In discussing out-of-order superscalar pipelined CPUs we commented that the costs grow very quickly relative to the performance gains, leading to the cost-performance curve shown here.

If we move down the curve, we can arrive at more efficient architectures that give, say, 1/2 the performance at a 1/4 of the cost.

When our applications involve independent computations that can be performed in a parallel, it may be that we would be able to use two cores to provide the same performance as the original expensive core, but a fraction of the cost.

If the available parallelism allows us to use additional cores, we'll see a linear relationship between increased performance vs. increased cost.

The key, of course, is that desired computations can be divided into multiple tasks that can run independently, with little or no need for communication or coordination between the tasks.

What is the optimal tradeoff between core cost and the number of cores?

If our computation is arbitrarily divisible without incurring additional overhead, then we would continue to move down the curve until we found the cost-performance point that gave us the desired performance at the least cost.

In reality, dividing the computation across many cores does involve some overhead, e.g., distributing the data and code, then collecting and aggregating the results, so the optimal tradeoff is harder to find.

Still, the idea of using a larger number of smaller, more efficient cores seems attractive.

Many applications have some computations that can be performed in parallel, but also have computations that won't benefit from parallelism.

To understand the speedup we might expect from exploiting parallelism, it's useful to perform the calculation proposed by computer scientist Gene Amdahl in 1967, now known as Amdahl's Law.

Suppose we're considering an enhancement that speeds up some fraction F of the task at hand by a factor of S. As shown in the figure, the gray portion of the task now takes F/S of the time that it used to require.

Some simple arithmetic lets us calculate the overall speedup we get from using the enhancement.

One conclusion we can draw is that we'll benefit the most from enhancements that affect a large portion of the required computations, i.e., we want to make F as large a possible.

What's the best speedup we can hope for if we have many cores that can be used to speed up the parallel part of the task?

Here's the speedup formula based on F and S, where in this case F is the parallel fraction of the task.

If we assume that the parallel fraction of the task can be speed up arbitrarily by using more and more cores, we see that the best possible overall speed up is 1/(1-F).

For example, you write a program that can do 90% of its work in parallel, but the other 10% must be done sequentially.

The best overall speedup that can be achieved is a factor of 10, no matter how many cores you have at your disposal.

Turning the question around, suppose you have a 1000-core machine which you hope to be able to use to achieve a speedup of 500 on your target application.

You would need to be able parallelize 99.8% of the computation in order to reach your goal!

Clearly multicore machines are most useful when the target task has lots of natural parallelism.

Using multiple independent cores to execute a parallel task is called thread-level parallelism (TLP), where each core executes a separate computation "thread".

The threads are independent programs, so the execution model is potentially more flexible than the lock-step execution provided by vector machines.

When there are a small number of threads, you often see the cores sharing a common main memory, allowing the threads to communicate and synchronize by sharing a common address space.

We'll discuss this further in the next section.

This is the approach used in current multicore processors, which have between 2 and 12 cores.

Shared memory becomes a real bottleneck when there 10's or 100's of cores, since collectively they quickly overwhelm the available memory bandwidth.

In these architectures, threads communicate using a communication network to pass messages back and forth.

We discussed possible network topologies in an earlier lecture.

A cost-effective on-chip approach is to use a nearest-neighbor mesh network, which supports many parallel point-to-point communications, while still allowing multi-hop communication between any two cores.

Message passing is also used in computing clusters, where many ordinary CPUs collaborate on large tasks.

There's a standardized message passing interface (MPI) and specialized, very high throughput, low latency message-passing communication networks (e.g., Infiniband) that make it easy to build high-performance computing clusters.

In the next couple of sections we'll look more closely at some of the issues involved in building shared-memory multicore processors.