Contemporary C++: *Learning Modern C++ in a Modern Way*

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 23/24

Session 23. Introduction to Concurrency and Parallelism

- The Von Neumann Architecture
- Serial programming
- Moore’s Law
- The free lunch is over!
- Concurrency: definitions
- Parallel vs. Concurrent
- Threads: definitions
- C++11 threads: Single-threaded vs. Multi-threaded Hello, world!
- Launching threads
- Writing simple multi-threaded programs
- Q&A

150 min (incl. Q & A)

1

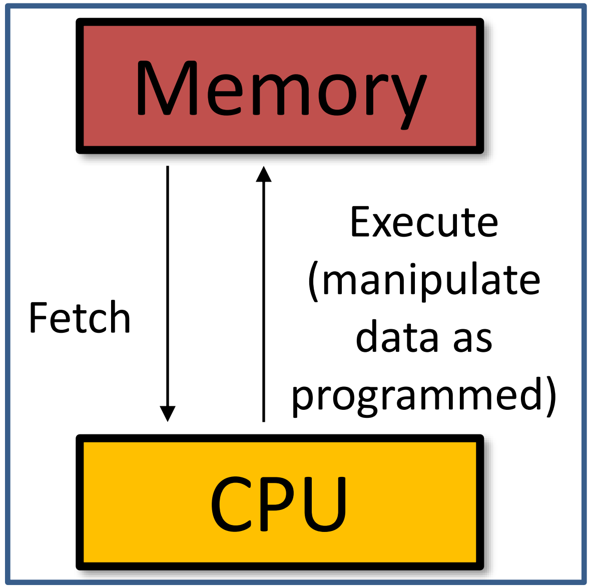
Von Neumann Architecture

Von Neumann Architecture

- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

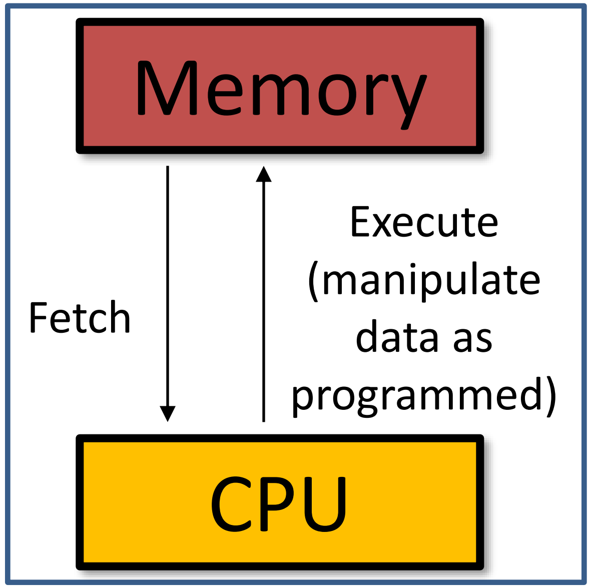
Von Neumann Architecture

- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.



Von Neumann Architecture

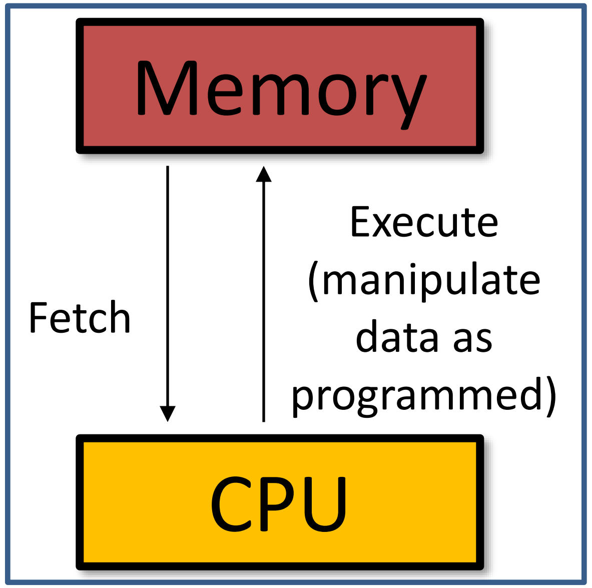
- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.



- Basic design:
  - Memory is used to store both program and data instructions.
  - Program instructions are coded data which tell the computer to do something.
  - Data is simply information to be used by the program.
  - A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

Von Neumann Architecture

- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

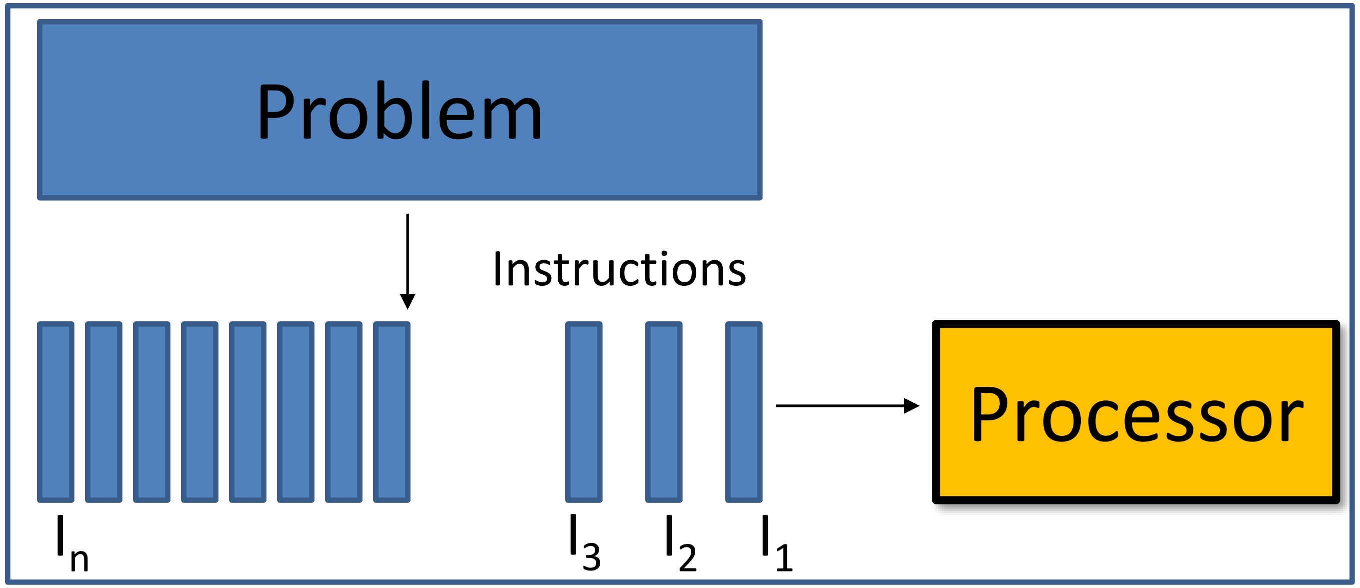


- Basic design:
  - Memory is used to store both program and data instructions.
  - Program instructions are coded data which tell the computer to do something.
  - Data is simply information to be used by the program.
  - A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

ALPHA آلفا

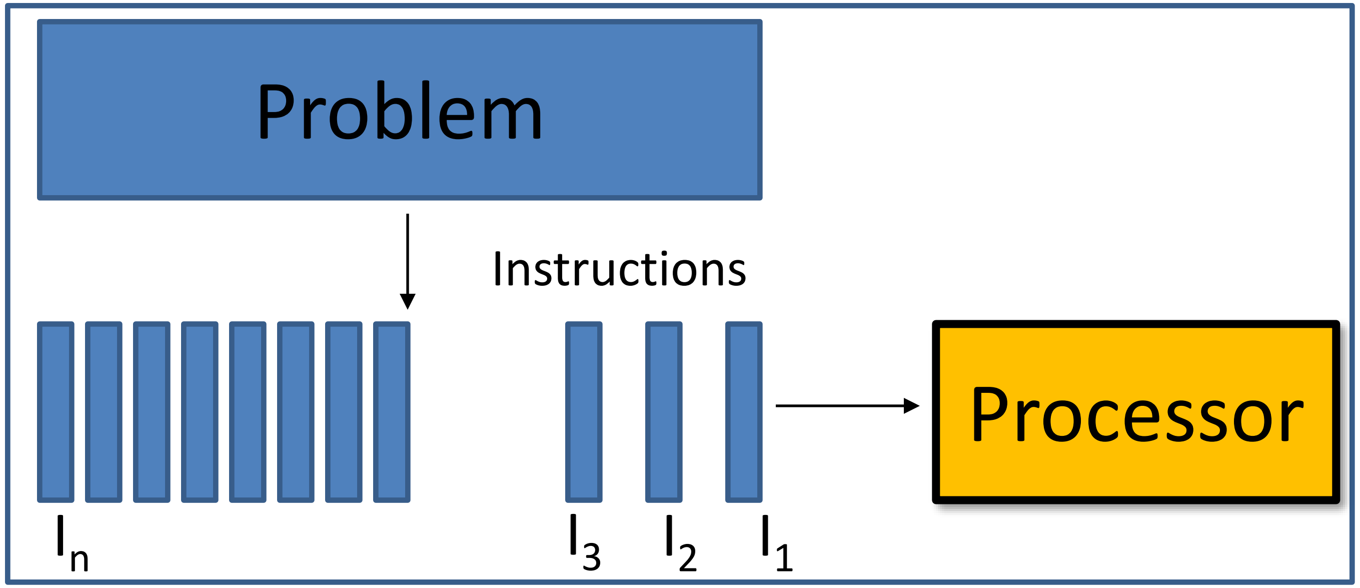
Serial programming

Serial programming



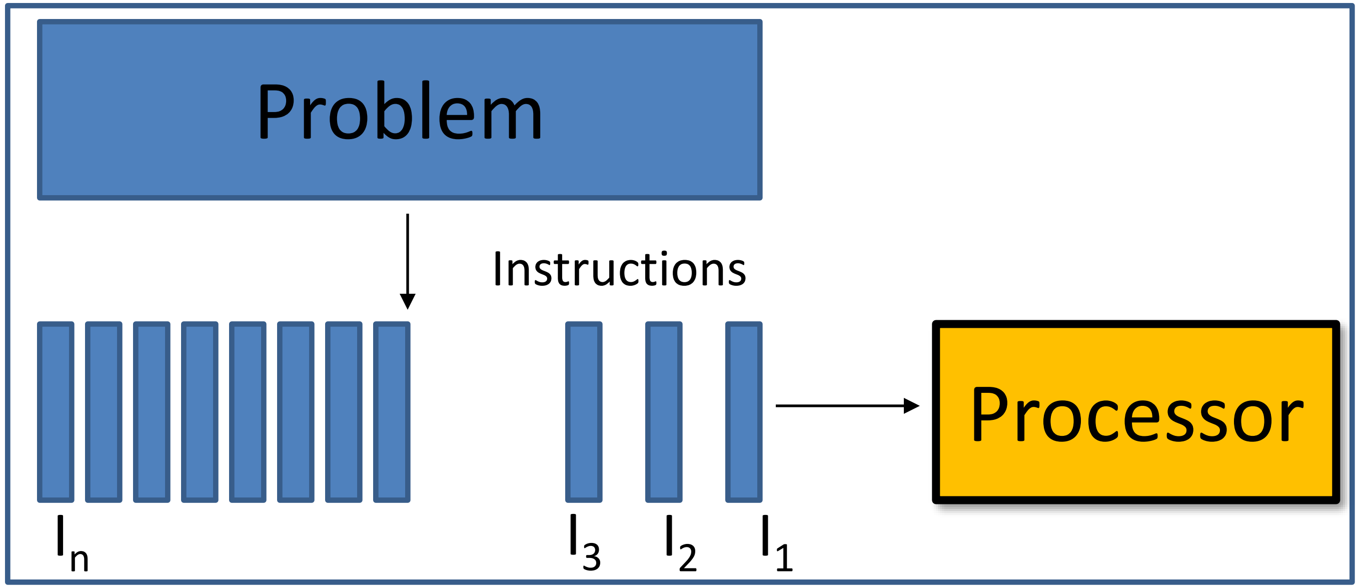
Serial programming

- Example: payroll system



Serial programming

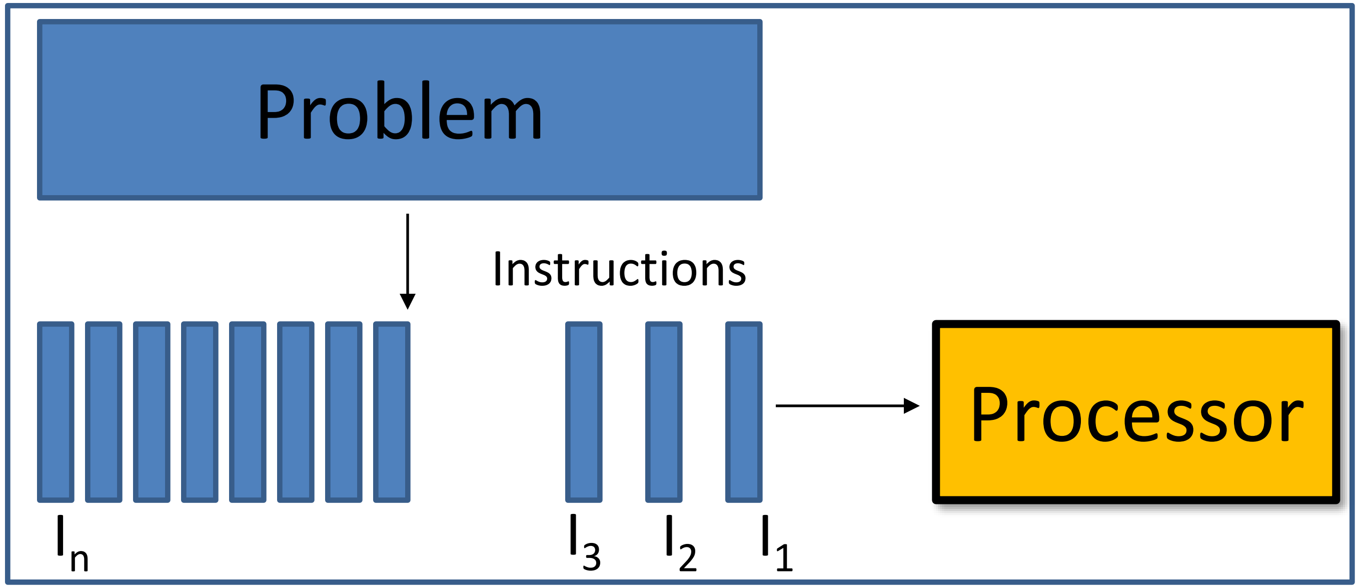
- Example: payroll system



- Benefits
  - It's simple to reason about
  - Serial programs are *deterministic*. Serial programs always do the same operations in the same order.
  - Debugging, verification, and testing of deterministic codes is easier.

