PARALLEL COMPUTING AS A COMMODITY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

David Spencer Pearson

January 1998

PARALLEL COMPUTING AS A COMMODITY

David Spencer Pearson, Ph.D.
Cornell University 1998

Massively parallel computers have become undisputed champions in the supercomputing arena. The global computer industry, however, is increasingly dominated by consumer machines. In this thesis, we argue that everyday computers must become highly parallel machines in order to sustain the performance gains we have come to expect.

For parallel computation to become a commodity, there must be an architecture which can be scaled as easily as memory arrays are now. We also need to establish that parallelism can benefit everyday applications and that operating systems for such machines can provide as comfortable and robust an environment as we have on sequential machines.

We introduce an architecture based on cellular automata—meshes of simple, locally-connected processors in either 2 or 3 dimensions. We argue that this architecture is easily scalable, and from a theoretical viewpoint it is essentially as efficient as any other scalable architecture. We show two instances of this architecture, a simple one implemented in silicon and a more complex one implemented through a simulator.

To show the viability of this architecture for everyday tasks, we have developed fast parallel algorithms for RSA encryption (using residue number systems and a new method for converting one residue number system to another) and for document formatting (using a space-filling curve for data layout to achieve optimal $O(\sqrt[d]{n})$ running time on a d-dimensional mesh).

We also describe the design of an operating system for a cellular array based on the notion of the OS as the periphery, rather than the kernel, and show several advantages in security and performance that this confers. Finally, we investigate the 3-dimensional dynamic allocation problem faced by such a system. This problem is NP-hard even in its static form, but we describe a simple best-fit allocator that works well in practice.

Biographical Sketch

David Pearson grew up on an apple farm in upstate New York, where he developed a keen love of nature, long walks, and of course, apples. His educational journey started in England, then moved back to an American kindergarten, to which he foolishly wore the shorts, blazer and red tie of the English school. He attended a now-defunct school a short walk from Niagara Falls, and continued on to Dartmouth College at age 16.

At Dartmouth, David officially studied mathematics, while learning all he could about computers and working as a student systems programmer on Dartmouth's home-grown time-sharing system. A semester at Edinburgh University studying philosophy also gave him his first taste of the rarefied air of academic computer science.

After graduating from Dartmouth in 1975, David worked for 14 years as a software engineer, first for Data General, where he helped develop the AOS operating system, then for Dartmouth, where he wrote the software for an early local-area network and learned how to be calm in a crisis. In 1983, he co-founded True Basic, a small software company which aimed to undo some of the harm Bill Gates did when he popularized such a handicapped language. Despite teaming up with the original inventors of Basic, True Basic was no match for Microsoft.

Wanting to solve work on harder problems, David came to Cornell. In Ithaca, he met the love of his life, Susan. They were married in 1993 and David made an instant transition from bachelorhood to being in a family with two wonderful children.

In 1997 David left the ivory tower and ventured once more into industry, like Rip van Winkle awakening to find the world changed beyond recognition. He currently works for BBN in Cambridge, Massachusetts.

To Susan

Acknowledgements

I owe a great debt of gratitude to my graduate committee, Dexter Kozen, Keshav Pingali, Lov Grover, and Adam Bojanczyk, for supporting this research even when it went off the beaten track. Dexter Kozen's insightful mind and his enthusiasm kept computer science exciting for me, and his careful reading greatly improved this presentation. Adam Webber and Bob Durie joined me in a very stimulating and productive project to implement the prototype VLSI cellular array described in section 4.1.

In a time when academic research often seems to be struggling to catch up with the R&D being done in industry, I have been particularly fortunate to be able to pursue research that looks forward 20 years or more. I am grateful to the Fannie and John Hertz foundation for the support of a graduate fellowship, and to the Cornell Computer Science department for allowing me to pursue this research.

I am also grateful to those friends who first set me on the road to graduate school, getting me interested in academic research by involving me in some fascinating projects—S. U. Pillai, Peter Doyle, David Epstein, John Kemeny, Chip Elliott, and Stan Dunten. Several friends, climbing partners and housemates helped me stay sane and (generally) good-natured through my time at Cornell: Jim Caldwell (who deserves everlasting gratitude for introducing me to Susan), Kate White, Amy Briggs, Ben Hao, Greg Kimmel, and Vicky Chang. My children, Claire and Ethan, didn't shorten the process, but made it much more enjoyable.

Finally, to my beloved, Susan, my gratitude is inexpressible—her love, insight, and wisdom run through this entire work. She always brings me back to the meeting place of vision and practicality, between the ideal and the real, where magic can be found.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> Where there is no vision, the people perish
>
> — Proverbs 29:18

What if the computer on your desk were a massively parallel machine, capable of performing thousands or millions of operations simultaneously, instead of a conventional sequential computer, operating one step at a time? Would it be more useful? Would it provide as comfortable a computing environment as we now have? What could that machine do better, and what would it do worse? And why don't we have ubiquitous parallel computers now?

