Summary

1.10 Summary

For many years we've reaped the benefits of having ever-faster processors. However, because of physical limitations, the rate of performance improvement in conventional

A very dense overview of the material and definitions that will be covered in the course

processors has decreased dramatically. To increase the power of processors, chipmakers have turned to **multicore** integrated circuits, that is, integrated circuits with multiple conventional processors on a single chip.

Ordinary **serial** programs, which are programs written for a conventional single-core processor, usually cannot exploit the presence of multiple cores, and it's unlikely that translation programs will be able to shoulder all the work of converting serial programs into **parallel programs**—programs that can make use of multiple cores. As software developers, we need to learn to write parallel programs.

When we write parallel programs, we usually need to **coordinate** the work of the cores. This can involve **communication** among the cores, **load balancing**, and **synchronization** of the cores.

In this book we'll be learning to program parallel systems, so that we can maximize their performance. We'll be using the C language with four different **application program interfaces** or **APIs**: MPI, Pthreads, OpenMP, and CUDA. These APIs are used to program parallel systems that are classified according to how the cores access memory and whether the individual cores can operate independently of each other.

In the first classification, we distinguish between **shared-memory** and **distributed-memory** systems. In a shared-memory system, the cores share access to one large pool of memory, and they can coordinate their actions by accessing shared memory locations. In a distributed-memory system, each core has its own private memory, and the cores can coordinate their actions by sending messages across a network.

In the second classification, we distinguish between systems with cores that can operate independently of each other and systems in which the cores all execute the same instruction. In both types of system, the cores can operate on their own data stream. So the first type of system is called a **multiple-instruction multiple-data** or **MIMD** system, and the second type of system is called a **single-instruction multiple-data** or **SIMD** system.

MPI is used for programming distributed-memory MIMD systems. Pthreads is used for programming shared-memory MIMD systems. OpenMP can be used to program both shared-memory MIMD and shared-memory SIMD systems, although we'll be looking at using it to program MIMD systems. CUDA is used for programming Nvidia **graphics processing units** or **GPUs**. GPUs have aspects of all four types of system, but we'll be mainly interested in the shared-memory SIMD and shared-memory MIMD aspects.

Concurrent programs can have multiple tasks in progress at any instant. **Parallel** and **distributed** programs usually have tasks that execute simultaneously. There isn't a hard and fast distinction between parallel and distributed, although in parallel programs, the tasks are usually more tightly coupled.

Parallel programs are usually very complex. So it's even more important to use good program development techniques with parallel programs.