Serial programming

- Example: payroll system



- Benefits
  - It's simple to reason about
  - Serial programs are *deterministic*. Serial programs always do the same operations in the same order.
  - Debugging, verification, and testing of deterministic codes is easier.

- Problems
  - Poor performance

Serial Trap

- Serialization is a learned skill that has been over-learned.
- Serial traps are constructs that make, often unnecessary, serial assumptions.

Serial Trap

- Serialization is a learned skill that has been over-learned.
- Serial traps are constructs that make, often unnecessary, serial assumptions.

```
for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

Serial Trap

- Serialization is a learned skill that has been over-learned.
- Serial traps are constructs that make, often unnecessary, serial assumptions.

```
for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

- Do something with a number of objects *one after the other*.

Serial Trap

- Serialization is a learned skill that has been over-learned.
- Serial traps are constructs that make, often unnecessary, serial assumptions.

```
for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

- Do something with a number of objects *one after the other*.

Serial Trap

- Serialization is a learned skill that has been over-learned.
- Serial traps are constructs that make, often unnecessary, serial assumptions.

```
for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

- Do something with a number of objects *one after the other*.

```
paralle_for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

Serial Trap

- Serialization is a learned skill that has been over-learned.
- Serial traps are constructs that make, often unnecessary, serial assumptions.

```
for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

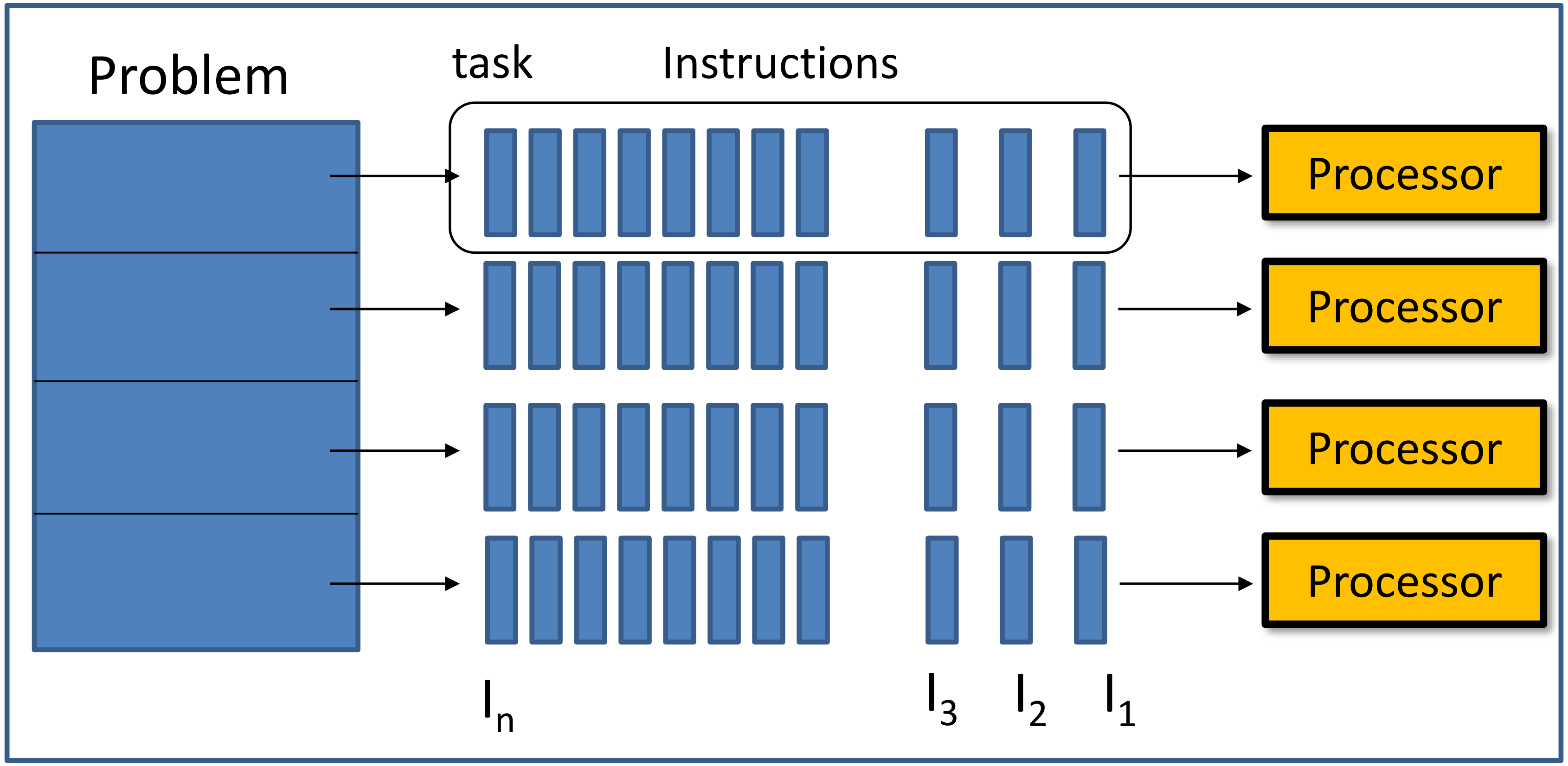
- Do something with a number of objects *one after the other*.

```
paralle_for (i = 0; i < number_web_sites; ++i) {
  search(searchphrase, website[i]);
}
```

- Do something with a number of objects.