This thesis is an investigation of those questions, motivated by the belief that eventually, sometime in the twenty-first century, parallel computers will be as ubiquitous as standard sequential machines are now. The microelectronic revolution, in the 50 year history of electronic computers, has delivered far more computational power on a smart credit card than the Eniac, filling a small warehouse, did 50 years ago. What seems almost as astonishing is the fact that the basic architecture of the stored-program computer, outlined by von Neumann just when the Eniac was being completed, has hardly changed at all in the same time.

If we peer into the future, however, there are clear reasons for the architecture to change. While shrinking feature sizes continually give us more logic gates to use, they also give us the increasingly arduous design task of using the extra gates to achieve higher performance. If we pack 100 times as many components on a chip as a Pentium II, will there be enough engineers in the world to design that chip? There will, of course, but the design will be forced toward greater simplicity and regularity. Instead of having one extremely complex processor on the chip, we will have a large number of simpler processors.

The research I describe is guided by a vision of computing in 2050. In the vision, the computer will be a crystal lattice, a 3-dimensional array of identical molecular-scale cells, each capable of simple computation, and each communicating only with its immediate neighbors in the array. The function of each cell will be dictated more by physics than by engineering. The cells will act more like programmable logic gates than conventional CPUs. Programs will be like special-purpose computers constructed out of these gates.

I don't claim to be enough of a futurist to spell out the shape of things to come so clearly. This scenario is likely to be wrong in innumerable ways, some quite significant. The future has an uncanny way of evading our expectations. But I will still claim that certain features of this vision are likely to be realized: specifically, that the dominant computer architecture for general applications will be a highly parallel machine, consisting of very simple processors with their own small memories, connected in a mesh-like structure.

This structure is not new. It is, in fact, nearly as old as the dominant sequential architecture, and ironically it was first described by the same person, John von Neumann. He called the mesh of identical cells a "cellular automaton." He apparently never envisioned cellular automata as a way to build practical machines, only as a simplified mathematical model in which to demonstrate the possibility of a machine which could emulate a living organism's miraculous ability to reproduce itself. In this work, I will use the term cellular automaton when describing theoretical (infinite) grids, and use "cellular array" to describe practical computers based on the same concept; the abbreviation CA can stand for either cellular automata or cellular arrays, when it's clear from context.

1.1 Research goals

This vision requires some justification. Many parallel computing architectures have been proposed and implemented—the mesh is only one of them. I will argue, from both a practical and theoretical standpoint, for the benefits of the mesh. This main virtue of the mesh is *scalability*, the ability to increase the number of processors without running into bottlenecks or limitations. Scalability becomes a necessity if we wish to find architectures that will gracefully extend into the realm of nanometer-scale devices.

The other part of the vision that flies in the face of accepted wisdom is that massively parallel machines will become the standard computing platform. In the past, when parallel machines have been proposed, they have been specifically targeted to a particular class of problems, a niche that they can fill better than any other architecture. Historically, all those niches have later closed up, and the machines

that filled them have been overtaken by the relentless march of faster sequential machines.

I believe that parallel computers will have lasting success only when they become commodities, and can partake of the same economies of scale that standard microprocessors enjoy today. In order to gain commodity status, it is not enough for them to do something extremely well—they must be able to do *everything* reasonably well. The goal of this research is to show that simple, scalable parallel computers can become a general-purpose computational substrate, useful for everyday computational tasks. This is, in part, a manifesto, arguing for the desirability of that goal, and in part a research report, showing the feasibility of reaching it.

Several interrelated themes run through both the theoretical and practical sides of this thesis:

- Bringing the idea of space into the study of algorithms. Space has unfortunately been often used to mean "amount of memory;" here I mean real, 3-dimensional space. The layout of data becomes important in cellular arrays, and geometry becomes a useful tool for software design.

- Removing the distinction between hardware and software. In the past, hardware designers had to pay attention to layout, physical proximity, and signal propagation speed. The software designer worked in an abstract mathematical world in which these issues never arose. The price to be paid was giving up scalability and the inherent parallelism of hardware.

- Becoming responsible as algorithm designers for all the resources we use. We write programs according to several convenient fictions now: that memory is infinite, that randomly accessing a large memory costs no more than repeatedly accessing the same location, or that communication through a shared memory costs no more than accessing a local memory. Hardware designers struggle to maintain these fictions, with complex hardware that hides the costs in most cases. By writing algorithms with physics in mind, we can get better solutions by taking the true costs into account.

- Finding the parallelism in everyday problems. Designing parallel algorithms for these problems has striking benefits, and is much easier than we might have expected from the history of parallel supercomputers. It seems that everyday problems may, simply because they are less complex, be easier to parallelize.

