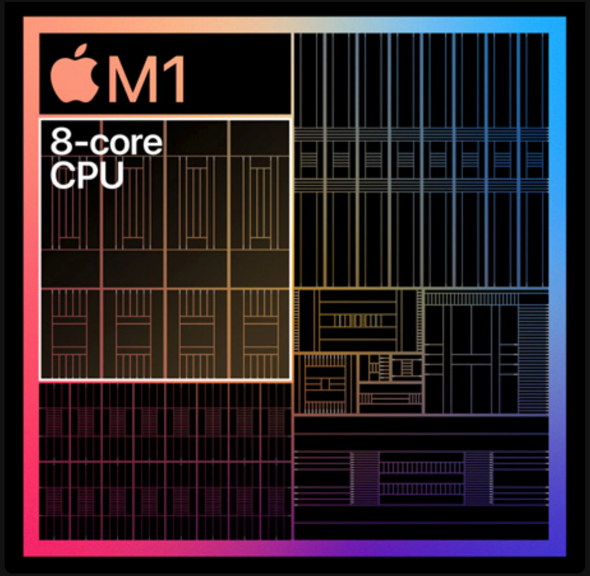
Introduction

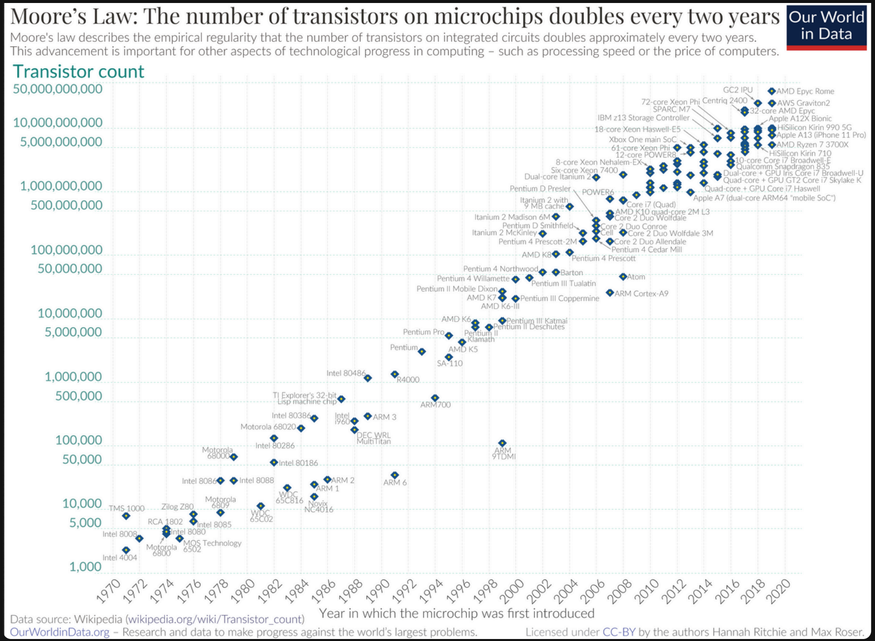
Introduction

Parallel computing is an important aspect of modern computing systems.

And while you *may* not be writing parallel code ourselves, with the tools we use every day automatically leveraging parallelism, it is important to understand the concepts and principles behind parallel computing.



Single processor performance



- 1986 - 2003
- 2003
- 2015 - 2017

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**.

> Intel Pentium D and AMD Athlon 64 x 2

For software developers:

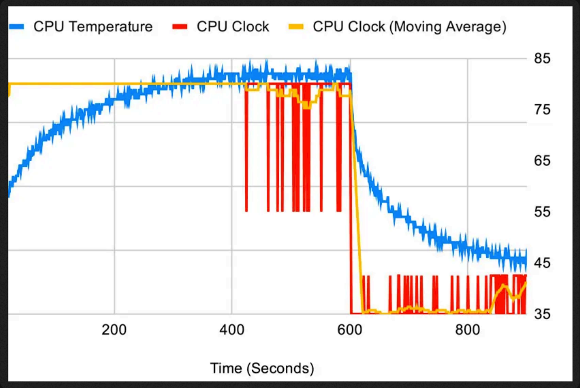
- simply adding more processors will not improve the performance of a programs

Why care about parallel computing?

To increase performance

limitations of single processor performance

- As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased.
- As the speed of transistors increases, their power consumption also increases.
- Most of this power is dissipated as heat, and air-cooled integrated circuits reached the limits of their ability to dissipate heat



Therefore the only valid path for improved performance is **not** relying on ever-faster single processors

why write parallel programs

- Why do we have to actually write parallel programs?