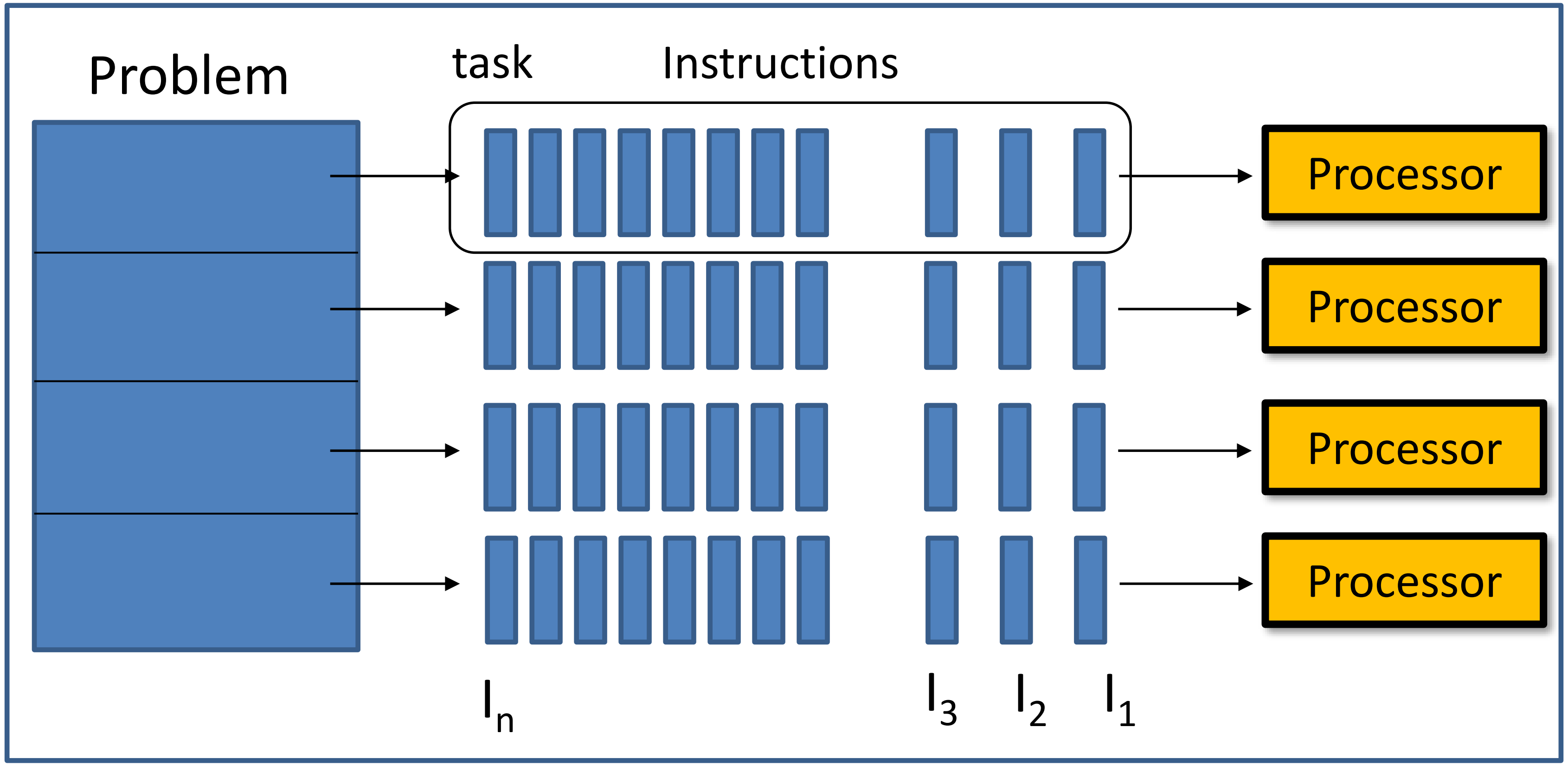
Parallel programming

Parallel programming



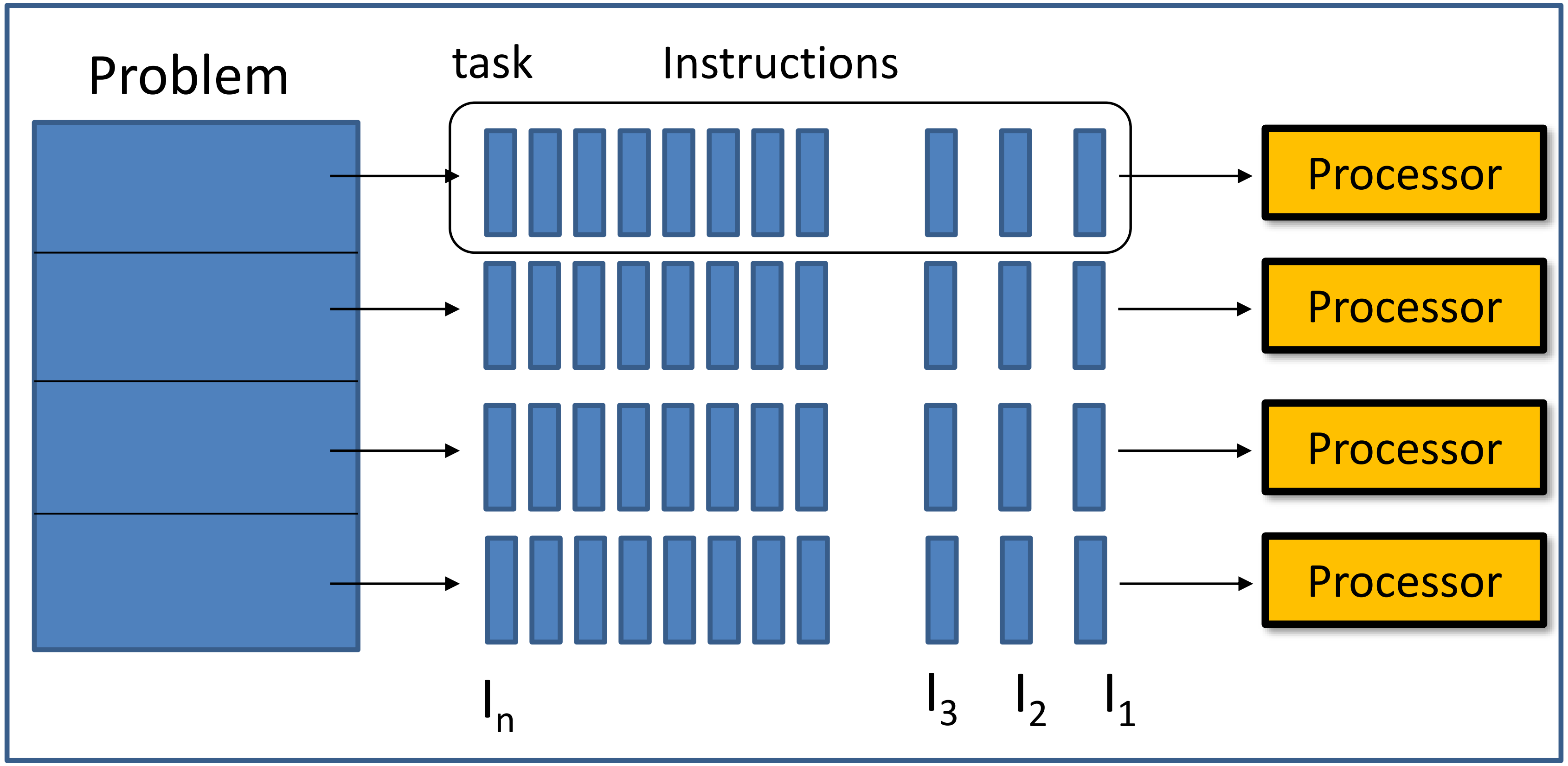
Parallel programming

- Analogy:
    Geometry: parallel lines
    Programming: parallel tasks



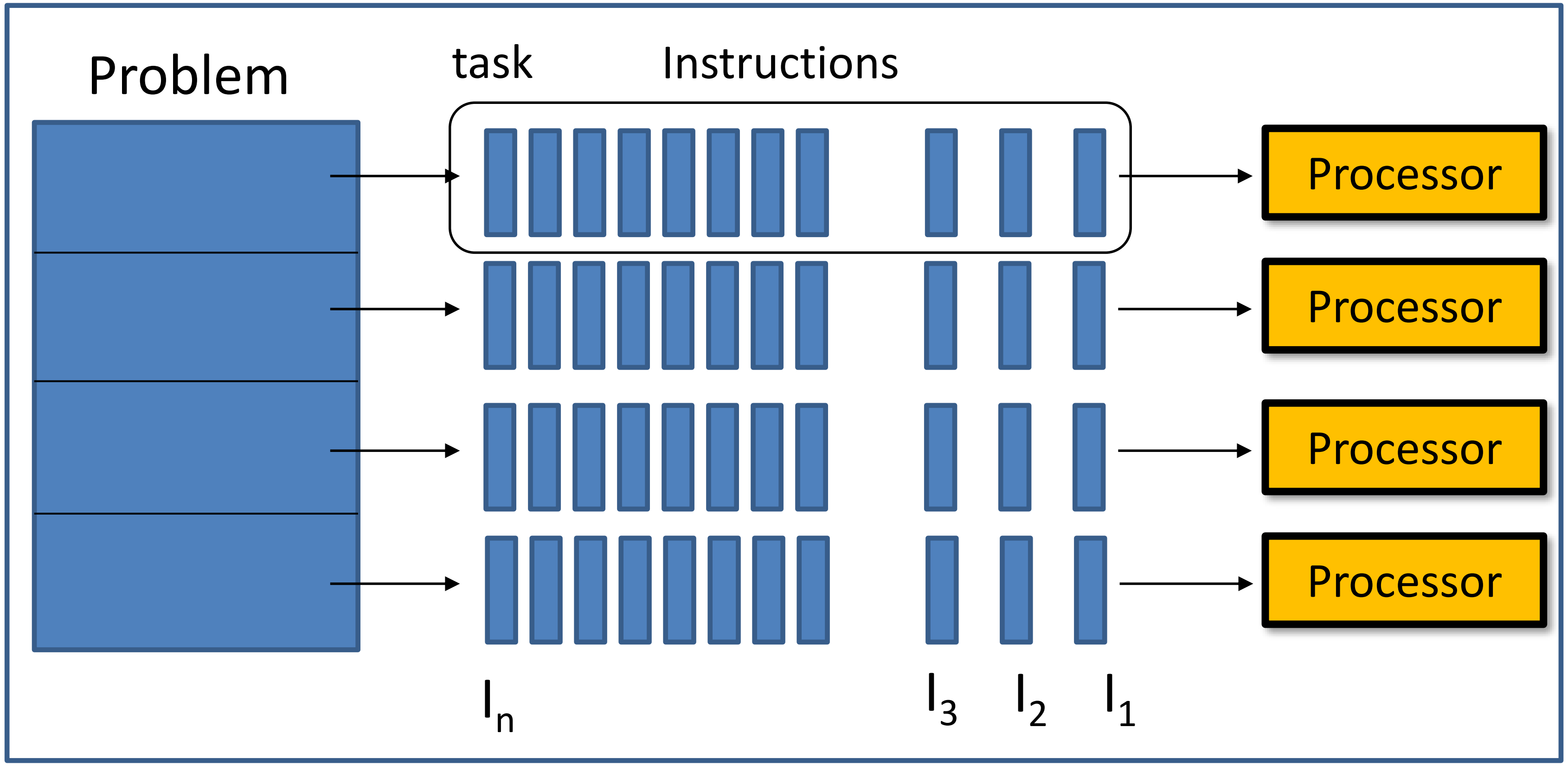
Parallel programming

- Analogy:
  Geometry: parallel lines
  Programming: parallel tasks
- Example: payroll system



Parallel programming

- Analogy:
  Geometry: parallel lines
  Programming: parallel tasks
- Example: payroll system
- Benefits
  - Improved performance



2

Moore's Law

Moore’s Law

- **Moore's law** is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. http://en.wikipedia.org/wiki/Moore%27s_law

Moore’s Law

- **Moore's law** is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. http://en.wikipedia.org/wiki/Moore%27s_law
- Moore’s law is an observation or projection rather than law.
- Exponential growth of *transistor densities* and as a result *clock speeds*.

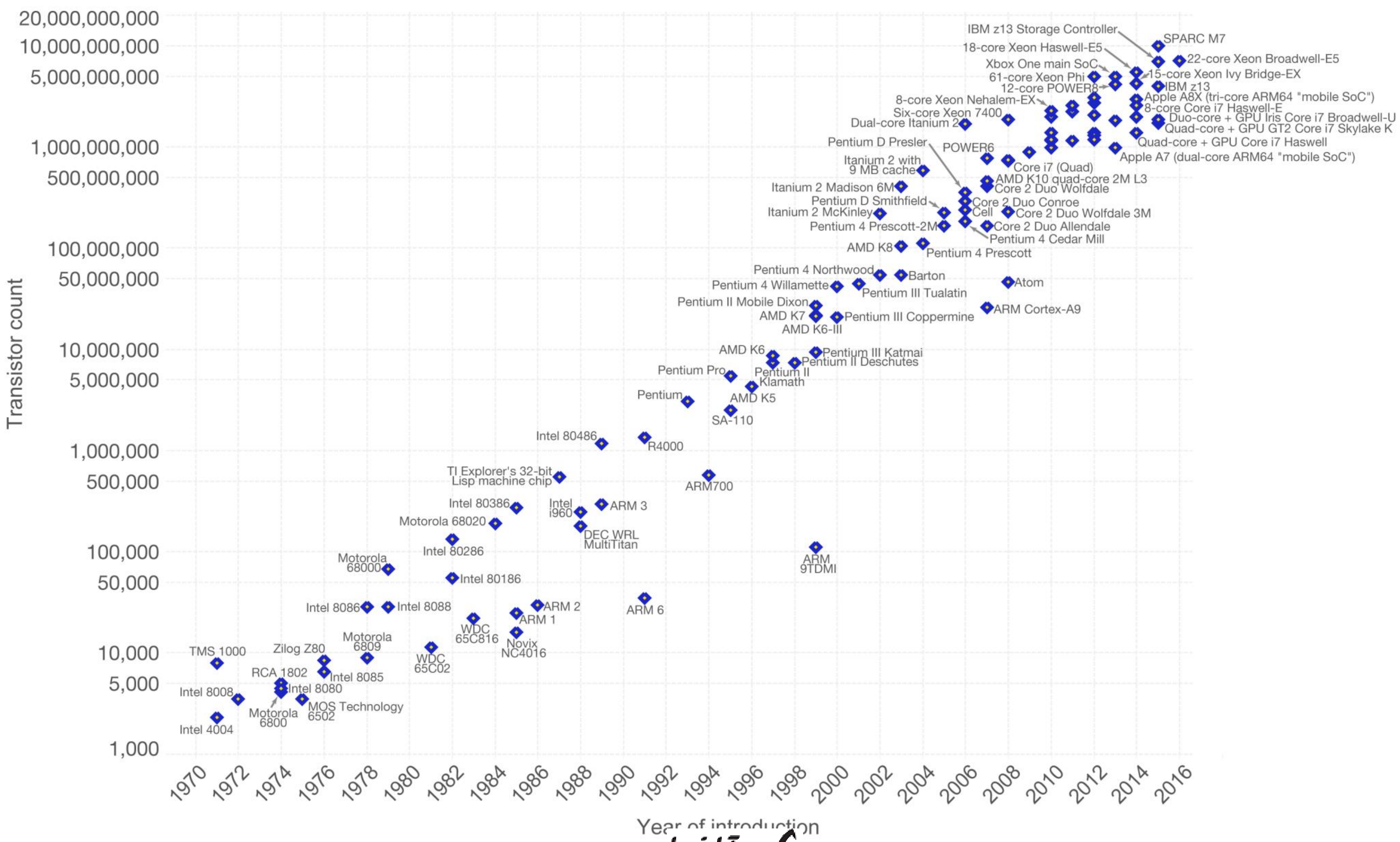
Moore’s Law

- **Moore's law** is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. http://en.wikipedia.org/wiki/Moore%27s_law
- Moore’s law is an observation or projection rather than law.
- Exponential growth of *transistor densities* and as a result *clock speeds*.
- 1965-1975: every year
- 1975-1985: every two years
- 2013- : every two and half a year

Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



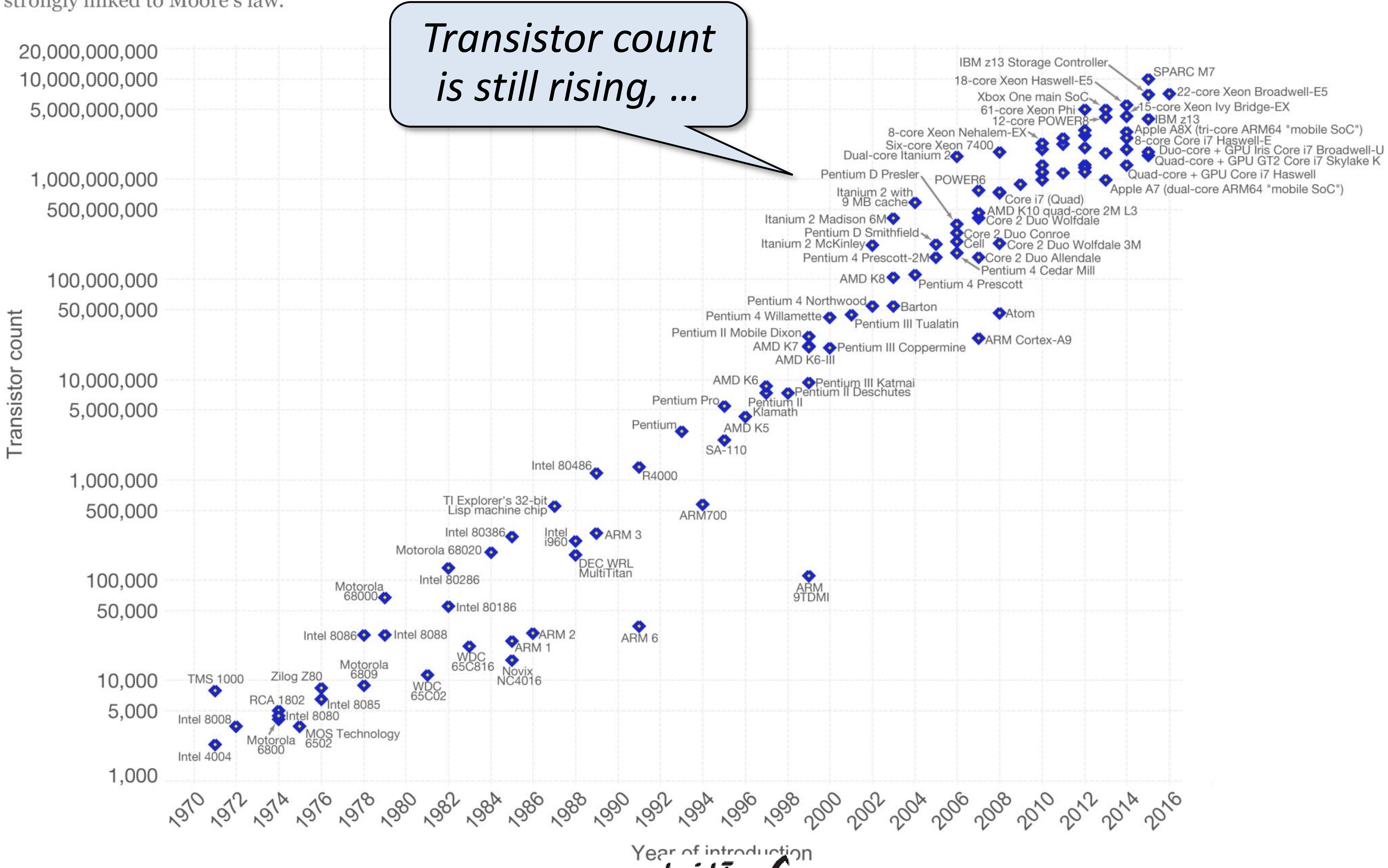
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations a topic.