- Making computers truly general-purpose. As we make parallel computation available to software, it reduces the need for special-purpose hardware. The original vision of computers was that they were universal machines, that they

could perform any kind of control. But there are now an increasing array of specialized controllers on off-the-shelf PCs, not because the main processor isn't capable of doing the same job, just because adding extra hardware allows the task to be performed in parallel with other work.

- Keeping the hardware simple. The more complex the hardware becomes, the more specific it is. Complex hardware will do certain tasks well, but others will suffer. Simple hardware retains generality—when more complex structures are needed, they can be provided with software (which, because of the parallelism, should be able to at least approximate the speed of a hardware solution).

1.2 Research contributions

The results of this research are somewhat eclectic. This is intentional—necessary, in fact, given the broad goal: to establish the feasibility and the usefulness of massively parallel computers as the architecture of choice for everyday computing applications. I have chosen to attack several of the most problematical areas of that vision rather than concentrate in depth on one aspect. In this way, I hope, there is less chance that I have overlooked a major problem that renders the projected system useless. I have investigated:

- Theory, summarizing and extending arguments for the optimality of the cellular automaton. I also relate cellular automata to other models of parallel and sequential computation, and give some bounds on their relative speeds.
- Architecture, giving a design for a practical system with strictly nearest-neighbor communication, including a local solution to the re-booting problem.
- Operating system design, showing how efficiency and security both benefit by placing the operating system at the periphery of the array, rather than in the kernel of each processor. I tackle the problem of 3-dimensional dynamic allocation, and describe a simple best-fit allocation that works reasonably well in practice.
- Software replacements for specialized hardware. I present an implementation of RSA encryption which is faster than any current hardware implementation, assuming the same clock speed. It uses a new highly-parallel modular multiplication algorithm based on residue arithmetic.
- Application programs, giving a new parallel algorithm for document formatting using a space-filling curve for data layout to achieve optimal running time.

Chapter 2

Parallel computation

> Those who cannot remember the past are condemned to repeat it.
>
> — George Santayana, *Life of Reason*

In 1976 the Cray-1 was first introduced and took on the mantle of the world's most powerful supercomputer. It had an amazingly fast clock rate of 80 MHz, a great wealth of registers, 64-bit words, and 32 Mb of RAM. Except for the 64 bit words, it sounds much like a typical desktop computer from 1996, twenty years later.

In 1997, the world's fastest supercomputer is the just-delivered Intel ASCI Teraflop computer. It contains about 10,000 processors running at 200MHz, connected in a 2-dimensional mesh. Could this be what a desktop computer is like in 2017? That is close to what we are suggesting here.

Parallel computing has for years seemed poised to be the next wave in computers. In 1989, George Almasi wrote "parallel computing is ready to happen" [AG89]. Since that time, Thinking Machines (maker of the Connection Machine) and Kendall Square Research have gone out of business and MASPAR has turned into a software company. The entire field of supercomputers, which fueled much of the interest in parallel computation, has been shrinking.

This chapter will sketch, in broad strokes, the pros and cons of parallelism, showing why it has been so persistently attractive and why its success has been so limited. If we don't understand the obstacles, we are likely to run into the same obstacles ourselves.

2.1 The von Neumann bottleneck

The central feature of the conventional von Neumann architecture is a central processing unit (CPU), consisting of a control unit and an arithmetic and logic unit. The CPU communicates with input and output units and a memory system. It fetches data and instructions from the memory system, executing one instruction at a time in sequence. The instructions may write results back to memory, perform arithmetic or logical operations, or communicate with the I/O system.

This is called the von Neumann architecture because its principles were first explained in the "Preliminary report on the EDSAC," of which he was the sole author, but except for keeping the instructions in the memory system with the data, the idea of sequential instruction execution was already present in the ENIAC [Ste81]. So it is perhaps a bit unfair that the bottleneck inherent in sequential execution is named after von Neumann.

The von Neumann bottleneck is the observation that of all the logic in the computer, only a small fraction is active in processing the single instruction that is currently in execution. The total computational power of the logic could be approximately measured by the number of gates times their switching frequency. In early computers only a few percent of the logic was involved in an instruction, a fraction which roughly measures how much of the potential power of the machine was being utilized. Now, despite a great increase in the amount of active logic through aggressive use of pipelining, wider data paths, and out-of-order execution, the ratio is much lower, often less than .1%. Although the CPU design may keep 100,000 gates active, a 128Mb memory contains over one billion gates.

The reason the bottleneck keeps getting worse is the relentless increase in the total amount of logic in the machine. That has been growing at an exponential rate, whereas the amount of active logic has been growing much more slowly. To increase the amount of active logic, the CPU designers have to exploit cleverness, finding hidden opportunities for parallelism in a machine which is, overall, sequential. The increase in total logic, on the other hand, has been following an exponential curve known as Moore's law.

