number of components in the most complex airplane. Modern engineering methods are not really up to designing objects of such complexity. A modern computer can have literally millions of logical operations going on simultaneously, and it is impossible to anticipate the consequences of every possible combination of events. The methods of functional abstraction described in the previous chapters help keep the interactions under control, but these abstractions depend on everything interacting as expected. When unanticipated interactions occur (and they do), the assumption on which the abstraction rests breaks down, and the consequences can be catastrophic. In a practical sense, the behavior of a large computer system, even if no failures occur, is sometimes unpredictable—and this is the overarching reason that it is impossible to design a perfectly reliable computer.

CHAPTER 7

SPEED: PARALLEL COMPUTERS

Besides differing in the amount of memory, one type of universal computer can differ from another in the speed with which its operations are carried out. The speed of a computer is generally governed by the amount of time it takes to move data in and out of memory.

The sort of computers we have been talking about so far are sequential computers—they operate on one word of data at a time. The reason that conventional computers operate this way is largely historical. In the late 1940s and early 1950s, when computers were being developed, the switching elements (relays and vacuum tubes) were expensive but relatively fast. The memory elements (mercury delay lines, magnetic drums) were relatively cheap and slow. They were especially suitable for producing sequential streams of data. Computers were designed to keep the expensive processor switches as usefully busy as possible, while not placing too much demand on the speed of the memory. These early computers were room-size, with the expensive processor on one side and the slow memory on the other. A trickle of data flowed between.

As computer technology improved, software grew increasingly complex and expensive and it became increasingly difficult to train programmers, so in order to preserve

the investment in software and training, the basic structure of computers remained unchanged. There was little motivation to rethink the two-part design, because technology was progressing at such a rapid rate that it was easy to build faster and cheaper machines just by re-creating the same kind of computer in a new technology.

The speed of computers tended to double every two years. Vacuum tubes were replaced by transistors and eventually by integrated circuits. Delay-line memories were replaced by magnetic-core memories, and then also by integrated circuits. The room-size machines shrank to a silicon chip the size of a thumbnail. Through all this changing technology, the simple processor-connected-to-memory design stayed the same. If you look at a modern single-chip computer under a microscope, you can still see the vestiges of the room full of vacuum tubes: one area of the chip is devoted to processing, another to memory. This is true in spite of the fact that the processor and memory parts of the computer are now made by the same methods, often on the same piece of silicon. The patterns of activity are still optimized for the earlier, two-part design. The section of the silicon chip that implements processing is kept very busy; the part that implements memory still trickles out data one word at a time.

The flow of data between processor and memory is the bottleneck of a sequential computer. The root problem is that the memory is designed to access a single location on each instruction cycle. As long as we stick with this fundamental design, the only way to increase the speed of the computer is to reduce the cycle time. For many years, the cycle time of computers could be reduced by increasing the speed of the switches: faster switches led to faster computers. Now this is no longer such an effective strategy. The speed of today's large computers is limited primarily by the time required to propagate information along their wires, and that in turn is limited by the finite speed of light. Light travels at about a foot per nanosecond (a billionth of a second). The cycle time

of the fastest computers today is about a nanosecond, and it is no coincidence that their processors are less than a foot across. We are approaching the limits of how much we can speed up the computer without changing the basic design.

PARALLELISM

To work any faster, today's computers need to do more than one operation at once. We can accomplish this by breaking up the computer memory into lots of little memories and giving each its own processor. Such a machine is called a *parallel computer*. Parallel computers are practical because of the low cost and small size of microprocessors. We can build a parallel computer by hooking together dozens, hundreds, or even thousands of these smaller processors. The fastest computers in the world are *massively parallel* computers, which use thousands or even tens of thousands of processors.

As I have earlier described, computers are constructed in a hierarchy of building blocks, with each level of the hierarchy serving as the step to the level above. Parallel computers are the obvious next level in this scheme, with the computers themselves as building blocks. Such a construction can be called either a parallel computer or a *computer network*. The distinction between the two is somewhat arbitrary and has more to do with how the system is used than how it works. Usually a parallel computer is in one location, whereas a network is spread out geographically, but there are exceptions in both these rules. Generally, if a group of connected computers is used in a coordinated fashion, we call it a parallel computer. If the computers are used somewhat independently, we call the connected computers a computer network.