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

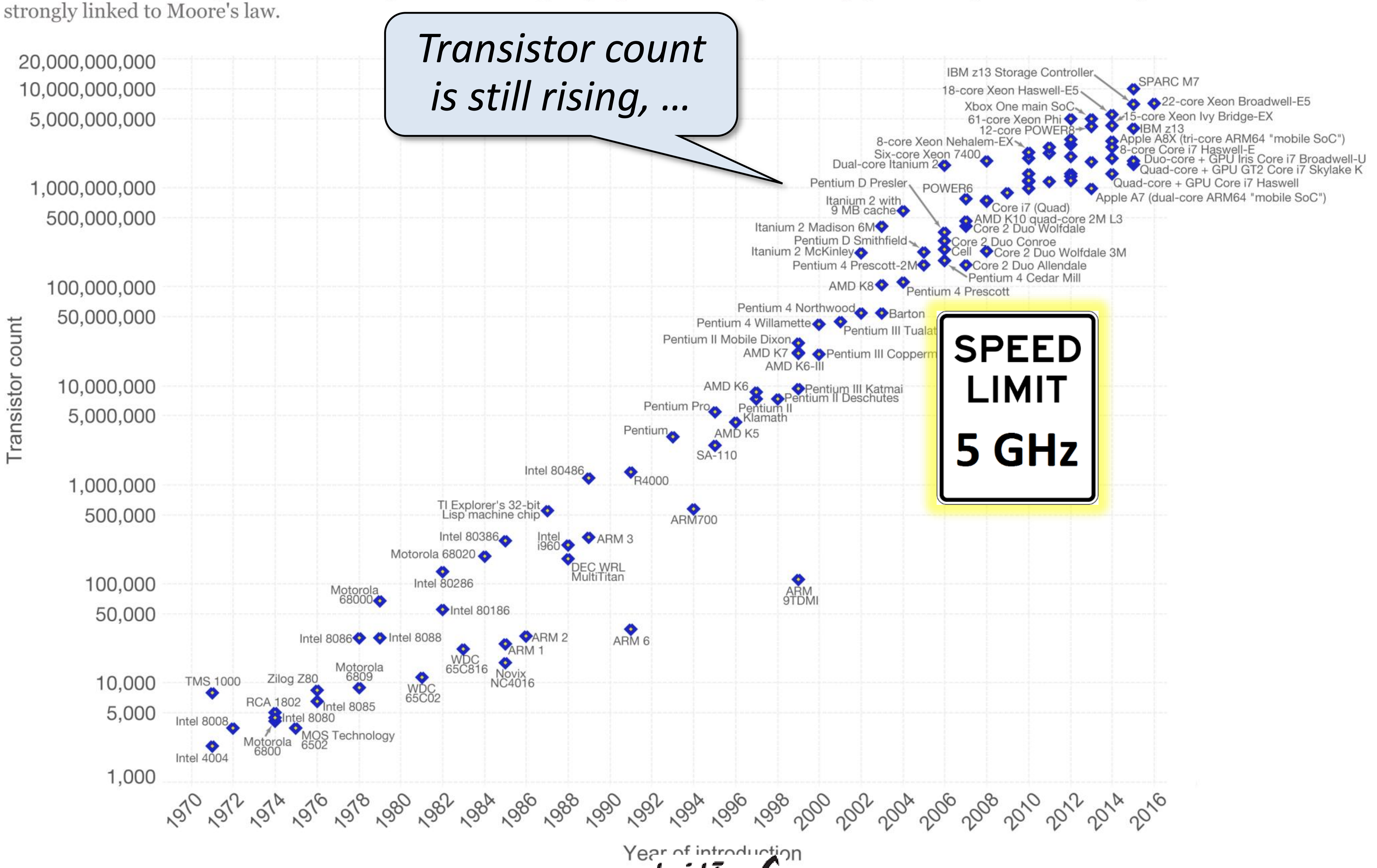


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations a[…] topic.

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

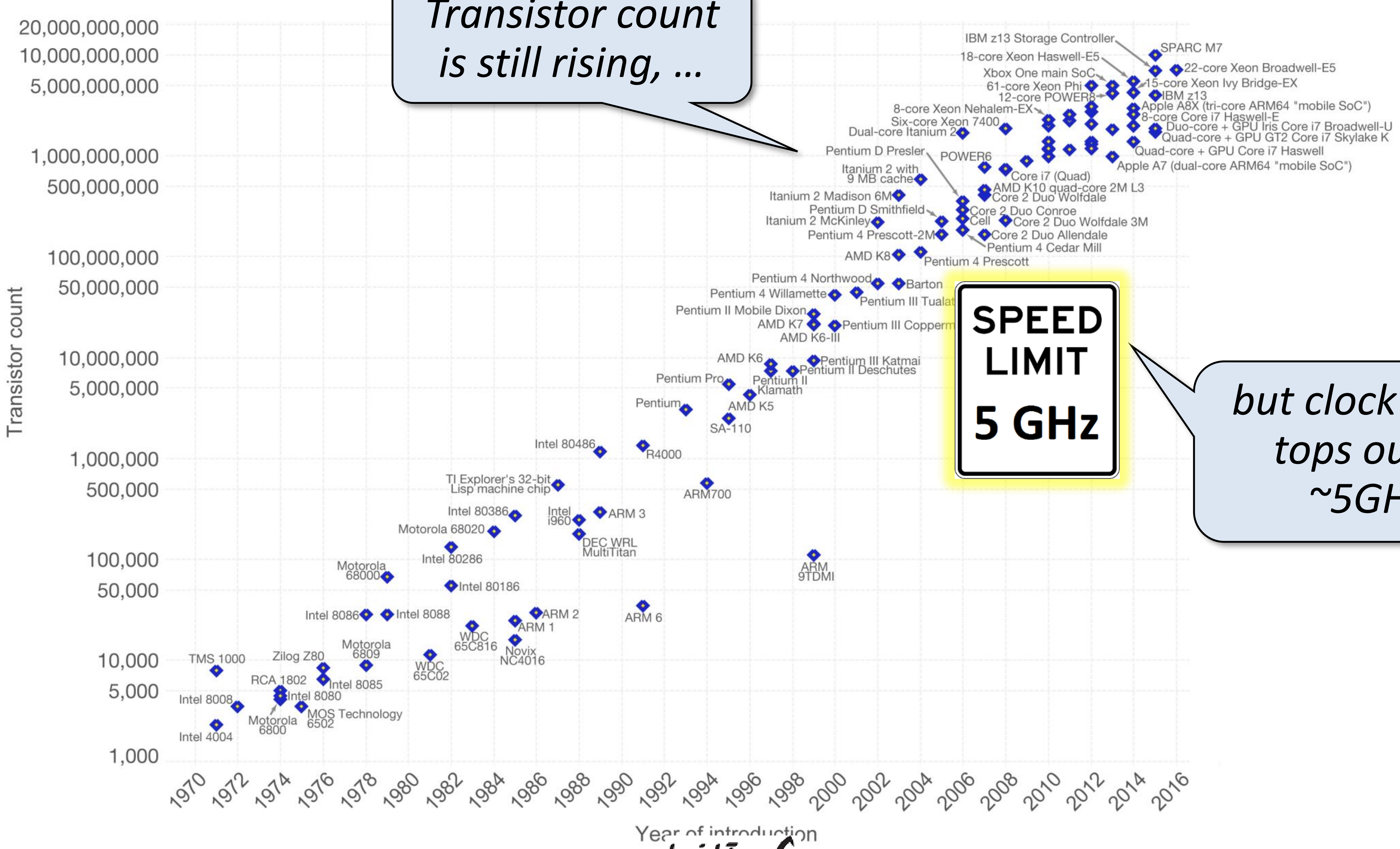


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations a[…] topic.

Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

Transistor count is still rising, …

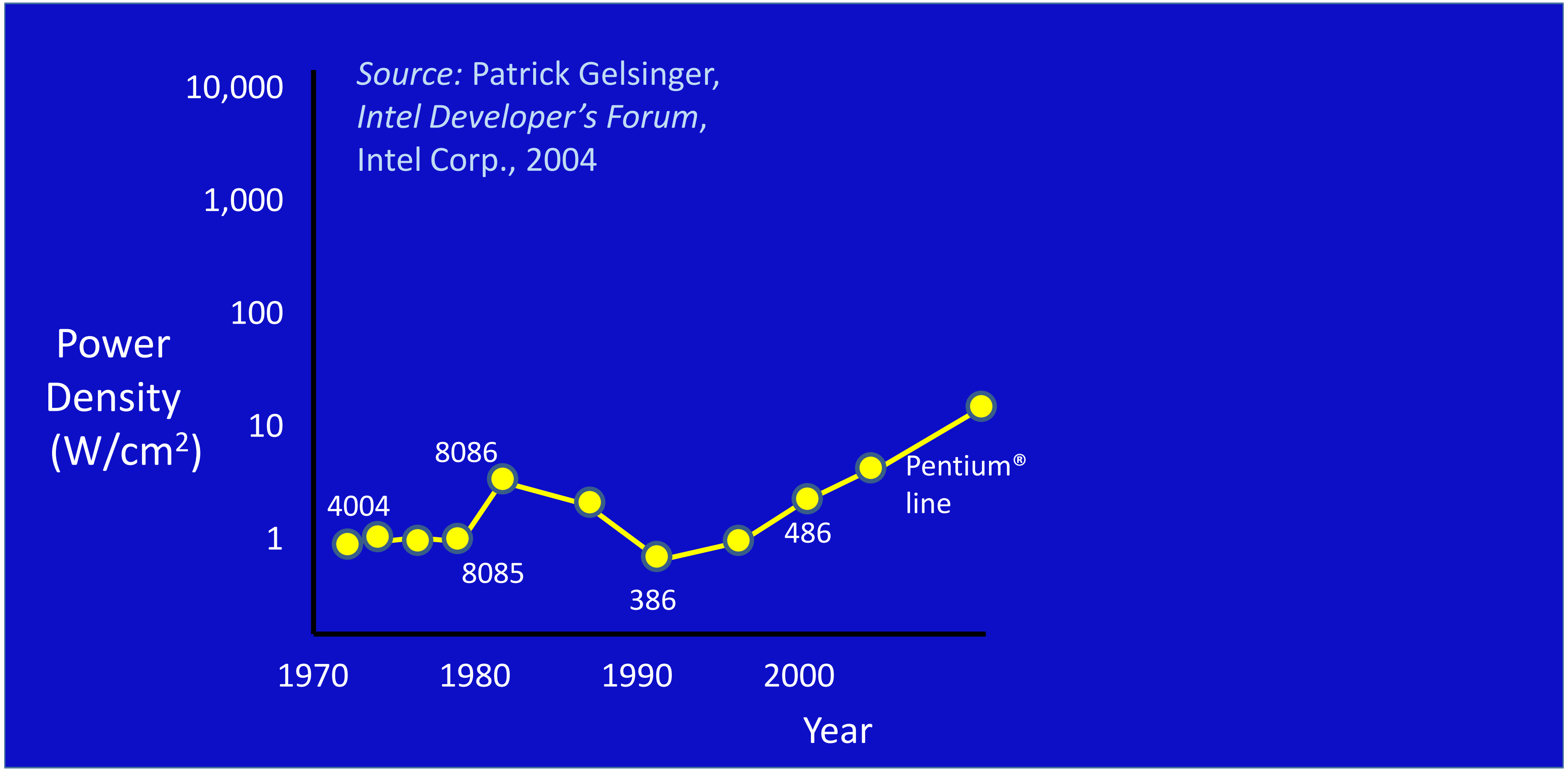


but clock speed tops out at ~5GHz.