Gordon Moore, one of the founders of Fairchild and Intel, made the observation that the amount of logic on a single chip was doubling every 2 years, and seemed to poised to do so for some time into the future. His first prediction, made in 1965, was that the amount of logic would double every year. At that time, circuit design and layout were still evolving, and better arrangement (which he called "cleverness") added an extra growth curve. In 1975, he reviewed his earlier forecast, and foresaw the waning influence of clever layout [Moo95]. The revised law has been extremely accurate. In some ways, it has become a self-fulfilling prophecy, a benchmark which the semiconductor industry is required to meet [Ass94].

Moore's law, then, keeps increasing the influence of the von Neumann bottleneck. But at the same time, it keeps increasing the computational power available, since smaller feature sizes can generally translate into faster switching times. This was first pointed out by another Intel founder, and the inventor of the first microprocessor, Bob Noyce. His observation, that the amount of computational power available per dollar keeps doubling each year, is sometimes called Noyce's thesis. Because the computers keep getting faster, we don't worry too much about their overall efficiency getting lower.

How long will Moore's law continue to hold? Without a radical departure from the technology of lithography, current estimates foresee it holding until 2010 [Ass94]. There are theoretical reasons why it cannot continue indefinitely (see section 3.3.1). Moore himself offers some nearer-term problems: to attain the higher densities has required growth in the capital costs of chip fabrication. These costs are rising faster than the revenue of the semiconductor business, so there is likely to be an economic limit to the growth in density before there is a technological one [Moo95].

If Moore's law slows down or ends completely, then the quest for more computational power will have to shift to the arena of parallelism.

2.2 The broken promise

Parallel computers promised to alleviate the von Neumann bottleneck. That fact has certainly not gone unnoticed—there have been many ambitious attempts to build practical parallel machines. Despite this, sequential machines seem to be only solidifying their dominance.

Let's look at some of the most prominent parallel architectures and what happened to them.

SIMD machines

The first large-scale parallel computer designed was the Illiac IV. It was conceived in the mid 1960s and the design was complete in 1967, but it wasn't operational until 1975. Its architecture was SIMD—single instruction-stream, multiple data-stream, meaning that each processor was executing the same instruction at the same time, but operating on its own local data. The problems for which it was intended—weather simulations, involving matrix arithmetic and differential equations on a grid of points—had a high degree of regularity, and mapped well onto the SIMD architecture. The 64 processing elements (one quarter the number originally designed) were connected in a 2-dimensional grid.

The Illiac IV did not give parallel processing a good name. The problem of writing parallel software was given short shrift until the machine was delivered, and software became its biggest stumbling block. In addition, the system suffered from reliability problems throughout its life. A great deal of new technology was pioneered during the project, which resulted in large delays and cost overruns—the final cost of the machine, $31 million, was 4 times the original estimate for 1/4 the number of processors.

Despite its troubles, for problems which adapted well to its architecture, the Illiac IV remained the fastest computer in the world until its retirement in 1981. Some of the technological ground it broke has had a great impact on more mainstream computers—it was the first mainframe to use emitter-coupled logic (ECL), and the first to use semiconductor memory.

Other somewhat less ambitious SIMD machines have performed quite well in niche markets, including the MPP from Goodyear Aerospace, the Connection Machine, and the MASPAR. None of these are still produced.

Dataflow machines

Dataflow architecture proposes to directly execute the computation tree which is implicit in a program. It is based on the observation that a great deal of the sequentiality of a program is not in the dependencies between the operations, but simply an artifact of writing a program in a linear way. Dataflow is a software-first approach to parallelism—the intention is that the programmer would write a program in the most natural way, and the machine would schedule the different operations at run-time to get the most parallelism.

To be able to identify the dependencies automatically in the compiler, the various dataflow groups created new languages, with side-effects and aliasing removed.

Despite many ambitious projects, no dataflow computers have achieved any significant success. The technology developed for compilers for dataflow machines, including compilers for conventional languages like Fortran, has been influential in high-performance compilers for other parallel and sequential machines.

VLIW

Very long instruction word machines (VLIW) were another software-driven approach. Taking some of the advanced dependency-analysis ideas of dataflow, a compiler could statically re-arrange the computations of a basic block and find a reasonable degree of parallelism. The parallelism would then be compiled directly into a long instruction word, one which contained fields for controlling 8 or more

execution units simultaneously. Although the degree of parallelism isn't large, it is much easier to find 8 operations to do simultaneously than 100.

The idea was commercially pioneered by the Multiflow corporation, now defunct. It was targeted to the mini-supercomputer market, a niche which closed up soon thereafter. Many engineers from the project later went to Hewlett-Packard, which has an active VLIW project.

VLIW has progressively moved down the price ladder—the newest VLIW processor on the market is a digital signal processing chip (DSP) from Texas Instruments, the TMS320C6x, an 8-way VLIW machine which currently ranks as the fastest DSP. It is used, for example, in many of the newest 56k modems. The evolution of VLIW from a supercomputer into a component in consumer electronics nicely captures the theme of this chapter.