Putting a large number of computers together to achieve greater speed seems like an obvious step, but for many years

the consensus among computer scientists was that it would work for only a few applications. I spent a large part of my early career arguing with people who believed it was impractical, or even impossible, to build and program general-purpose massively parallel computers. This widespread skepticism was based on two misapprehensions—one about how complex such a system would have to be, and the other about how the components of such a system would work together.

Scientists tended to overestimate the complexity of parallel computers, because they underestimated—or at least underappreciated—the rate of improvement in microelectronics-fabrication technology. It was not so much that they were ignorant of this trend as that the rate of technological change was unprecedented in history; it was thus extremely difficult for expectations and intuitions to keep abreast of the change. I remember giving a talk at a computer conference at the New York Hilton in the mid 1970s, in which I pointed out that current trends indicated that there would soon be more microprocessors than people in the United States. This was considered an outrageous extrapolation at the time. Although microprocessors had already been produced, the popular view of a computer was still that of a large set of refrigerator-size cabinets with blinking lights. In the question period at the end of the talk, one of my ill-disposed listeners asked, in a voice laden with sarcasm, "Just what do you think people are going to do with all these computers? It's not as if you needed a computer in every doorknob!" The audience burst out laughing, and I was at a loss for an answer, but as a matter of fact in that same hotel today, every doorknob contains a microprocessor that controls the lock.

Another reason that people were skeptical about parallel computers was more subtle, and also more valid. It was the perceived inefficiency of breaking a computation into many concurrent parts. This problem continues to limit the application of parallel computers today, but it is not nearly as serious a limitation as it was once thought to be. Part of the

reason for the overestimation of the difficulty was a series of misleading experiences with early parallel machines. The first parallel digital computers were built in the 1960s by connecting two or three large sequential computers. In most cases, the multiple processing units shared a single memory, so that each of the processors could access the same data. These early parallel computers were usually programmed to give each processor a different task: for example, in a database application, one processor would retrieve the records, another processor would tabulate statistics, and the third would print out the results. The processors were used much as different operators on an assembly line, each doing one stage of the computation.

There were several inefficiencies inherent in this scheme, all of which seemed to grow along with the number of processors. One inefficiency was that the task had to be divided up into more or less independent stages. While it was often possible to divide tasks into two or three stages, it was difficult to see how to divide one task into ten or a hundred stages. As one detractor of parallel processing explained it to a newspaper reporter, "Well, two reporters might be able to write a newspaper article faster by having one of them gather the news while another writes the story, but a hundred reporters working on the article together would probably not be able to get it written at all." Such arguments were pretty convincing.

Another inefficiency stemmed from the shared access to a single memory. A typical memory could retrieve only one word at a time from a given area of memory, and this limitation on the rate of access created an obvious bottleneck in the system, which limited its performance. If more processors were added to a system already limited by its fetch rate, the processors would all end up spending more time waiting for data, and the efficiency of the system would decrease.

Moreover, processors had to be careful not to create an inconsistency by altering data that another processor was

looking at. For example, consider an airline-reservation system. If one processor is working on the problem of reserving seats, it looks to see if a seat is empty and then reserves the seat if it is. If two processors are booking seats for two different passengers simultaneously, there is a potential problem: they may both notice that the same seat is empty and decide to reserve it before either has a chance to mark it as taken. To avoid this kind of mishap, a processor had to go through an elaborate sequence that locked out other processors from accessing data while that processor was looking at a data word. This further aggravated the inefficiencies attendant on the competition for the system's memory, and in the worst case it would reduce the speed of a multiprocessor system to the speed of a single processor—and even to less than the speed of a single processor. As noted, these inefficiencies worsened as the number of processors increased.

The final source of inefficiency appeared to be even more fundamental: the difficulty of balancing the tasks assigned to the various processors. To return to the assembly-line analogy: we can see that the rate of computation will be governed by the speed of the slowest step. If there is just one slow operation, the rate of computation is set by that single operation. It is not unreasonable to expect that in this case, too, the efficiency of the system will decrease as we increase the number of processors.