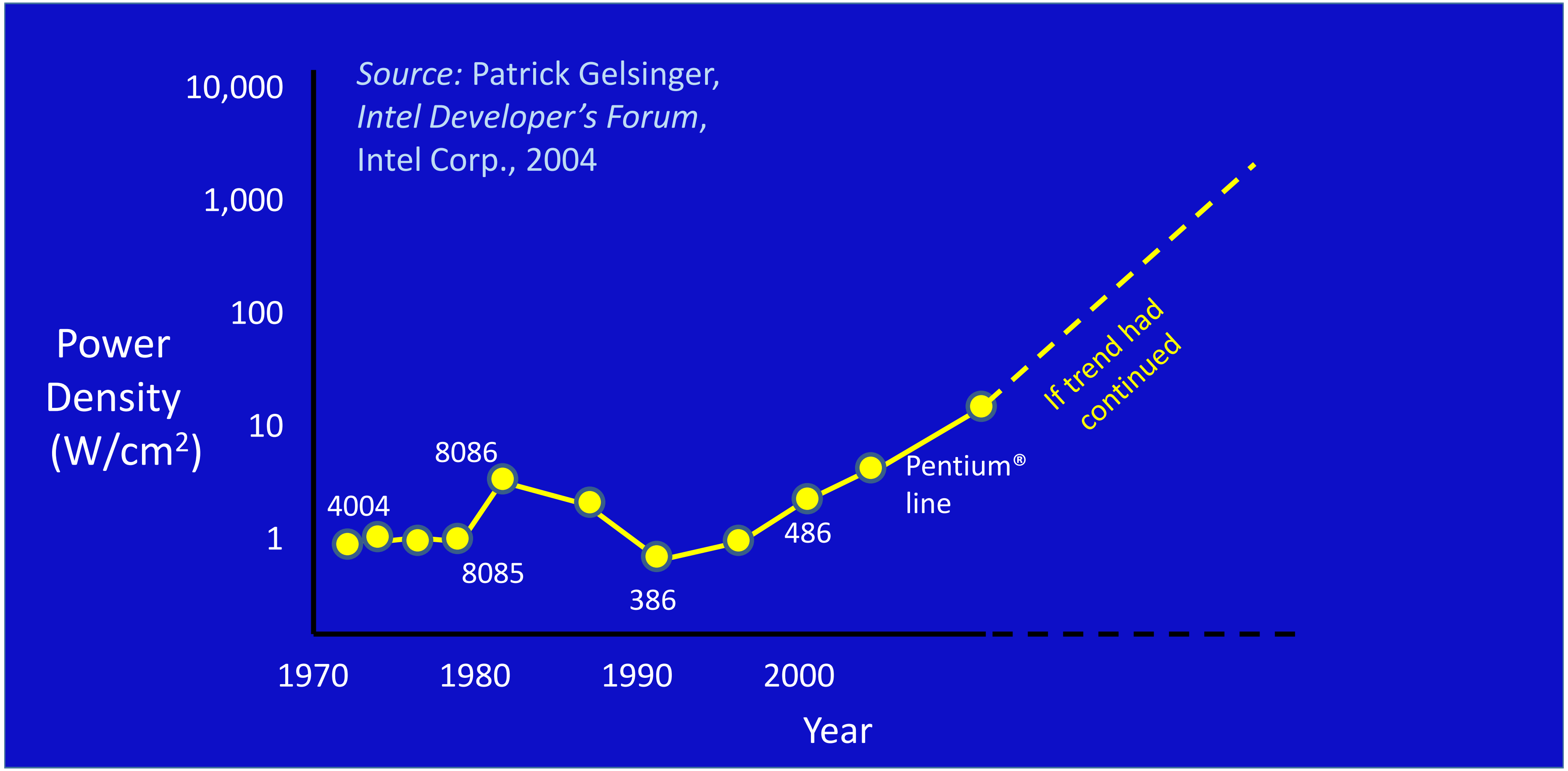
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations a topic.