2.3 What went wrong?

Many of the problems that have kept parallel computers from achieving commercial success are generic, applying to all the cases above. Each type of parallel machine has its own particular challenges, but on the whole, that is not what led to their demise. We will examine the major obstacles here.

Moore's law

Despite the fact that Moore's law keeps exacerbating the von Neumann bottleneck, it has played a significant role in the death of many promising parallel architectures. Most parallel machines are significantly more complex than their sequential competitors. The engineering effort is generally more expensive, both in money and time, particularly as the technology is often new. The result of this is that the parallel machines lag behind conventional machines by a few years. Because of Moore's law, this delay means that conventional machines have a chance to become faster and more powerful in the interim. The edge the parallel machine started with is partly or totally erased. The cost is also higher, both because of the engineering effort and the smaller market, and all these factors together make the parallel machine less cost-effective than the newer sequential machines.

Difficulty of programming

Parallel computers can only outperform sequential machines if they execute programs which exploit the parallelism. Often the most straightforward algorithm to solve some problem is inherently sequential, and a great deal of insight is needed to remove the sequential dependencies. Porting a program to a parallel machine

is likely to involve both creativity and hard work. Porting a program from one sequential machine to another, by contrast, is increasingly straightforward.

Algorithm design is not the only hard part of programming. To express the algorithm in a programming language is usually hard. Either the language is quite different, or it is a standard language augmented with a library of subroutines to permit communication between processes. A large amount of detailed bookkeeping is required, and the original calculation can get lost in the details. Bugs can be extremely subtle, but debugging tools are not as well developed in the parallel environment. Finally, the performance of the resulting program is often disappointing, requiring even more time to hand-optimize it, because the abstractions used by the language don't connect in an intuitive way with the real performance model of the machine.

Compounding the difficulty of programming is the fact that the demand for the high performance offered by parallel machines has often come from applications that are old, large, complex, unwieldy Fortran programs, which have been modified over the years to the point that they are not very well understood. This has motivated considerable research in compilers that automatically parallelize programs. These compilers have had some success, but seldom near the level of performance the machines are capable of.

Many programming models

Every class of parallel computers has a different programming model, and within the classes there are often quite significant differences. The result of these differences is that parallel programs do not port easily to different parallel machines. The software effort, already much larger than for sequential machines, has to be duplicated for every new parallel machine a program runs on. Generally parallel architectures haven't had very long lifespans, so just to keep up with current technology means rewriting the software every few years.

Some attempts have been made to address this problem, by providing standard languages (like high-performance Fortran), or standard libraries, with the intention that programs written to these specifications can all be ported by simply porting the compiler or the library. While these attempts have been somewhat successful, the standard model is in some ways a "least common denominator," in which programs aren't really optimized for any machine. The models are all coarse-grained, since fine-grained parallelism seems more difficult to abstract away from the machine, but fine-grained parallel machines have some of the highest degrees of parallelism, and for certain applications, some of the most impressive performance.

2.4 Commodity parallelism

The history of parallel computers up to this point is part of the history of supercomputers. But Moore's law, DARPA funding cuts and the economics of consumer electronics have almost eliminated the supercomputer industry. Within the much-diminished supercomputer business, it is true that the only significant developments are highly parallel machines, built out of commercial microprocessors. But we believe the real future of parallel computers will not be in supercomputers, but in computers for everyday use, in desktop machines, games, cellular phones—in any place we would use a standard microprocessor today.

Most of the generic problems for parallel computers that we listed above are a direct outgrowth of their supercomputer orientation. For example, if parallel machines become a commodity item, then they will no longer lag behind the current technology by a few years. The engineering costs become amortized over far more machines, and should no longer be a factor—indeed, we can make the case that engineering a high-performance parallel computer may be easier than engineering a chip like the Pentium II. Parallel programming will still be hard, but it is easier to write, say, a new highly parallel videoconferencing program than to parallelize a large partial differential equation solver. Supercomputing problems are difficult, complex ones—we will try to show in the rest of this work that parallelism is not so hard to find in everyday applications. Finally, the problem of too many programming models will go away in a mass market, once one or two architectures become dominant.

Moore's law again

Moore's law, as we pointed out earlier, has only exacerbated the von Neumann bottleneck, so its continuation over the next 10-15 years makes parallelism more attractive. But it has other effects that push us in the same direction.

First, Moore's law applies most directly to memory, since the density translates directly into more bits. Unfortunately, it doesn't translate into more speed. The speed of DRAM (dynamic RAM) has stayed relatively constant over the last 10 years. This means the memory access time will become increasingly non-uniform, an effect that is likely to exceed the ability of caches to hide. Locality is becoming of prime importance, a fact which is not reflected in our conceptual machine model or our programming languages.

