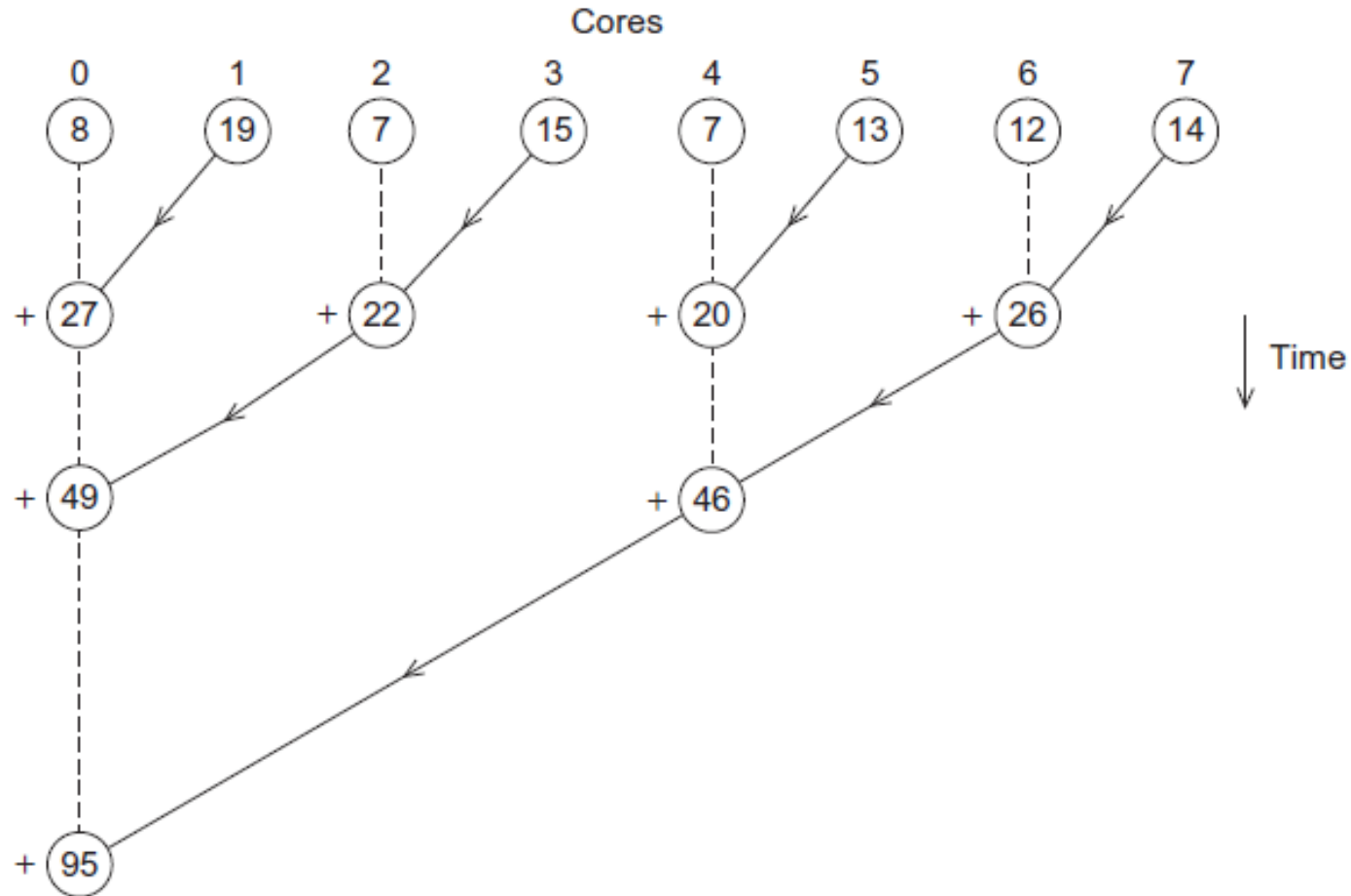
# Better parallel algorithm

- 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.

# Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.
  
- 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 version, the master core performs 7 receives and 7 additions.
- In the second version, 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!!



# Two Ways Of Thinking ... And one Strategy!

- Strategy: **Partitioning!**
- Two ways of thinking:
  - **Task-parallelism**
  - **Data-parallelism**
- Some constraints:
  - communication
  - load balancing
  - synchronization

# Conclusions

- Due to technology constraints, we moved to multicore processors.
- Parallel programming is now a must →  
The free lunch is over!
- There are different flavors of parallel hardware that we will discuss and also many flavors of parallel programming languages that we will deal with.