3

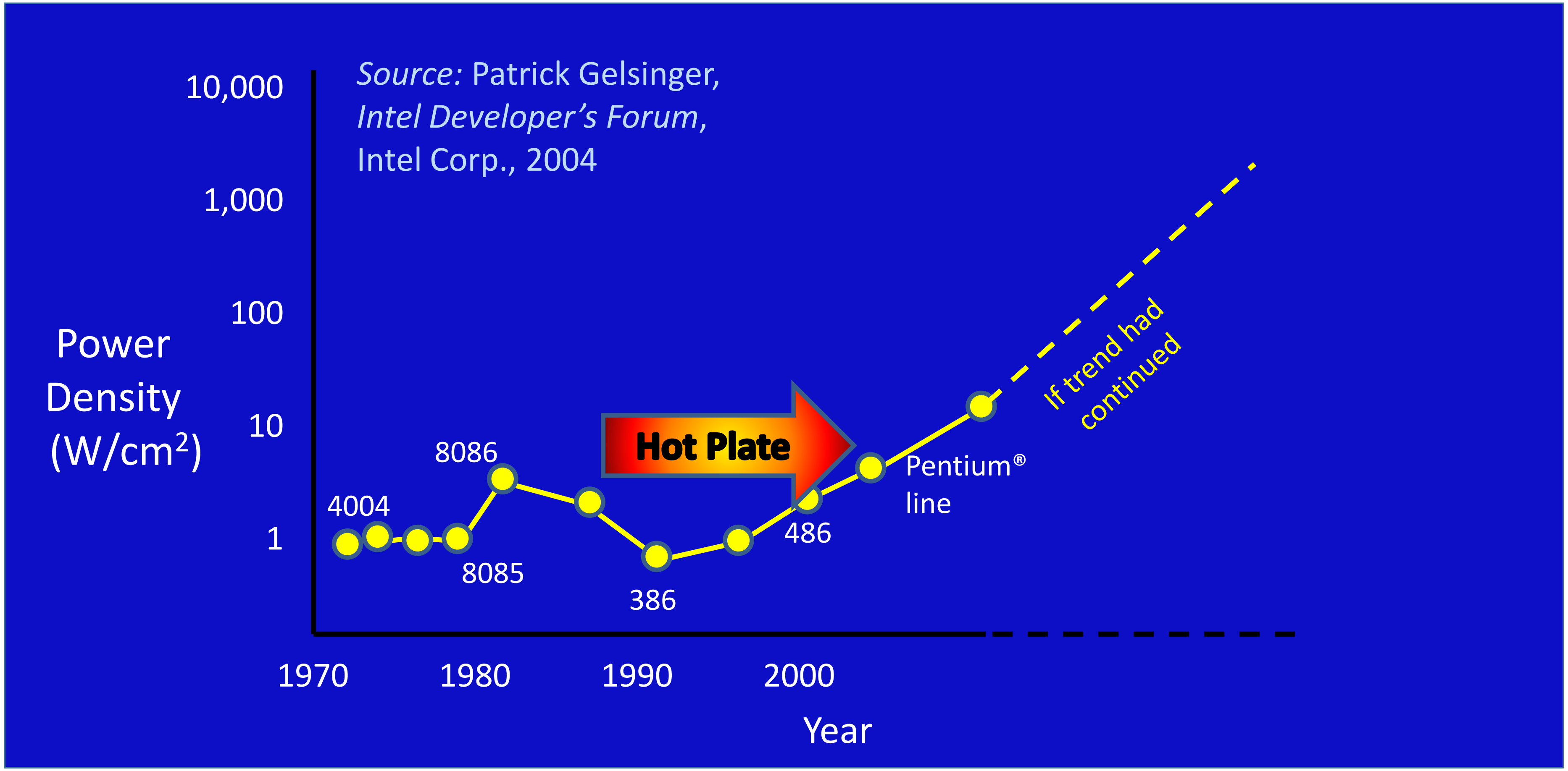
The single-core power/heat wall



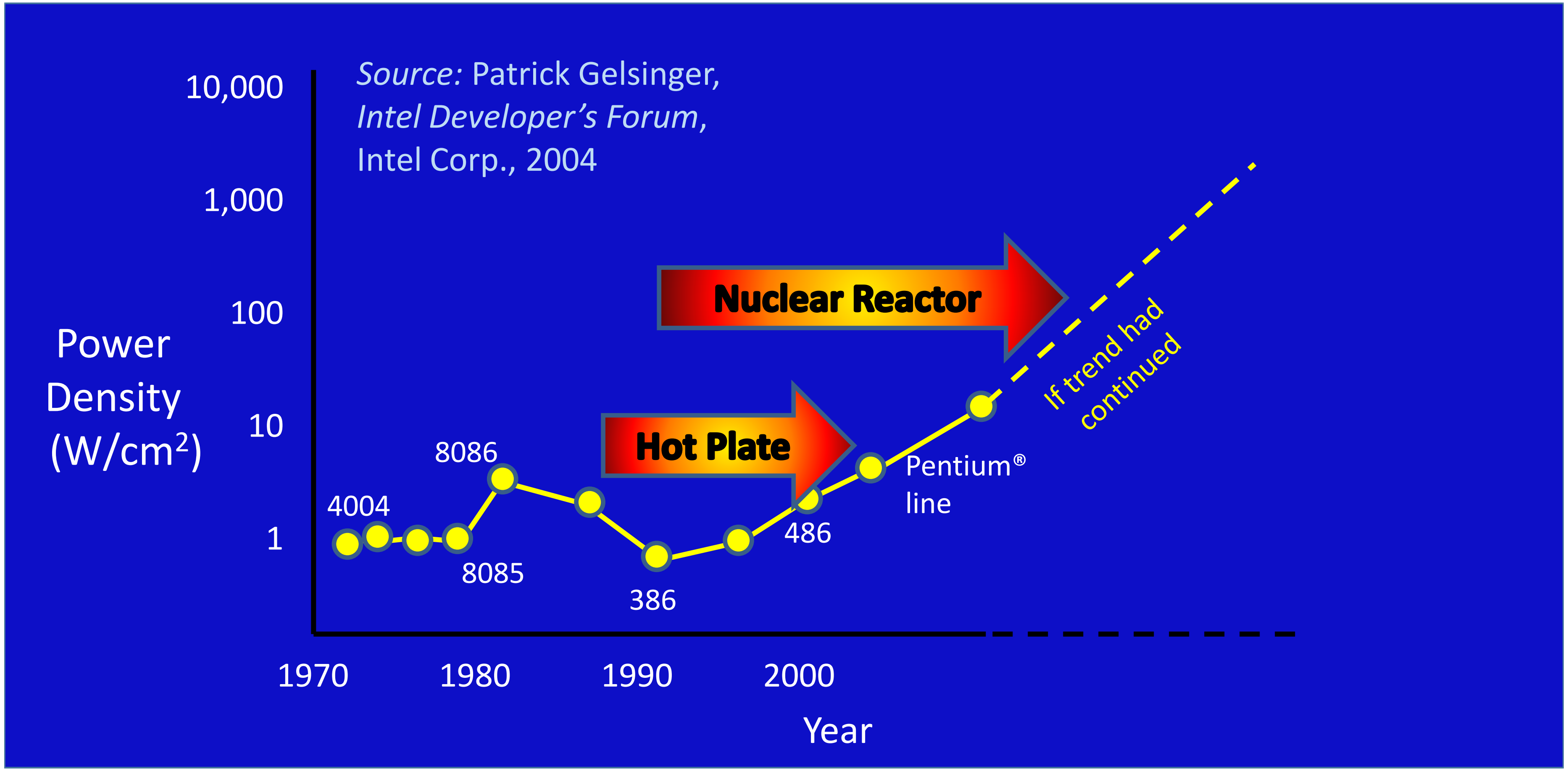
The single-core power/heat wall



The single-core power/heat wall

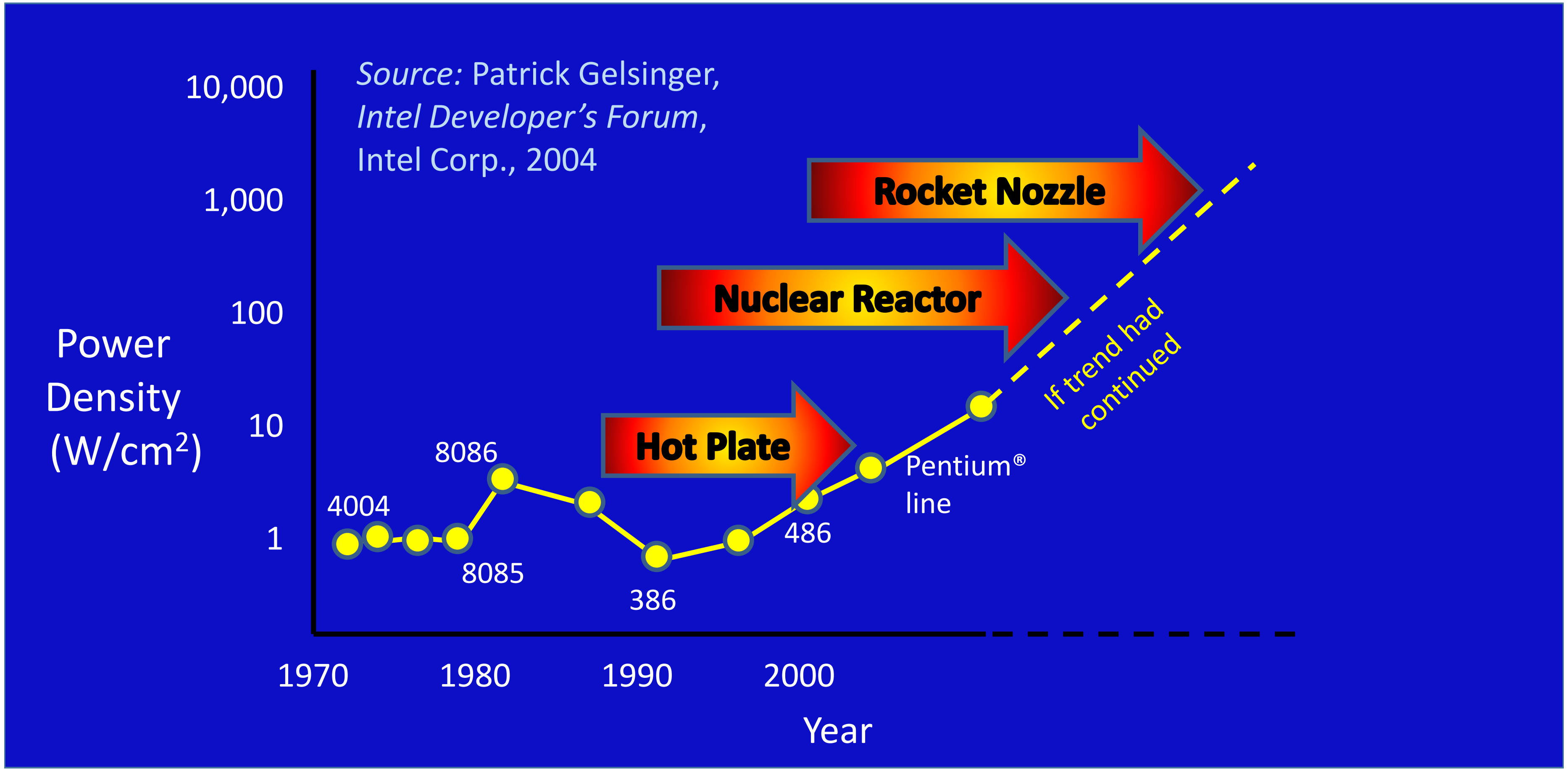


The single-core power/heat wall

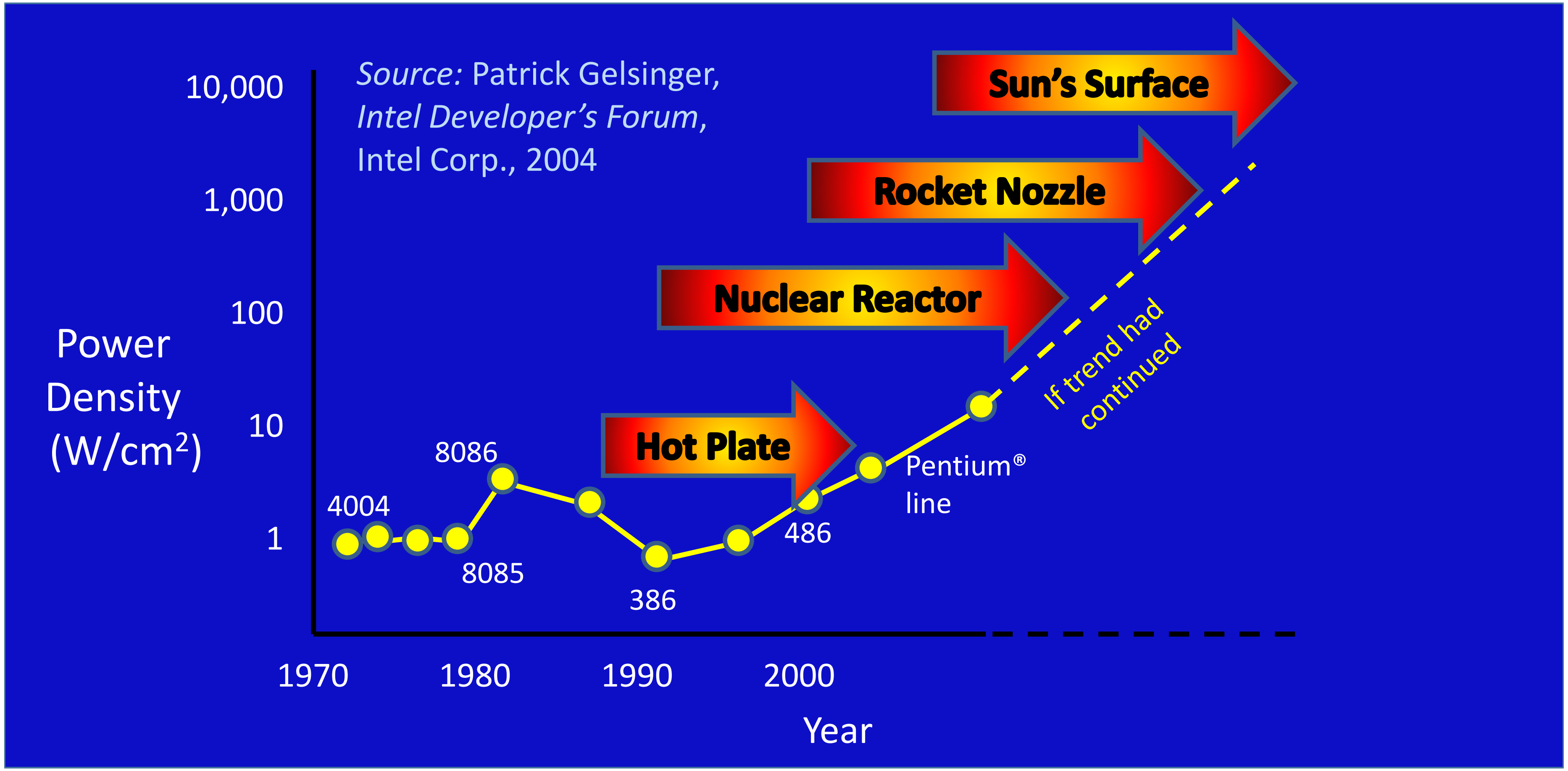


ALPHA آلفا

The single-core power/heat wall



The single-core power/heat wall



Moore's Law- cont.

Moore’s Law- cont.

- Moore’s law continues to increase transistor count, but clock speed has topped out.

Moore's Law- cont.

- Moore's law continues to increase transistor count, but clock speed has topped out.

- In Moore's law era:

  Each year we get faster processors.

Moore’s Law- cont.

- Moore’s law continues to increase transistor count, but clock speed has topped out.

- In Moore’s law era:

  Each year we get faster processors.

VS.

Moore's Law- cont.

- Moore's law continues to increase transistor count, but clock speed has topped out.

- In Moore's law era:

  Each year we get faster processors.

vs.

- Currently:

  Each year we get more processors.

The free lunch is over!

The free lunch is over!

Herb Sutter

The free lunch is over!

- Seminal and legendary article:

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, March 2005.

Herb Sutter

The free lunch is over!


Herb Sutter

- Seminal and legendary article:

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, March 2005.

- The "free lunch" of performance - in the form of ever faster clock speeds - is over!

آلفا
ALPHA

The free lunch is over!

- Seminal and legendary article:

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, March 2005.

Herb Sutter

From around 2004-2005

- The "free lunch" of performance - in the form of ever faster clock speeds - is over!

The free lunch is over!

- Seminal and legendary article:

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, March 2005.

Herb Sutter

From around 2004-2005

- The "free lunch" of performance - in the form of ever faster clock speeds - is over!
- The free lunch is over since more than a decade ago.

The free lunch is over!

- Seminal and legendary article:

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, March 2005.

Herb Sutter

From around 2004-2005

- The "free lunch" of performance - in the form of ever faster clock speeds - is over!
- The free lunch is over since more than a decade ago.
- Processors are no longer getting faster. The number of transistors on a chip still increases according to Moore's law, but those transistors are used for more processors and more memory.

Why parallel processing?

- Serialization is a learned skill that has been over-learned.
- All computers are now parallel.
- Programming means *parallel* programming.

Multicore … definitions

- Another sequence of doublings and redoublings …
- A *multi-core processor* is a single computing component with two or more independent actual processing units (called "cores"), which are units that read and execute program instructions.

Wikipedia: https://en.wikipedia.org/wiki/Multi-core_processor

- ***Chip multiprocessors*** contain a single ***multicore*** integrated-circuit chip that houses multiple processing “cores,” each of which is a full-fledged processor that can access a common memory.

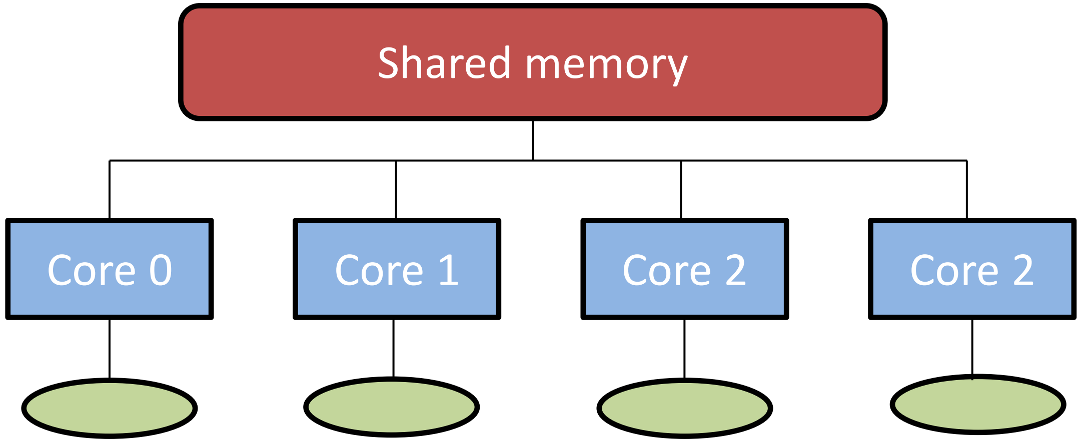
- Multicore vs. Multiprocessor: A computer may have multiple –actual- processors, each of which has multiple cores.

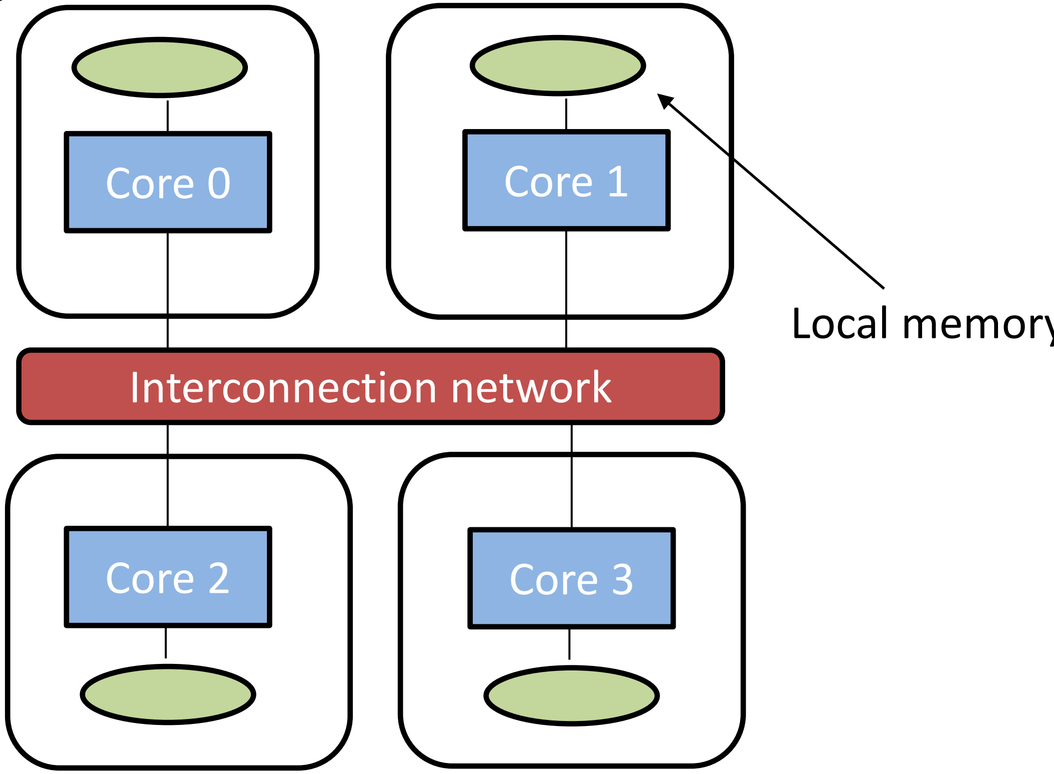
  - Core count: total number of instructions that an be executed simultaneously.