Second, the processor chip may, in 14 years, be able to have 2^7 times as much logic on it, but Moore's law tells us nothing about what to do with that much logic. A great deal of the performance increase of current microprocessors is due, actually, to parallelism of various sorts on the processor: deep pipelines, fast multipliers,

floating-point coprocessors, out-of-order or speculative execution. This is just a grab-bag of techniques, and there is no systematic way to extend it to get still more performance. Smaller features will still increase the speed, but not as quickly as we have been accustomed to. The easy way to use this extra space is in populating the chip with several processors, instead of one gigantic one.

Third, even if there were a way to use the space to make a faster single-processor system, it would still be a major design problem. If we could put 100 million transistors on a chip, the design problem would be enormous. Already, only a few large corporations have the resources to design a chip like the Pentium II, with 7 million transistors. Achieving some sort of regularity, like that enjoyed by memory arrays, would be a major benefit of a parallel architecture.

Creeping parallelism

In proposing that desktop PCs become highly parallel machines, we are, perhaps, only stating the obvious. Personal computers are already highly parallel machines, but the parallelism is generally hidden. If we take a census of the general-purpose computers inside the box, in addition to the main processor we find a computer in the graphics card, one in the SCSI controller, a DSP chip in the modem (which, if the modem is a new one, may itself be a parallel machine), and one on the sound card. In addition, there is special-purpose parallelism in the numeric coprocessor and (in newer Intel processors) the multi-media extension, MMX, which is like a very small-scale SIMD machine.

All this shows that the mandate for parallelism is strong. If it can't come in the front door, it will come in the back. But the special-purpose parallelism of these separate devices is not as flexible or powerful as a homogeneous parallel computer, in which the processors are identical and can be reassigned as the need arises. But a uniform array of identical processors should be able to accelerate those special-purpose tasks.

What will we use the power for?

If we were to have so much parallel processing power in our desktop machines, what could we use it for? Parallel supercomputers have often seemed like solutions in search of a problem, and the problems they have proven good for—finite-element analysis, linear algebra, protein folding and the like—are not likely to be in high demand in home or office PCs.

We will explore some of the applications in the rest of this thesis. Good uses for large amounts of processing power are easy to find. Graphics, video, or virtual reality can absorb as much processing power as we can get for the near-term future.

So can audio, voice recognition and telephony. Simply eliminating the specialized controllers can give us lower-cost devices, much the same way that higher-speed processors have allowed cheaper laser printers which depend on the main computer for bitmapping. More speed can also allow for simpler programming interfaces, where multiple layers of software provide modularity, but at the same time degrade performance.

We have witnessed several orders of magnitude performance gains since computers were introduced, and there has been no shortage of uses for the power. Sometimes it seems as if a computer version of Parkinson's law is at work, with programs expanding to fill the available space *and* time. In any case, we can expect the demand for higher performance to continue, and we can meet the demand through parallelism.

The question—what to use all the power for—is a bit misleading, since it will have to be answered in any case. Moore's law will continue to give us increases in performance and amount of logic. If we don't have ubiquitous parallel computers, it will be a question for the hardware designers, a challenge to them to find a use for a billion transistors on a chip. We are just proposing to shift the solution from hardware to software, by making the hardware a general-purpose circuit.

2.5 Scalability

In order for parallel machines to achieve commodity status, they must be scalable, like DRAM. It should be as easy to add processing power to an existing machine as it is to add memory now, and it should be as easy to shrink the features of the design to make denser and more powerful chips.

The feature of memory that permits ever-denser chips is its regularity; that also gives it more than an order of magnitude advantage over microprocessors. So the processor arrays should be, as much as possible, completely regular. This regularity also reduces the design costs of the array.

This suggests that a mesh interconnection would be most appropriate, since it has the benefit of complete regularity. But there are additional reasons for using only local interconnection. Non-local interconnections require long wires. These degrade the performance in two ways—they take up valuable real estate on the chip (or the board), and they slow down the system. If their length measures in the inches, they can slow things down because of the speed of light. If they are shorter, there is still a speed penalty because of their capacitance; this can be overcome, but only at the expense of using more power to switch the line.

Cellular arrays seem like the best candidate for commodity parallel machines. They are cheap, regular, scalable, have excellent throughput, and can easily support

massive parallelism. As we shall see, there are also compelling theoretical reasons to believe they are the best architecture.

Chapter 3

The limits of parallelism

> He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.
>
> — Leonardo da Vinci

The central idea of this thesis is that cellular automata can serve as an excellent general-purpose computer architecture. The evidence for this is a blend of practical and theoretical considerations. Most of the following chapters concern practical issues and are devoted to showing that cellular automata can be useful, simple, high-performance computer systems. In this chapter, however, we will plunge into theory to make the argument that they are, under reasonable assumptions, the *most* efficient computers. More precisely, we will claim that cellular automata are asymptotically as efficient, within a constant factor, as any architecture that can be physically built.

