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.

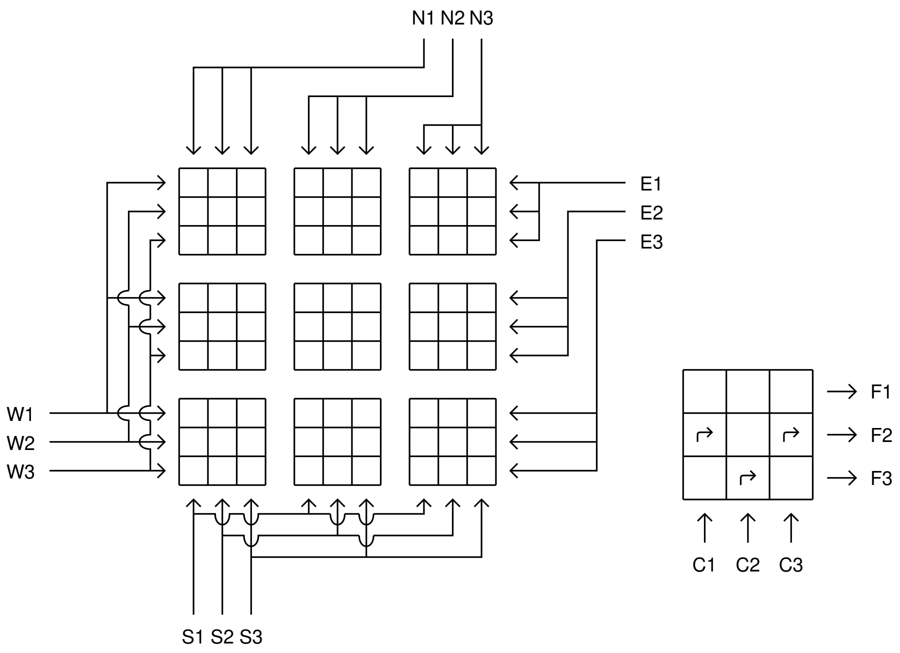
For an automaton with q states, we will encode the state in unary as q individual signals coming in from each side of the square block representing the cell, and this cells state will be sent out on q signals to each neighbor. The overall size of the block will be $q^3 \times q^3 + O(q)$, with a central 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. $\square$

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



(a) Overall block structure (b) Detail of one square

Figure 3.1: Simulating a 3-state 2-dimensional automaton.

along wires, we can actually simulate any such circuit with a 3-dimensional CA to within a constant factor in speed and size.

The components we need to simulate in a digital circuit are wires and gates. Wires must be able to be driven to a zero or a one (represented by the voltage of ground and the power supply, respectively), or they can be unconnected, not driven to either voltage, in which case they hold the last value they were driven to (capacitance). The voltage may gradually drain if the wire is unconnected for a long time, but we do not need to model this, since it is an analog effect, and does not affect correctly-functioning digital circuits. The other component, a gate, acts as a switch—if a particular signal is present on the control line (either a zero or a one, depending on whether this is a PMOS or NMOS transistor), then two other wires are connected, otherwise they are disconnected. These components can easily be realized with cellular automata, and can probably be realized by any universal CA. See [BCG82] for a realization of wires and gates in Conway's game of life.

Simulating physics

It is reasonably clear that a CA can simulate, in linear time, anything built from conventional components. But can't we be even more general, and say that a CA can simulate any physical device to within a constant factor? Surely in any device we build, the components will have some minimum size, which can be the quantum of space we simulate with a single cell. And communication will propagate along channels no faster than the speed of light, which can determine our quantum of time. Even if the computer were analogue, or opto-electronic, the same limitations would apply. But continuous processes can be much more difficult to simulate well—the digital simulation we referred to above is comparable to IRSIM, a speedy event-driven circuit simulator which generally works well. To simulate the analogue behavior of circuits, however, requires a tool like SPICE, which is orders of magnitude slower, and sometimes is unable to complete a simulation. And SPICE only simulates the components found in standard integrated circuits, not all of physics.