Machine model

Shared memory machine model

Distributed memory machine model





- Shared memory machine model: Each processor can directly access any location of memory.
- Distributed memory machine model: each processor’s memory is private, and an explicit message must be sent between processors in order for one processor to access the memory of another.
- We focus on Shared memory machine model.

Models of concurrency

Parallel vs. Concurrent

- A system is said to be concurrent if it can support two or more actions *in progress* at the *same time*.

A system is said to be parallel if it can support two or more actions executing *simultaneously*.

- Sports analogy: Basketball vs. Running

- More Concurrent sports: Football, Baseball, …
- More parallel sports: Golf, Swimming, …

the picture adapted from Pablo Halpern. Overview on Parallel Programming in C++. CppCon2014, September 2014.

- In parallel execution, there must be multiple cores available within the computation platform. In that case, the two or more threads could each be assigned a separate core and would be running simultaneously.

- Parallel is subset of Concurrent

Parallel vs. Concurrent

Parallel vs. Concurrent

vs.

Parallel vs. Concurrent

Concurrency

vs.

Parallel vs. Concurrent

Concurrency

vs.

Parallelism

Parallel vs. Concurrent

Concurrency

Concurrent Programming

vs.

Parallelism

Parallel vs. Concurrent

Concurrency

Concurrent Programming

VS.

Parallel Programming

Parallelism

Parallel vs. Concurrent

Concurrency → Single core/Multicore

Concurrent Programming

vs.

Parallel Programming

Parallelism

Parallel vs. Concurrent

Concurrency → Single core/Multicore

Concurrent Programming

vs.

Parallel Programming

Multicore ← Parallelism

Parallel vs. Concurrent

Concurrency → Single core/Multicore

↓

Multithreading

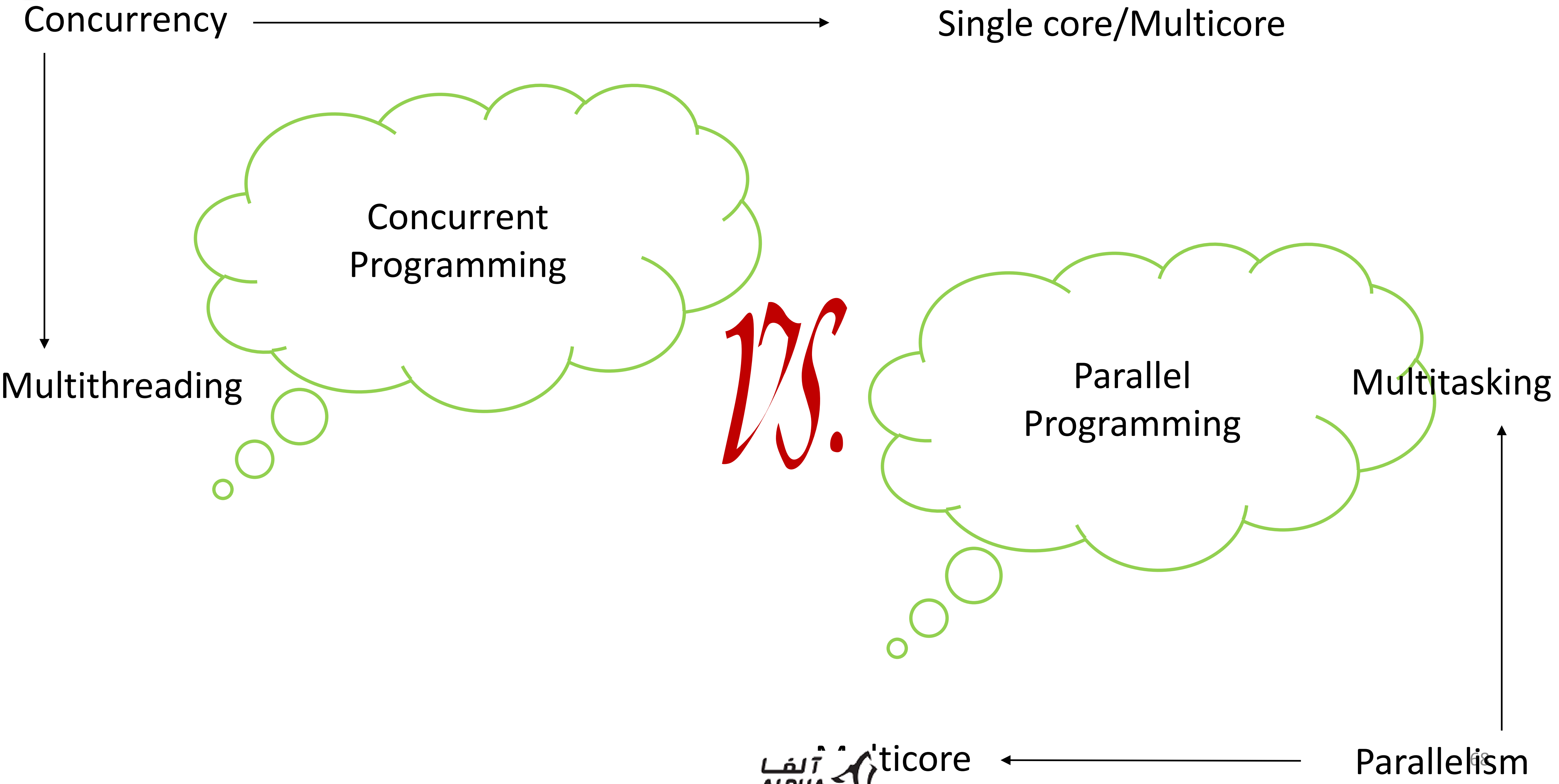
Concurrent Programming

vs.

Parallel Programming

Multicore ← Parallelism

Parallel vs. Concurrent



Parallel vs. Concurrent

Concurrency → Single core/Multicore

Interacting threads

Concurrent Programming

Multithreading

vs.

Parallel Programming

Multitasking

Multicore ← Parallelism

Parallel vs. Concurrent

Concurrency → Single core/Multicore

Interacting threads

Concurrent Programming

Multithreading

vs.