This seems very surprising on the face of it, since the nearest-neighbor communications of cellular automata means that many simple problems, like finding the parity of a bit string, must take time proportional to $\sqrt[3]{n}$ on a 3-dimensional CA, whereas the time on a hypercube would be much faster: $\log n$. This is because of the higher connectivity of the hypercube—the longest end-to-end path (called the *diameter*) is only $\log n$ instead of $\sqrt[3]{n}$ on a 3-dimensional cellular automaton. This low diameter is precisely the reason that the hypercube model (along with many other common models of parallel computation) isn't scalable. As we increase the number of processors n without bound, the end-to-end communication time cannot grow as $\log n$, since the physical processors must occupy a volume proportional to n, which requires a physical diameter proportional (at least) to $\sqrt[3]{n}$. Because we

believe no communication signal can propagate faster than the speed of light, the only way to scale the hypercube model to larger numbers of processors is to slow down the clock.

In this chapter we formally define the cellular automaton model, and introduce a notion of computational universality for CAs which preserves efficiency. We then make the argument that 3-dimensional cellular automata are universal in this sense for all physical digital computers. We examine whether this argument can be extended beyond conventional digital logic to other novel devices—current evidence suggests that we cannot. We also relate cellular automata to other theoretical models of computation.

3.1 Cellular automata

The theoretical model we use is the d-dimensional cellular automaton. Cellular automata were introduced by John von Neumann [vN66], as a mathematical model in which to give an account of how self-reproduction is possible.

A large body of theory was developed through the 1960s and 70s [Bur70, Cod68, Ban71], but more recently, cellular automata have been more studied as theoretical tools in modeling other fields, including physics [Mar91, Wol86, CKCN93], geology, biology and economics.

3.1.1 Formal definition of cellular automata

For our purposes, we can define a cellular automaton as a 3-tuple

$$(d, Q, \delta)$$

where $d \in \mathbb{N}$ is the dimension, Q is a finite set of states, and $\delta : Q^{2d+1} \longrightarrow Q$ is the transition function, taking as arguments the current state of a cell and the states of the two immediate neighbors in each of the d directions and returning the new state. For our purposes, d will usually be 2 or 3, since 1-dimensional automata aren't as powerful as we want, but automata in 4 or more dimensions aren't scalable.

The *configuration* of a CA at time t can be represented by a function $f_t : \mathbb{Z}^d \longrightarrow Q$ assigning to each point on the d-dimensional grid the state of the cell at that point. Given the configuration at time t, the configuration at time $t+1$ is given by applying the transition function to the state of each cell and its immediate neighbors (at time t), as follows:

$$f_{t+1}(x_1, x_2, \ldots, x_d) = \delta(f_t(x_1, x_2, \ldots, x_d), n_1^-, n_1^+, n_2^-, n_2^+, \ldots, n_d^-, n_d^+)$$

where n_i^- and n_i^+ are the states of the two immediate neighbors along dimension i:

$$n_i^- = f_t(x_1, x_2, \ldots, x_{i-1}, x_i - 1, x_{i+1}, \ldots, x_d),$$

$$n_i^+ = f_t(x_1, x_2, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots, x_d),$$

There are other definitions, allowing more general finite neighborhoods (cells two away, diagonally adjacent, or any uniformly bounded neighborhood), but a CA as we have defined it can simulate the general kind easily. Another generalization (see [Gar95]) allows the index set ($\mathbb{Z}^n$ in our definition) to be instead the Cayley graph of any finitely-generated group.

It is also reasonable to define cellular automata on finite grids, since any real machine we build with them will have only a finite number of processors. Some research has been done on the computational complexity of decision problems involving finite CAs [Nor89]. For asymptotic analysis of algorithms, however, all the standard models are unbounded, and so we will use infinite cellular automata as our abstract model of parallel computation.

3.1.2 Universality

A computational system is universal if it can solve, in some finite time, any problem which can be solved in finite time by any Turing machine. Essentially, a universal system is one on which we can run any computer program (without ever running out of memory—these are abstract systems). The fact that various simple cellular automata are computationally universal has been demonstrated many times, and was even implicit in von Neumann's original paper on cellular automata (see the introduction of [Bur70]). Even extremely simple automata, like Conway's famous game of life, in which cells can take on only two states, have been proved to be computationally universal [BCG82].

This notion of universality makes no reference to efficiency. If one system can simulate another, universal, one, the first is also universal even if the simulation takes exponentially longer. We now want to show something stronger, though just as intuitively obvious: that a system can simulate other computers without being too inefficient. In the world of cellular automata, we want to show that there are simple CAs that can simulate any other CA of the same dimensionality with only a constant expansion in space and time.