Can physics, which is continuous, be accurately simulated by a discrete system like a cellular automaton? Calculus often gives a good description of the macroscopic behavior of systems that are actually discrete at the microscopic level. Some researchers (notably Fredkin [Fre90]) even believe that the universe may *be* a cellular automaton. Very many quantities we once thought continuous have proven to be discrete, from atoms to energy quanta, and it is plausible that space and time may be also. Certainly CAs have had a great deal of success in modeling continuous physical processes, including the wave equation [CKCN93]. Unfortunately, quantum mechanics is not determined entirely by the evolution of the wave equation. We don't know precisely how to simulate wave function collapse, or precisely when it happens, but when it does it can result in non-local effects, which violate the nearest-neighbor communication of a CA model.

All this is speculation—what we can say with certainty is that so far, we do not know enough to be able to simulate all of physics by a cellular-automaton model. It may be possible, but there is already some evidence to the contrary (see section 3.3.2).

3.1.3 Related work

Others have argued the case that the speed of light must be brought into our models of computation. One of the pioneering works in the area was Chazelle and Monier's paper [CM85] introducing the idea of VLSI complexity. Their goal was to lay "the foundation of a general theory of physical computability."

Feldman and Shapiro [FS92] invented an abstract model of computation very similar to our 3-d CAs, with an interesting twist. Their parallel computers, called *spatial machines*, are 3-dimensional cellular automata in which all but a finite number of cells are in special states which only relay bits in each of the 6 canonical directions. Some finite number of cells contain active components, or processors, which are more general finite automata, and can move to any neighboring cell in any time-step. Essentially, they wanted to model computers which had a fixed amount of hardware, but arbitrarily much empty space. The empty space, as they point out, can hold information—in this way the machine can model a Turing machine's infinite tape even though it has only finitely many processors.

More recently, Bilardi and Preparata [BP95] argued that the 2-dimensional mesh is the only scalable computational model. They assumed planarity because of the predominance of planar layout in current computing systems, and because of the roots of their work in VLSI complexity. The mesh they investigate is nearly the same as our cellular automaton model, except that the amount of state information (as bits) in each cell is allowed to grow proportionally to $\log(p)$ where p is the number of processors. In this way, each "word" can address any of the processors. We have found that global addressing is unnecessary in most algorithms, but in any event, the log factor results in only a minor difference in performance between the two models.

Although they argue for mesh interconnect at the highest level, Preparata and Bilardi also point out that in a small enough domain the speed-of-light time between cells is likely to be negligible compared to the cycle time. Within such a domain, called a *synchronous domain*, communication time can be considered a constant independent of distance. The wires used to route such data still take up considerable space, however. They suggest using a fat-tree, which is provably nearly as efficient, for any collection of messages, as any other network using the same layout area. These synchronous domains are themselves connected only to their neighbors, so at the highest level, the computer is still a mesh, albeit with much larger cells which are, in themselves, parallel computers.

3.2 Relation to other models

We have argued that a 3-dimensional cellular automaton can simulate any physical computer to within a constant factor. But how well can it simulate theoretical computers in those models we have argued are not scalable? The other theoretical models are in widespread use in the computer science community, and providing some relationship between CAs and those models will immediately make a great body of work on parallel algorithms applicable to cellular automata.

3.2.1 RAM

The first model we consider is not a parallel model at all, but the standard model of sequential computation, the RAM (Random-access machine) model, introduced by Cook and Reckhow [CR73]. It is simply an idealization of the von Neumann architecture, with an infinite memory in which each cell can contain an arbitrary integer and simple arithmetic operations or memory references take unit time.

One would expect that a CA would, like most other parallel models, simulate the RAM in linear time—we can't expect to improve on straight simulation for some inherently sequential programs, but we also don't expect the parallel machine to take any *longer* than a sequential one. But the CA can, in fact, take longer than a RAM for the same problem. One reason is the unit-cost assumption for arbitrarily large integers, which is a known unrealistic feature of the RAM model. Modular exponentiation of large integers, for example, is used in cryptographic systems like RSA. It could be performed in $O(n)$ time in the pure RAM model, but actually requires $O(n^3)$ operations on real computers. Therefore we will limit the RAM model to words of a fixed size.

The other, more serious reason that a CA will be slower than a RAM is the fact that an arbitrarily large memory cannot be accessed in constant time. Suppose the memory size is m. On a d-dimensional CA, the memory accesses will require $O(\sqrt[d]{m})$ time in the worst case. Is this reasonable? I claim that it is, because the RAM model itself is not scalable. Assuming a constant volume for each bit of memory, in a memory of m bits, some accesses will have to travel a distance of $\Omega(\sqrt[3]{n})$. Because of the speed of light, the time required for the access will also be of the same order.