Parallel Programming

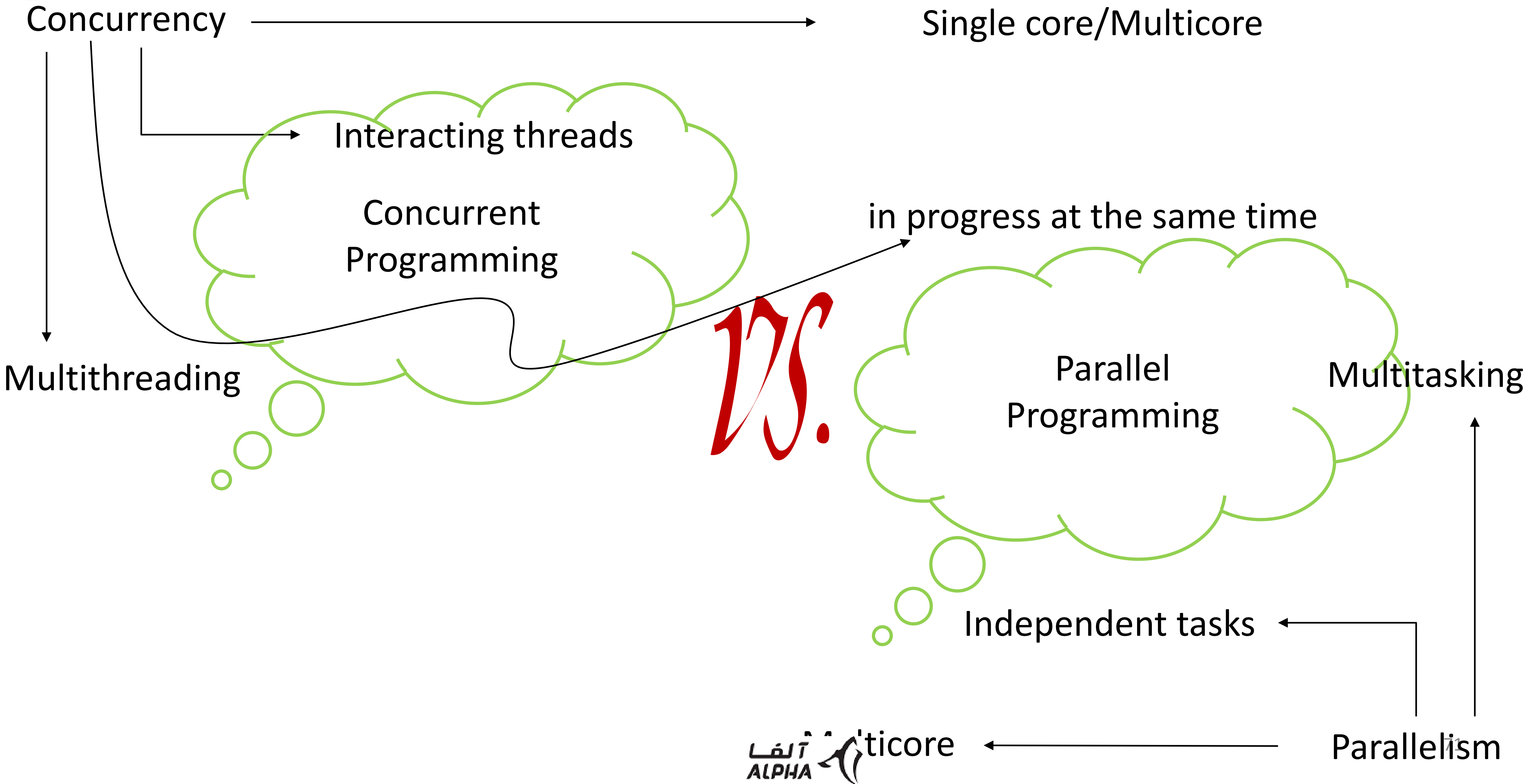
Multitasking

Independent tasks

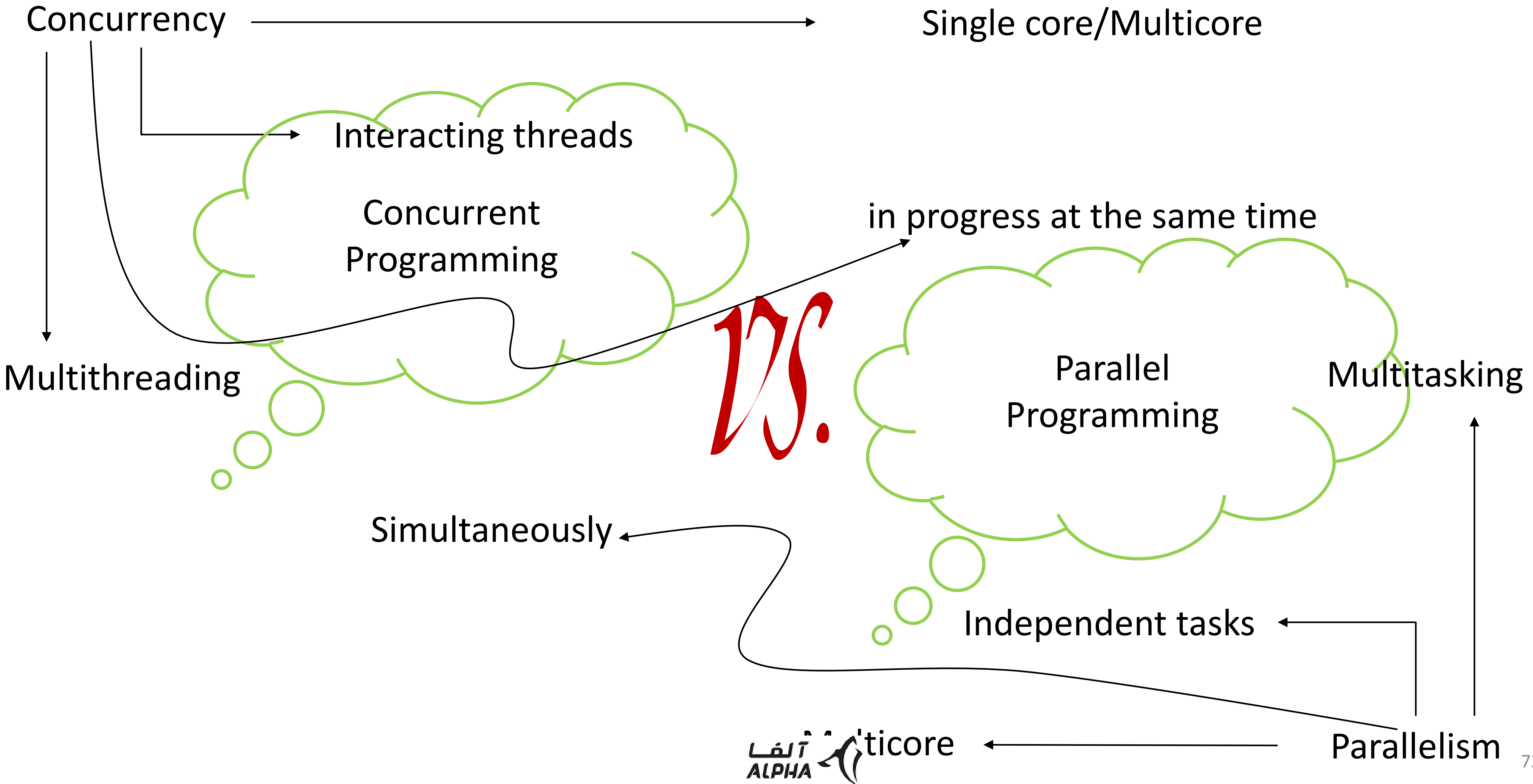
Multicore

Parallelism

Parallel vs. Concurrent



Parallel vs. Concurrent



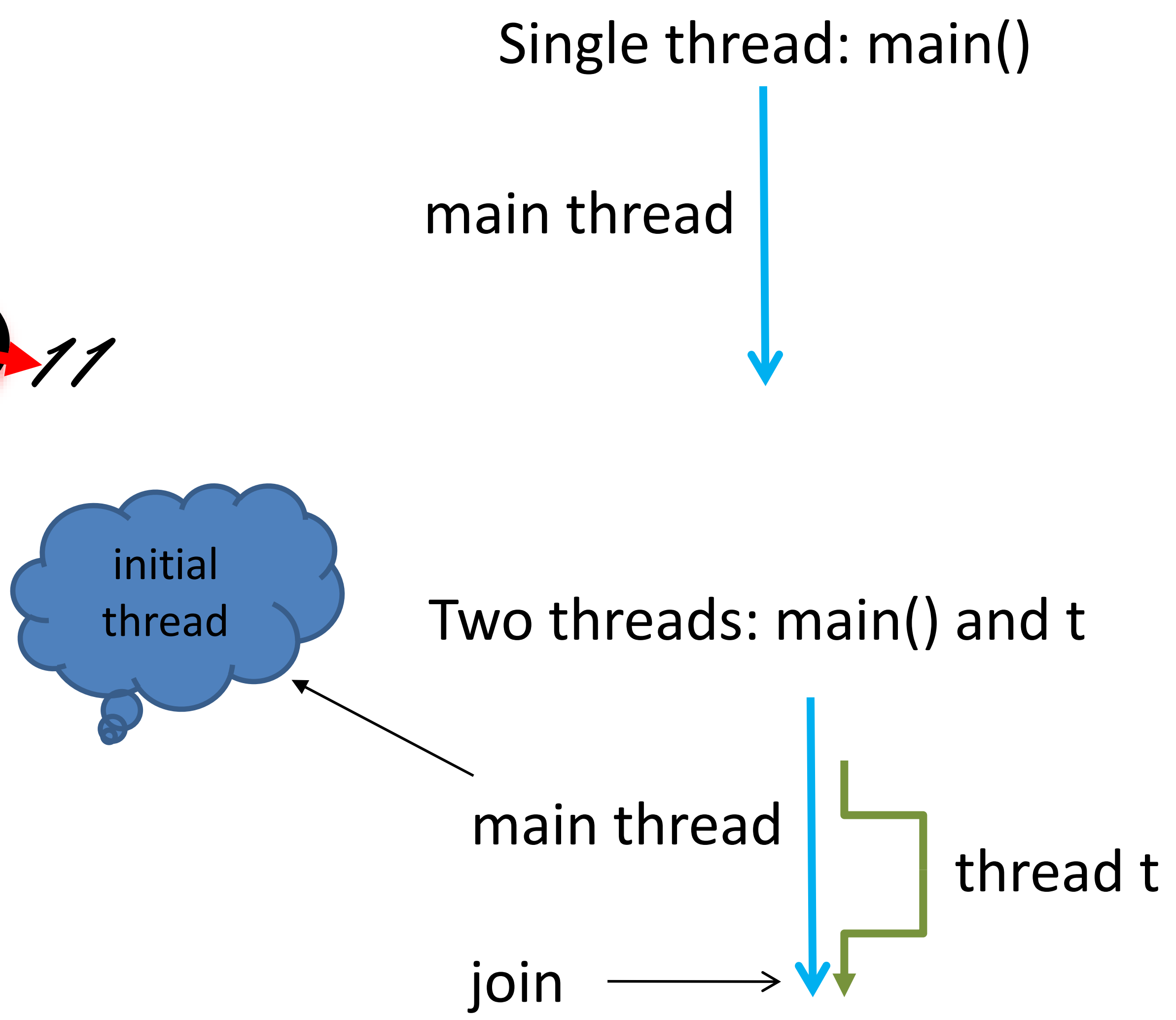
Thread as a model of concurrency

- A concurrent application will have two or more threads in progress at some time.

Single-threaded vs. Multi-threaded Hello, world!

```
// C++98, say “hello, world”
#include <iostream>
int main()
{
  std::cout << “Hello, world!\n”;
  return 0;
}
```

```
#include <thread>
#include <iostream>

void f()
{
    std::cout << “Hello, Concurrent world!\n“;
}

int main() // initial thread
{
    std::thread t{f}; // start a separate thread
    t.join(); // wait for the thread to terminate

    return 0;
}
```



- Multithreaded (actually two threads) : main(), t
- Join means wait for the thread to terminate.

Multi-threaded Hello, world!: details

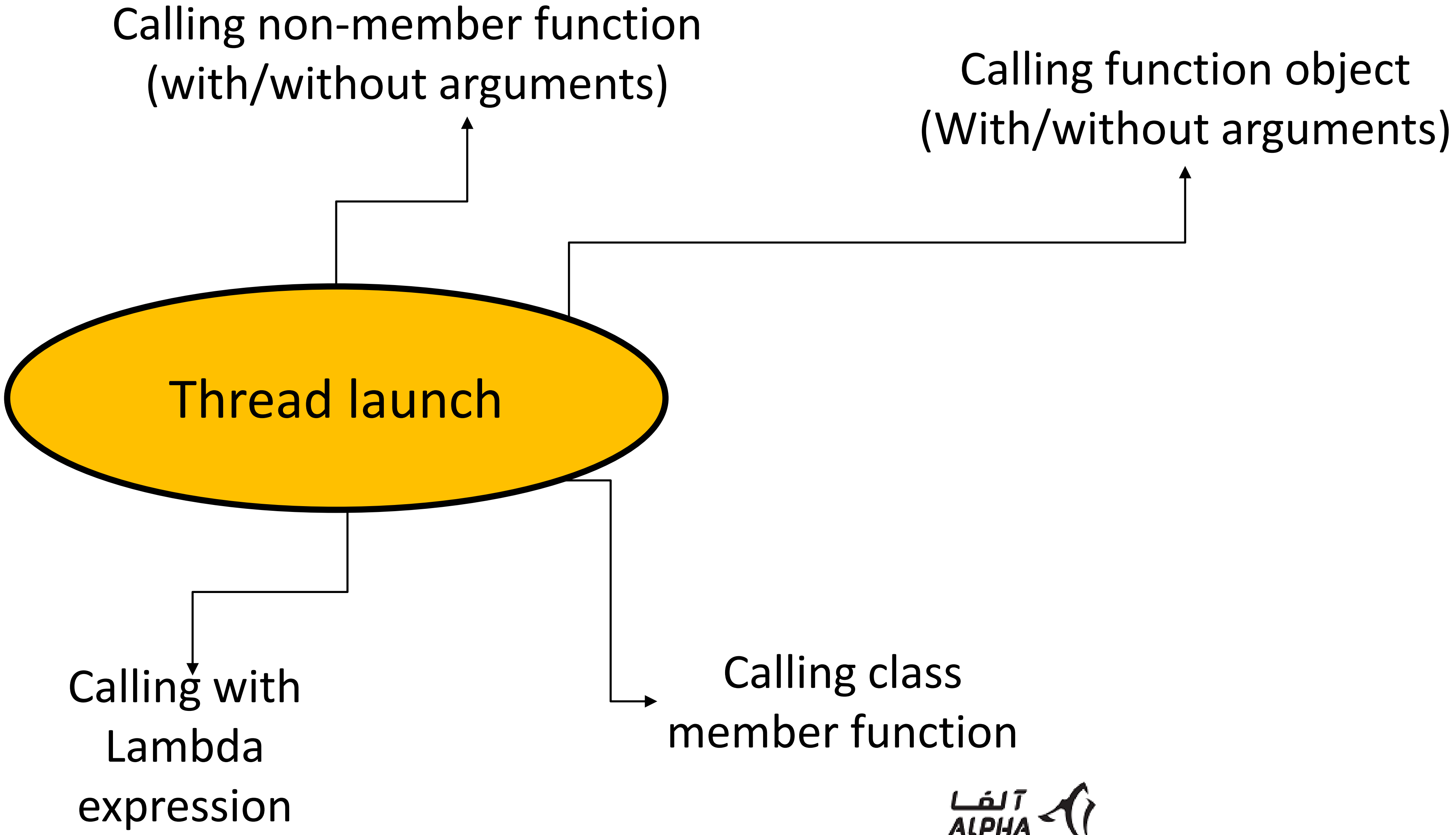
```
#include <thread>
#include <iostream>

void f()
{
    std::cout << "Hello, Concurrent world!\n";
}

int main()
{
    std::thread t{f}; // start a separate thread
    t.join(); // wait for the thread to terminate

    return 0;
}
```

1. the functions and classes for managing threads are declared in <thread>
2. thread's initial function
3. thread launch: start a new thread
4. Wait for the thread to terminate

- C++98 is a sequential programming language.
- A C++98 program had one *thread of execution*.
- A C++11 program can have more than one *thread of execution* running concurrently.

thread Launch

- A thread is launched by constructing a std::thread with a non-member (global) function, function object (functor) and member function call.



Launching threads

- Starting threads

  - with *ordinary functions*

  - with *function objects*

  - with *initializer list*

  - *with bind*

  - *with member functions*

  - *with Reference arguments*

Launch a thread with Function object

```
class Greeting {
     string message;
public:
     explicit Greeting(const string& m) : message{m} {}
     void operator()() const { cout << message << endl; }
};
int main()
{
     Greeting g{"Good bye ...“};
     thread t1{g}; // call Greeting::operator()
     thread t2{Greeting{“Hello”}}; // call Greeting::operator()
     t1.join();
     t2.join();
     return 0;
}
```

Launch a thread with lambda expressions

```
#include <thread>
#include <map>
#include <string>
#include <iostream>

std::map<int, std::string> m = { { 1, "one" }, { 0, "zero" } };

int main()
{
     std::string s1 = m[1];
     std::thread t([&]() {std::cout << s1 << ", "; });
     std::string s2 = m[0];
     t.join();
     std::cout << s2 << std::endl;
}
```

Passing arguments to thread functions

1

```
void Greeting(const string& message)
{
    cout << message << endl;
}

int main()
{
    thread t{Greeting, "Welcome …"};
    t.join();
    return 0;
}
```

2

```
void sum(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;
    cout << sum << '\n';
}

int main()
{
    thread t{sum, 1000};
    t.join();
    return 0;
}
```

3

```
void sum(int i, float f, short s, long l, double d)
{
  // …
}

int main()
{
  thread t{sum, 1, 1.0F, 1, 1L, 1.0};
  t.join();

  return 0;
}
```

Member functions and reference arguments

1

```
struct SayHello {
    void Greeting()
    {
        cout << "Hello " << std::endl;
    }
};

int main()
{
    SayHello h;
    std::thread t{&SayHello::Greeting, &h};
    t.join();

    return 0;
}
```

2

```
class SayHello {
public:
    void greeting(const std::string & message) const
    {
        std::cout<<message<<std::endl;
    }
};

int main()
{
    SayHello x;
    std::thread t(&SayHello::greeting, &x, "goodbye");
    t.join();
}
```

The Join function

- The join() ensure that we don't terminate until the threads have completed.
- To join means to *wait for the thread to terminate*.

1

```
#include<thread>
#include <iostream>
struct F {
    void operator()() {
        std::cout << "Hello, multithreaded world!\n";
    }
};

int main()
{
    std::thread my_thread{F()};
    // the program would have terminated before printing anything
    return 0;
}
```

2

```
// as before
int main()
{
    std::thread my_thread{F()};
    my_thread.join(); // don't proceed until my_thread completes
    return 0;
}
```

- A thread should be join, if it’s joinable.

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant

Q & A