The best formulation of these inefficiency problems is known as Amdahl's Law, after Gene Amdahl, the computer designer who came up with it in the 1960s. Amdahl's argument went something like this: There will always be a part of the computation which is inherently sequential—work that can be done by only one processor at a time. Even if only 10 percent of the computation is inherently sequential, no matter how much you speed up the remaining 90 percent, the computation as a whole will never speed up by more than a factor of 10. The processors working on the 90 percent that can be done in parallel will end up waiting for the single

processor to finish the sequential 10 percent of the task. This argument suggests that a parallel computer with 1,000 processors will be extremely inefficient, since it will be only about ten times faster than a single processor. When I was trying to get funding to build my first parallel computer—a massively parallel computer with 64,000 processors—the first question I usually got at the end of one of my pitches was "Haven't you ever heard of Amdahl's Law?"

Of course, I *had* heard of Amdahl's Law, and to tell you the truth I saw nothing wrong with the reasoning behind it. Yet I knew for certain, even though I couldn't prove it, that Amdahl's Law did not apply to the problems I was trying to solve. The reason I was so confident is that the problems I was working on had already been solved on a massively parallel processor—the human brain. I was a student at the Artificial Intelligence Laboratory at MIT, and I wanted to build a machine that could think.

When I first visited the MIT Artificial Intelligence Lab as a freshman undergraduate in 1974, the field of AI (as it was coming to be known) was in a state of explosive growth. The first programs that could follow simple instructions written in plain English were being developed, and a computer that understood human speech seemed just around the corner. Computers were excelling at games like chess, which had been considered too complicated for them just a few years earlier. Artificial vision systems were recognizing simple objects, like line drawings and piles of blocks. Computers were even passing calculus exams and solving simple analogy problems taken from IQ tests. Could general-purpose machine intelligence be all that far off?

But by the time I joined the laboratory as a graduate student a few years later, the problems were looking more difficult. The simple demonstrations had turned out to be just that. Although lots of new principles and powerful methods had been invented, applying them to larger, more complicated problems didn't seem to work. At least part of the

problem lay with the speed limitations of computers. AI researchers found it unfruitful to extend their experiments to cases involving more data, because the computers were already slow, and adding more data just made them slower. It was frustrating, for example, to try to get a machine to recognize a pile of objects when recognizing a single object required hours of computing time.

The computers were slow because they were sequential; they could do only one thing at a time. A computer must look at a picture pixel by pixel; by contrast, a brain perceives an entire picture instantly and can simultaneously match what it sees to every image it knows. For this reason, a human being is much faster than a computer at recognizing objects, even though the neurons in the human visual system are much slower than the transistors in the computer. This difference in design inspired me, as it did many others, to look for ways of designing massively parallel computers—computers that could perform millions of operations concurrently and exploit parallelism more like the brain does. Because I knew that the brain was able to get fast performance from slow components, I also knew that Amdahl's Law did not always apply.

I now understand the flaw in Amdahl's argument. It lies in the assumption that a fixed portion of the computation, even just 10 percent, *must* be sequential. This estimate sounds plausible, but it turns out not to be true of most large computations. The false intuition came from a misunderstanding about how parallel processors would be used. The crux of the issue is in how the work of a computation is divided among the processors: it might seem at first that the best way to divide a computation among several processors is to give each a different part of the program to execute. This works to an extent, but (as the aforementioned journalistic analogy suggests) it suffers from the same drawbacks as assigning a task to a team of people: much of the potential concurrency is lost in the problems associated with coordination. Programming a computer by breaking up the program is like having a large team of people paint a fence by assigning one person to the job of opening the paint, another to preparing the surface, another to applying the paint, and another to cleaning the brushes. This functional breakdown requires a high degree of coordination, and after a certain point adding more people doesn't help speed up the task.

A more efficient way to use a parallel computer is to have each processor perform similar work, but on a different section of the data. This so-called *data parallel decomposition* of the task is like getting a fence painted by assigning each worker a separate section of fence. Not all problems break up as easily as painting a fence, but where large computations are concerned this method works surprisingly well. For instance, image-processing tasks can be composed in concurrent parts by assigning a little patch of the image to each processor. A search problem, like playing chess, can be decomposed by having each processor simultaneously search different lines of play. In these examples, the degree of speed-up achieved is almost proportional to the number of processors—so the more of them the better. A little additional time must be spent dividing the problem among the processors and gluing the answers together, but if the problem is large the computation can be performed very efficiently, even on a parallel machine with tens of thousands of processors.