This seems to fly in the face of 50 years of experience with the von Neumann architecture. In this case, however, recent history actually shows evidence of the RAM model's inadequacy. Most current computers now have highly hierarchical memories, consisting of registers, on-chip (L1) cache, off-chip (L2) Static RAM cache, Dynamic RAM arrays, and finally paging storage on disk, with a significant performance degradation at each step. The fact that the caches work well and that most programs have fairly good locality of reference gives the *illusion* of uniform access time, but the illusion breaks down for programs with poor locality. If we use a CA for *practical* simulation of RAM programs, we will want exploit the same locality. Moving the frequently-accessed data closer to the working registers will usually reduce average memory latency just as it does in the hierarchical memory systems actually in use today.

3.2.2 PRAM

The Parallel RAM model (PRAM) was introduced by Fortune and Wyllie [FW78]. It is similar to the sequential RAM model with the addition of a shared memory by which the processors can communicate with each other. The shared memory can be accessed with unit cost, which makes communication extremely cheap. Several variations are in use today [Rei93], allowing concurrent read (CR), exclusive read

(ER), concurrent write (CW), exclusive write (EW) or owner write (OW). These variations are all reasonably similar, in that they can all simulate the others within a $\log(p)$ factor (where p is the number of processors).

As with the RAM, we will assume that each word in the memory is of constant size. We also assume that we have a PRAM program using p processors and m words of memory. We can simulate this with a CA having $p + 2m$ cells. We simulate a single step as follows: collect all the memory references from the processors and sort them. If there are duplicates, choose only one representative from each group. Route the representatives to their destination memory cells. If the memory references are writes, that is the end of the process. If they are reads, the value in the memory cell must be routed back to the representative in the sorted array. There is then a phase in which the value is propagated to all the requests that referred to the same address. Finally, these values are routed back to the requesting processors. Note that this correctly simulates the most powerful model of PRAM, the CRCW.

The time required to simulate the single step is $O(\sqrt[d]{p+m})$ on a d-dimensional CA. Sorting the m references can be done in $O(\sqrt[d]{p+m})$ time, as can routing (see section 5.3). The distribution of the value requires passing it along to a set of now-contiguous cells, so it also requires $O(\sqrt[d]{m})$ steps, and routing the results back to the processors adds another $O(\sqrt[d]{p+m})$ steps.

A t step PRAM program can therefore be simulated in $t\sqrt[d]{p+m}$ steps on a d-dimensional CA. For many well-known parallel algorithms, this is close to the best we can do—any NC algorithm which uses $O(n)$ processors and memory and depends on all its input will require at least $\Omega(\sqrt[d]{n})$ time on the CA, whereas the simulation would take only a polylog factor more: $O(\log^k(n)\sqrt[d]{n})$. In our experience, the $\log^k(n)$ factor can usually be eliminated.

3.2.3 BSP

The PRAM is a theoretical model, and is, by now, known to be unrealistic within the parallel computing community. We know that a cellular automaton can run programs written for this idealized machine reasonably well. But what about programs written for a real machine, like a Hypercube or a CM-5? There are, unfortunately, as many performance models as there are types of parallel computers, and it would be infeasible to analyze all of them. The differences lie in the performance of the communications system as a function of message length, other traffic, location of the processors communicating, size of message, and so forth. A common characteristic for most of these machines is that the communication system has much higher bandwidth for large messages than small ones. This is the defining characteristic of coarse-grained parallelism.

Fortunately, there are some models which attempt to bridge the idealized models and the real machines. The LogP model, from Berkeley [CKP+93], and Valiant's BSP (Bulk Synchronous Parallel) model [Val90, McC95] are both ways to realistically account for communication costs in parallel algorithms, and both succeed rather well. There is a growing body of parallel software written for the BSP model, which runs efficiently on a wide range of machines, each of which require only the library of BSP primitives to be ported. The two models are similar; they have been shown to be nearly equivalent [BHP+96]. I will consider only Valiant's, which is slightly simpler.

