The topic of this chapter is thread-level parallelism. While, thread-level parallelism falls within the textbook’s classification of ILP and data parallelism. It also falls into a broader topic of parallel and distributed computing. In the next set of slides, I will attempt to place you in the context of this broader computation space. Of course a proper treatment of parallel computing or distributed computing is worthy of an entire semester (or two) course of study. I can only give you a brief exposure to this topic.
Task Level Parallelism

- **Task Level Parallelism**: organizing a program or computing solution into a set of processes/tasks/threads for simultaneous execution.

- Conventionally one might think of task level parallelism (and the MIMD processing paradigm) as being used for a single program or operation, however, request level parallelism (e.g., serving http requests for a website) is also generally addressed/studied by hardware solutions in this same space.

- Request level processing and related problems with independent transactions (e.g., bitcoin mining, web page requests) fall into a class of problems called **embarrassingly parallel**. Embarrassingly because they have virtually no synchronization requirements and thus show linear (measured by number of compute nodes) speedup.
Parallel Computing: a form of computation in which many calculations are carried out simultaneously; operating on the principle that large problems can often be divided into discrete parts that can be solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism.

Distributed Computing/Distributed Systems: a system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

Concurrent Computing a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel. Concurrent programs (processes or threads) can be executed: (i) on a single processor by time-slicing, or (ii) in parallel by assigning each computational process to one of a set of processors. The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational executions, and coordinating access to resources that are shared among executions.
The terms *concurrent computing*, *parallel computing*, and *distributed computing* have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as “parallel” and “distributed”; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly coupled form of distributed computing, and distributed computing may be seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as “parallel” or “distributed” using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange information between processors.

- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        wait(emptyCount);
        wait(mutex);
        putItemIntoBuffer(item);
        signal(mutex);
        signal(fillCount);
    }
}

procedure consumer() {
    while (true) {
        wait(fillCount);
        wait(mutex);
        item = removeItemFromBuffer();
        signal(mutex);
        signal(emptyCount);
        consumeItem(item);
    }
}

wait(s): repeat {if s >= 0 (s = s - 1; break;)}
signal(s): s = s + 1;
Multiprocessors: computers consisting of tightly coupled processors that typically present a shared memory space. The principle topic of this chapter.

Symmetric (shared-memory) Multiprocessors (SMP): small scale multiprocessors with a shared memory space providing mostly uniform memory access (UMA). Example: single processor multicore x86 machines.

Distributed Shared Memory (DSM) multiprocessors: generally larger solutions with a distributed memory solution that provides nonuniform memory access (NUMA). Much trickier to program effectively as non-local memory references (that present to the programmer as just another memory location in their address space) can be surprisingly costly to access.

Actually all parallel programming is much harder to exploit for speedup than it appears. The difficulty of balancing the computation and managing synchronization costs between the parallel tasks is quite difficult.
Multicomputers: computers consisting of loosely coupled processors that typically present a distributed memory space. Not to be confused with a distributed memory multiprocessor.

Beowulf Clusters: a collection of (general purpose) compute nodes that are networked together for parallel computing. Often presented as a rack of blade computers, but also existing as a collection of independent computers networked together for the purpose of parallel computing.

Networking support generally provided by standard networking hardware such as Ethernet or Infiniband.
Large scale (generally above 1K nodes) parallel computing. All forms, SIMD, MIMD, GPGPUs, Clusters, Tightly Coupled Multicomputers (*e.g.*, IBM Blue Gene family).

Very interesting problem space. Fault tolerance and fault recovery become far more important.
Clusters of 10s of thousands (and beyond) of independent compute nodes. Generally providing a compute platform for supporting large scale request level parallelism.

The topic of the next chapter.
Parallelism is Hard/Amdhal’s Law

Amdhal’s Law: Idealized parallelism

\[
\text{Speedup} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]

Restating

\[
\text{Speedup} = \frac{1}{\left(1 - \% \text{ affected} + \% \text{ left after optimization}\right)}
\]

The speedup just isn’t there by conventional approaches. Gaining speedup is very difficult. In fact, this graph shows an idealized speedup without any consideration for synchronization costs.
So we look at both fixed and variable overheads. The first line adds a fixed overhead to the parallel portion equal to 0.5% of the original computation. The second line shows a variable overhead equal to doubling the runtime costs of the parallel components (not an unreasonable possibility).

$$\text{Speedup} = \frac{1}{(1 - \% \text{parallel}) + (\frac{\% \text{parallel}}{\text{processors}} + 0.05)}$$

$$\text{Speedup} = \frac{1}{(1 - \% \text{parallel}) + (\frac{\% \text{parallel}}{\text{processors}} \times 2)}$$

Neither the fixed or the variable overhead costs are accurate, but they can give us some curves to consider. In reality, the costs of synchronization will be very difficult to establish. I cannot really give you much direction here, sorry.
Gustafson’s Law

\[ S(P) = P - \alpha \cdot (P - 1) \]

- **S**: Speedup
- **P**: number of processors
- **\( \alpha \)**: the non-parallelizable fraction of any parallel process

**From wikipedia:** Gustafson called his metric scaled speedup, because in the above expression \( S(P) \) is the ratio of the total, single-process execution time to the per-process parallel execution time; the former scales with \( P \), while the latter is assumed fixed or nearly so. This is in contrast to Amdahl’s Law, which takes the single-process execution time to be the fixed quantity, and compares it to a shrinking per-process parallel execution time. Thus, Amdahl’s law is based on the assumption of a fixed problem size: it assumes the overall workload of a program does not change with respect to machine size (i.e., the number of processors). Both laws assume the parallelizable part is evenly distributed over \( P \) processors.