Space-time universality

Each cell of a cellular automaton is a finite-state machine. It should come as no surprise that such a cell can be simulated by a block of simpler cells, with, naturally,

some slowdown. If we have a sufficiently powerful basic cell, we should be able to construct from it blocks which simulate any arbitrary cell of any CA of the same dimensionality. Any configuration of an automaton A that we're simulating can be mapped onto a configuration of the simpler automaton B in the obvious way, and after kt steps of the automaton B (where k is the slowdown factor), the configuration can be mapped back to give the configuration A would have reached after t steps.

This gives us a more direct version of universality for cellular automaton: a CA (d, Q, δ) is space-time universal for d-dimensional CAs if it can simulate any other d-dimensional CA in the way described above. This is more powerful than computation universality—if we can simulate arbitrary CAs of even one dimension then we can simulate Turing machines [AI87], but the converse is not necessarily true because a CA can have an infinite, not just unbounded, amount of state.

First, we define a *block map*, which is just a mapping from individual cells in one automaton to blocks of cells in a different automaton:

Definition 1 *Let $A = (d, Q, \delta)$ and $B = (d, R, \gamma)$ be d-dimensional CAs. An e^d block map from A to B is a function $\phi : Q \xrightarrow{1-1} (\mathbb{Z}_e^d \longrightarrow R)$.*

where $\mathbb{Z}_e$ is the set of integers modulo e. The block map ϕ is defined on states of the CA, but we can extend it in a natural way to configurations. Given a configuration f of A, let g be defined by

$$g(ej_1 + k_1, ej_2 + k_2, \ldots, ej_d + k_d) = \phi(j_1, j_2, \ldots, j_d)(k_1, k_2, \ldots, k_d)$$

where $j_i \in \mathbb{Z}$ and $k_i \in \mathbb{Z}_e$. Then we will say that $\phi(f) = g$. Now we define emulation of one CA by another, under a particular block map:

Definition 2 *Let $A = (d, Q, \delta)$ and $B = (d, R, \gamma)$ be d-dimensional CAs, and let ϕ be an e^d block map from A to B. Then A emulates B under ϕ if there is a positive integer s such that for any initial configuration f_0 of A, if $g_0 = \phi(f_0)$ then $g_s = \phi(f_1)$, following the normal rules of evolution for the two automata.*

This definition only specifies that B can simulate A for a single step, but a simple inductive proof will show that the simulation continues to work indefinitely. Note that the automaton must evolve to *exactly* the configuration it would be mapped into on the next step. A more forgiving definition of emulation would also be possible, but then the general induction wouldn't work.

Definition 3 *A d-dimensional cellular automaton U is* space-time universal *if for any other d-dimensional CA B there are positive integers e and s and an e^d block map ϕ such that U emulates B under ϕ.*

Surprisingly, this version of universality seems never to have been explored before, except quite recently in the 1-dimensional case. Martin [Mar94] showed the space-time universality (actually a slight variant which he called "intrinsic universality") of a 1-dimensional automaton. For completeness, we will prove the existence of space-time universal cellular automata in 2 or more dimensions.

Theorem 1 *For any $d \geq 1$, there is a space-time universal CA $U_d = (d, Q, \delta)$.*

Proof. Martin's construction, with slight modification, can be used for the case $d = 1$. We will sketch the construction for $d = 2$. This construction generalizes in an obvious way for higher dimensions.

For an automaton with q states, we will encode the state in unary as q individual signals coming in from each side of the square block representing the cell, and this cells state will be sent out on q signals to each neighbor. The overall size of the block will be $q^3 \times q^3 + O(q)$, with a central lookup table indexed by the 4 neighbor's states and our own, which will select a column of q cells. One of these q cells will be set to send a signal on its row, representing the new state. Figure 3.1(a) shows the block representing a cell for a 3-state automaton. The neighbor's state lines are routed to select one of q^4 $q \times q$ tables. One of the $q \times q$ tables is shown in figure 3.1(b). A signal representing the current state is sent in, and one representing the future state is output—in this case, states 1 and 3, carried by signals C1 and C3, becomes state 2, and state 2 becomes state 3. There is, of course, more wiring (not shown) to distribute this output state to each of the four sides.

We must add one additional complication: at the boundary of the cell the input signals must be gated, and only allowed to pass at the beginning of a global clock cycle. The clock can be implemented simply as a signal circulating in a wire, like a delay-line memory. The purpose of this is twofold: to keep the cells synchronized—the clock cycle is the time expansion factor—and to allow for a period when the gates are closed and the internal wires can be flushed of all traces of the input signals. The latter is to meet the definition of emulation above, in which the block must exactly match the mapping of the original automaton's next state.

This overall construction can be realized in a simple automaton with 64 states—32 for different kinds and states of wires, 8 for one-way wires (diodes), 16 for gates, 7 for a cell in the lookup table, and 1 for an insulator. Since the construction is straightforward but tedious, we omit it here. □

Simulating digital electronics

By analogy to the simulation of one cellular automaton by another, we can imagine simulating a digital circuit by a CA. Because of the finite speed of signal propagation