Most programs that have been written for conventional, single-core systems **cannot** use the presence of multiple cores

While we can run multiple instances of a program on a multicore system, that's not very useful

- We need to either rewrite our serial programs or write translation programs

The problem with translation

Researchers have had limited success writing programs that convert serial programs

An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of *each of its steps*.

The best parallelization tends to be from devising an entirely new algorithm

An example: sum

Problem

> Suppose that we need to compute n values and add them together

```
[1, 4, 3, 9, 2, 8,
 5, 1, 1, 6, 2, 7,
 2, 5, 0, 4, 1, 8,
 6, 5, 1, 2, 3, 9,]
```

An example implementation

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

With multiple cores

Assume eight cores

```
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;
}
```

With 24 values and 8 cores, how much should each core compute?

Global Sum

```
if (Im the master core) {
    sum = my_sum;
    for each core other than myself {
        recieve value from core;
        sum += value;
    }
} else {
    send my my_sum to master core;
}
```

How would you improve this program

hint: how would you lower the amount of time spent in the main core

How to write parallel programs

How to 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

There are some common approaches for this, but before that

Example

- Suppose that I have to teach a section of this subject
- Say that I have **100** students
- At the end of the semester, I have to grade their final exams
- So I recruit **4** student assistants to help me check
- the final exam consists of **five** open ended questions with a rubric

How would I use the five of us to grade the exams as quickly as possible?

Task Parallelism

I can give each of the 4 assistants a **task**

- SA 1 grades only question 1,
- SA 2 grades only question 2,
- and so on.

The total task is *partitioned* into five **different** tasks, each of which is done by one person (core)

This is called **task parallelism**

Data Parallelism

Alternatively, I can divide the one hundred exams into **five piles** of twenty exams each,

- SA 1 grades all the papers in the first pile,
- SA 2 grades all the papers in the second pile,
- and so on.

The total task is *partitioned* is not partitioned, but the **data** to be processed is partitioned into five **similar** tasks, each of which is done by one person

Give me pros and cons of each scenario

Back to the sum example

Is it task or data parallelism?

Coordination

Coordination in Parallel Programs

if the tasks can be done independently

> Writing parallel programs can be easy

When cores can work independently, it works the same way as serial programs

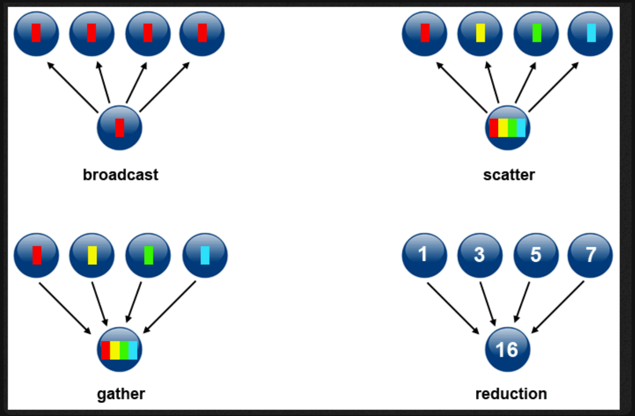
If not, there needs to be **coordination** between tasks

Coordination in Parallel Programs

The more common ways for tasks to be coordinated are:

Communication

- One or more cores **send** their current partial sums to another core
- **Sending** and **receiving** data



Load balancing

- we want the amount of time taken by each core to be roughly **the same**
- if one core has to do more work, we **waste** computing resources

Synchronization

> Instead of computing the values to be added, the values are human inputs.

Say, `x` is an array that is read in by the master core:

```
if ( Im the master core )
    for (my_i = 0; my_i < n; my_i ++)
        scanf("%d", &x[my_i]);
```

In most systems the cores are **not automatically synchronized**.

- Each core works at its own pace.

If we assume that the core is running the following code:

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
    my_sum += x[my_i];
```

Give me one runtime error

Synchronization

In this case, the cores need to **wait** before starting execution of the code:

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
    my_sum += x[my_i];
```

We need to add in a point of **synchronization** *between* the initialization of `x` and the computation of the `partial sums`:

```
Sync_cores();
```

The idea here is that each core will wait **in the function** `Sync_cores` until **all** the cores have entered the function

> So given, `input`, `compute partial`, `sync`, and `compute global`, what is the correct order of execution?

Quiz in NEO

15-minute study break