The computations just described can fairly obviously be decomposed to run in parallel, but data parallel decomposition also works on more complicated tasks. There are surprisingly few large problems that cannot be handled efficiently by parallel processing. Even problems that most people think are inherently sequential can usually be solved efficiently on a parallel computer. An example is the *chain-following problem*. My children play a game called Treasure Hunt, which is based on the chain-following problem. I give them a piece of paper with a clue about where the next clue

is hidden. That clue leads to the next clue, and so on, until they reach the treasure at the end. In the computational version of this game, the program is given, as input, the address of a location in memory containing the address of another location. That location contains the address of still another; and so on. Eventually, the address specifies a memory location containing a special word that indicates it is the end of the chain. The problem is to find the last location from the first.

Initially, the chain-following problem looks like the epitome of an inherently sequential computation, because there seems to be no way for the computer to find the last location in the chain without following the linked addresses through the entire chain. The computer has to look at the first location to find the address of the second, then look at the second to find the address of the third, and so on. It turns out, however, that the problem can be solved in parallel. A parallel computer with a million processors can find the last element in a chain of a million addresses in twenty steps.

The trick is to divide the problem in half at every step, a bit like the approach used in the sorting algorithms in chapter 5. Assume that each of the million memory locations has its own processor, which can send a message to any other processor. To find the end of the chain, every processor begins by sending its own address to the processor that follows it in the chain—that is, the processor whose address is stored in its memory location. Each processor then knows the address not only of the processor that comes after it but also of the one that precedes it. The processor uses this information to send the address of its successor to its predecessor. Now each processor knows the address of the processor that lies two ahead of it in the chain, so now the chain connecting the first processor to the last processor is half the length it was originally. This reduction step is then repeated, and each time it is repeated, the length of the chain is halved. After twenty iterations of the reduction step, the first processor in a million-memory chain knows the address of the last. Similar methods can be applied to many other tasks that seem inherently sequential.

As of this writing, parallel computers are still relatively new, and it is not yet understood what other types of problems can be decomposed to efficiently take advantage of many processors. A rule of thumb seems to be that parallelism works best on problems with large amounts of data, because if there are lots of data elements there is plenty of similar work to divide among processors.

One reason that most computations can be decomposed into concurrent subproblems is that most computations are models of the physical world. Computations based on physical models can operate in parallel because the physical world operates in parallel. Computer graphics images, for example, are often synthesized by an algorithm that simulates the physical process of light reflecting off the surfaces of objects. The picture is drawn from a mathematical description of the shapes of the objects by calculating how each ray of light would bounce from surface to surface while traveling from the source to the eye. The calculation of all the light rays can proceed concurrently, because the bouncing of light proceeds concurrently in the physical world.

A typical example of the kind of computation well suited to a parallel computer is a simulation of the atmosphere, used in predicting the weather. The three-dimensional array of numbers that represents the atmosphere is analogous to three-dimensional physical space. Each number specifies the physical parameter of a certain volume of air—for example, the pressure in a cube of atmosphere 1 kilometer on a side. Each of these cubes will be represented by a few numbers specifying average temperature, pressure, wind velocity, and humidity. To predict how the volume of air in one such cube will evolve, the computer calculates how air flows between neighboring volumes: for instance, if more air flows into a volume than out of it, then the pressure in that volume will

increase. The computer also calculates the changes attributable to factors such as sunlight and evaporation. The atmospheric simulation is calculated in a series of steps, each of which corresponds to a small increment of time—say, half an hour—so that the flow of simulated air and water among the cells in the array is analogous to the flow of real air and water in the pattern of weather. The result is a kind of three-dimensional moving picture inside the computer—a picture that behaves according to physical laws.

