Chapter 1: Role of parallelism

Why? How?

How to exploit parallelism?

No matter which computer we use, and no matter which programming tools we use, there is one ingredient that is absolutely necessary in order to exploit parallelism: we need to have **independent operations** in our program. Here are two examples:

Dependent operations	Independent operations
`a1 *= a0;`	`b1 *= a1;`
`a2 *= a1;`	`b2 *= a2;`
`a3 *= a2;`	`b3 *= a3;`
`a4 *= a3;`	`b4 *= a4;`
`a5 *= a4;`	`b5 *= a5;`

In the first example, each operation depends on all previous operations: To get started with the multiplication `a5 * a4`, we will need to know what is the value of `a4`, but that depends on the result of the multiplication `a4 * a3`, but this operation cannot be started until we know the value of `a3`, etc. The entire sequence of operations is **inherently sequential**. Even if we had a large number of parallel multiplication units available, we could not use more than one at the same time. Pipelining would not help, either; we cannot start an operation until we know the values of the operands. In this example, the performance is limited by the **latency** of multiplications.

However, in the second example there are no dependencies between the operations. If there are resources available in the processor, we could perform all multiplications simultaneously in parallel, or we could pipeline these in an arbitrary manner. There are lots of **opportunities for parallelism** in this code, and the only fundamental limitation for the performance is the **throughput** of multiplications.

It is good to note that these concepts are in no way limited to arithmetic operations. Here is another pair of examples in which the operations are array lookups:

Dependent operations	Independent operations
`a1 = x[a0];`	`b1 = x[a1];`
`a2 = x[a1];`	`b2 = x[a2];`
`a3 = x[a2];`	`b3 = x[a3];`
`a4 = x[a3];`	`b4 = x[a4];`
`a5 = x[a4];`	`b5 = x[a5];`

Note that the left example is, in essence, what happens when we follow a linked list; again, this is inherently sequential, and provides no opportunities for parallelism. In the second example, we can start another array lookup without waiting for the first operation to finish; there are plenty of opportunities for parallelism.

Creating potential for parallelism and realizing it

We have seen that

- If we want to get a **good performance** with modern CPUs, it is necessary to make use of **parallelism**.
- If we want to make any use of **parallelism**, it is necessary to have **independent operations**.

Two challenges now arise: First, many classical algorithms have lots of dependencies between the operations, so we do not seem to have any opportunities for parallelism. Second, even if we happen to have opportunities for parallelism, how do we instruct the hardware to make use of such opportunities?

This course largely revolves around these themes:

1. **Creating potential for parallelism:** We will learn how to **redesign algorithms** so that there are more independent operations and hence more opportunities for parallelism. Of course in the general case, this might be hard or impossible, but as we will see, there are many algorithms in which simple small modifications will reveal opportunities for parallelism.
2. **Exploiting potential for parallelism:** We will learn how to write programs that **instruct the hardware** to execute independent operations in parallel. This is the more technical part of the course, and we will have to learn a bit about both modern computer hardware and modern programming tools.

We start to explore both of these questions in the context of CPUs in the next chapter; we will look at GPU programming a bit later.