**How Parallel Processing Works**

If a computer were human, then its **central processing unit** (**CPU**) would be its brain. A CPU is a **microprocessor** -- a computing engine on a chip. While modern microprocessors are small, they're also really powerful. They can interpret millions of instructions per second. Even so, there are some computational problems that are so complex that a powerful microprocessor would require years to solve them.

Computer scientists use different approaches to address this problem. One potential approach is to push for more powerful microprocessors. Usually this means finding ways to fit more **transistors** on a microprocessor chip. Computer engineers are already building microprocessors with transistors that are only a few dozen **nanometers** wide. How small is a nanometer? It's one-billionth of a meter. A red blood cell has a diameter of 2,500 nanometers -- the width of modern transistors is a fraction of that size.

Building more powerful microprocessors requires an intense and expensive production process. Some computational problems take years to solve even with the benefit of a more powerful microprocessor. Partly because of these factors, computer scientists sometimes use a different approach: **parallel processing**.

In general, parallel processing means that at least two microprocessors handle parts of an overall task. The concept is pretty simple: A computer scientist divides a complex problem into component parts using special software specifically designed for the task. He or she then assigns each component part to a dedicated processor. Each processor solves its part of the overall computational problem. The software reassembles the data to reach the end conclusion of the original complex problem.

It's a high-tech way of saying that it's easier to get work done if you can share the load. You could divide the load up among different processors housed in the same computer, or you could network several computers together and divide the load up among all of them. There are several ways to achieve the same goal.

To understand parallel processing, we need to look at the four basic programming models. Computer scientists define these models based on two factors: the number of **instruction streams** and the number of **data streams** the computer handles. Instruction streams are algorithms. An algorithm is just a series of steps designed to solve a particular problem. Data streams are information pulled from computer memory used as input values to the algorithms. The processor plugs the values from the data stream into the algorithms from the instruction stream. Then, it initiates the operation to obtain a result.

**Single Instruction, Single Data** (**SISD**) computers have one processor that handles one algorithm using one source of data at a time. The computer tackles and processes each task in order, and so sometimes people use the word "**sequential**" to describe SISD computers. They aren't capable of performing parallel processing on their own.

**Multiple Instruction, Single Data** (**MISD**) computers have multiple processors. Each processor uses a different algorithm but uses the same shared input data. MISD computers can analyze the same set of data using several different operations at the same time. The number of operations depends upon the number of processors. There aren't many actual examples of MISD computers, partly because the problems an MISD computer can calculate are uncommon and specialized.

**Single Instruction, Multiple Data** (**SIMD**) computers have several processors that follow the same set of instructions, but each processor inputs different data into those instructions. SIMD computers run different data through the same algorithm. This can be useful for analyzing large chunks of data based on the same criteria. Many complex computational problems don't fit this model.

**Multiple Instruction, Multiple Data** (**MIMD**) computers have multiple processors, each capable of accepting its own instruction stream independently from the others. Each processor also pulls data from a separate data stream. An MIMD computer can execute several different processes at once. MIMD computers are more flexible than SIMD or MISD computers, but it's more difficult to create the complex algorithms that make these computers work. **Single Program, Multiple Data** (**SPMD**) systems are a subset of MIMDs. An SPMD computer is structured like an MIMD, but it runs the same set of instructions across all processors.

Out of these four, SIMD and MIMD computers are the most common models in parallel processing systems. While SISD computers aren't able to perform parallel processing on their own, it's possible to network several of them together into a **cluster**. Each computer's CPU can act as a processor in a larger parallel system. Together, the computers act like a single supercomputer. This technique has its own name: **grid computing**. Like MIMD computers, a grid computing system can be very flexible with the right software.

Individually, each processor works the same as any other microprocessor. The processors act on instructions written in **assembly language**. Based on these instructions, the processors perform mathematical operations on data pulled from computer memory. The processors can also move data to a different memory location.

In a sequential system, it's not a problem if data values change as a result of a processor operation. The processor can incorporate the new value into future processes and carry on. In a parallel system, changes in values can be problematic. If multiple processors are working from the same data but the data's values change over time, the conflicting values can cause the system to falter or crash. To prevent this, many parallel processing systems use some form of **messaging** between processors.

Processors rely on software to send and receive messages. The software allows a processor to communicate information to other processors. By exchanging messages, processors can adjust data values and stay in sync with one another. This is important because once all processors finish their tasks, the CPU must reassemble all the individual solutions into an overall solution for the original computational problem.