Of course, the accuracy of this simulation will depend both on the resolution and the accuracy of the three-dimensional image, which accounts for the notorious inaccuracy of weather predictions over time. If the resolution of the model is increased and the initial conditions are measured more accurately, then the prediction will be better—although even a very high resolution will never be perfect over a long period of time, because the initial state of the atmosphere cannot be measured with exact precision. Like the game of roulette, weather systems are chaotic, so a small change in initial conditions will produce a significant change in the outcome. On a parallel computer, each processor can be assigned the responsibility for predicting the weather in a tiny area. When wind blows from one area to the next, then the processors modeling those areas must communicate. Processors modeling geographically separated areas can proceed almost independently, in parallel, because the weather in these areas is almost independent. The computation is local and concurrent, because the physics that governs the weather is also local and concurrent.

While the weather simulation is linked to physical law in an obvious manner, many other computations are linked more subtly to the physical world. For instance, calculating telephone bills is concurrent, because telephones (and telephone customers) operate independently in the physical world. The only problems we don't know how to solve efficiently on a parallel computer are those for which the growing dimension of the problem is analogous to the passage of time. An example is the problem of predicting the future positions of the planets. (Ironically, this is the very problem for which many of our mathematical tools of computation were originally invented.)

The paths of the planets are the consequence of well-defined rules of momentum and gravitational interactions between the nine planets and the Sun. (For simplicity's sake, we will ignore the effects of small bodies, such as moons and asteroids.) All the information necessary to solve the problem can be represented by nine coordinates, so there isn't much data. The computational difficulty of the problem comes from the fact that the calculation must be made, as far as we know, by calculating the successive positions of the planets at each of billions of tiny steps, each representing a short period of time. The only way we know how to calculate the positions of the planets a million years in the future is to calculate their position at each intermediate time between now and then. If there is a trick to solving this problem concurrently, such as the one used in the chain-following problem, I am not aware of it. On the other hand, as far as I know, no one has proved that this orbit problem is inherently sequential. It remains an open question.

Highly parallel computers are now fairly common. They are used mostly in very large numerical calculations (like the weather simulation) or in large database calculations, such as extracting marketing data from credit card transactions. Since parallel computers are built of the same parts as personal computers, they are likely to become less expensive and more common with time. One of the most interesting parallel computers today is the one that is emerging almost by accident from the networking of sequential machines. The worldwide network of computers called the Internet is still used primarily as a communications system for people. The computers act mostly as a medium—storing and delivering information (like electronic mail) that is meaningful only to

humans. I am convinced that this will change. Already standards are beginning to emerge that allow these computers to exchange programs as well as data. The computers on the Internet, working together, have a potential computational capability that far surpasses any individual computer that has ever been constructed.

I believe that eventually the Internet will grow to include the computers embedded in telephone systems, automobiles, and simple home appliances. Such machines will read their inputs directly from the physical world rather than relying on humans as intermediaries. As the information available on the Internet becomes richer, and the types of interaction among the connected computers become more complex, I expect that the Internet will begin to exhibit *emergent behavior* going beyond any that has been explicitly programmed into the system. In fact the Internet is already beginning to show signs of emergent behavior, but so far most of it is pretty simple: plagues of computer viruses and unpredicted patterns of message routing. As computers on the network begin to exchange interacting programs instead of just electronic mail, I suspect that the Internet will start to behave less like a network and more like a parallel computer. I suspect that the emergent behavior of the Internet will get a good deal more interesting.

CHAPTER 8

COMPUTERS THAT LEARN AND ADAPT

The computer programs I have so far described operate according to fixed rules supplied by the programmer. They have no way of inventing new rules themselves, or of improving the ones they are given. The chess-playing programs, if their programmers do not tinker with them, will keep making the same mistakes over and over, no matter how many games of chess they play. In this sense, computers are completely predictable; it is in this sense that computers can "do only what they are programmed to do"—a point often raised by the defenders of humankind in the "man vs. machine" debate.

But not all software is this inflexible. It is possible to write programs that improve with experience. When they operate with such programs, computers can learn from their mistakes and correct their own errors. They accomplish this by making use of *feedback*. Any system based on feedback needs three kinds of information:

1. What is the desired state (the *goal*)?
2. What is the difference between the current state and the desired state (the *error*)?
3. What actions will reduce the difference between the current state and the goal state (the *response*).