BSP characterizes the parallel system by three parameters: p, the number of processors, l, the time required for a "barrier synchronization" of all the processors (no processor proceeds until all processors have reached the barrier), and g, which is approximately the inverse of the available communication bandwidth per processor. In BSP, a computation consists of a series of parallel *supersteps*. In each superstep every processor does some sequence of operations on local data, followed by a barrier synchronization, at which point remote communication occurs. The time required by a superstep is

$$T_s = w + gh + l$$

where w is the maximum time used in local computation by any processor and h is the maximum number of words sent or received by any processor.

BSP can easily be applied to the d-dimensional CA: the parameter l is just the diameter of the network, or $\sqrt[d]{p}$, and g is also approximately $\sqrt[d]{p}$, since routing h words of data for each processor (the classic h-h routing problem [Sib95]) requires $h\sqrt[d]{p}$ time. This is not quite right, however, since we also have to add a factor for the maximum memory size used by any processor. BSP, following the usual practice of the RAM model, assumes that unlimited amounts of memory may be accessed with no slowdown. If we assume that m is the maximum memory size, BSP can be simulated on a CA in time $T = \sqrt[d]{m}(w + gh + l)$.

3.3 The limits of theory

Now we must qualify, somewhat, the claim that cellular automata are as efficient as any physically realizable computer. There are two problematic areas which we will examine in this section.

The first is whether a 3-dimensional array is in fact physically realizable and scalable. Integrated circuits and printed circuit boards are all two-dimensional now, and to depart from this would be a major engineering problem. But there may be a more fundamental reason for the two-dimensionality: heat dissipation. Heat may be removed directly from the surface of a planar computer, but only from the perimeter of a solid. It might be impossible to adequately cool the interior cells, even from a theoretical standpoint.

The other qualification concerns the ability of a 3-dimensional cellular automaton to actually simulate all other physically-realizable computers, even using novel circuitry. In general, physical laws are local in nature, and a cellular automaton should in principle be able to simulate them in some constant fraction of real-time. However, certain effects in quantum mechanics require non-local communication, and there seems to be no way to simulate this with a cellular automaton. Conceivably, computers built to exploit these effects would be strictly more powerful than cellular automata. There seems to be some evidence that quantum computers can theoretically speed up certain tasks by more than a polynomial factor if we can meet the formidable challenges that stand in the way of building them.

3.3.1 Heat

Computers, of course, use energy to perform their computations. The necessity of this has been clear since Maxwell's demons were successfully explained. The energy used must be dissipated as heat, and the successful engineering of many of the world's fastest computers, including all of the Cray supercomputers, involved novel cooling systems. Nevertheless, the amount of energy dissipated by a gate switching has decreased exponentially since the introduction of integrated circuits, even as the switching speed has increased. It is of both theoretical and practical interest to determine if there is a lower bound on the energy dissipated. The well-known answer, apparently first observed by von Neumann [vN66] is that there is a limit. A logic element must use representations of one and zero which are distinct states. The presence of thermal noise means that the two must be separated by sufficient energy that the noise can't change a one to a zero spontaneously—or at least that the probability of this is low. That requires that the two states be separated by an an energy difference proportional to the absolute temperature, specifically $E >> kT$ where T is the absolute temperature and k is Boltzmann's constant, $1.3807 \times 10^{-23} J/K$. When an AND gate switches, for example, if the output value stays the same it may not consume any energy, but if it changes, the previous value is thrown away—in CMOS it is shorted to ground or power—and the energy is released as heat. This switching energy must be at least kT.

2 or 3 dimensions?

The problem of heat is not unique to cellular automata—it is shared by all parallel or classical architectures as they continue to miniaturize. At the current rate of progress in reducing feature size and switching energy, we will reach the limit of

kT (at room temperature) in about 18 years [Mer93]. But there is no fundamental problem with generating heat—we simply need to provide sufficient heat transfer to cool the array. The problem for cellular automata in 3 dimensions is that the maximum heat transfer across any surface (a sphere of radius r, say) must scale proportionally to the area of the surface, which grows as r^2, while the number of cells in the interior, in a 3-dimensional lattice, will grow proportionally to r^3. The heat generated will be proportional to the volume, and so will also grow as r^3. In a sufficiently large array, the heat dissipation will grow beyond our ability to cool it.

One possible solution to the problem is that 3-d arrays will be feasible to build, up to some size limit, and that most useful computers will be below that size limit. The size limit may apply only to a single dimension, so that "slabs" of computational substance may be possible up to some maximum thickness. Naturally, such a slab can be simulated on a 2-dimensional array with only a constant slowdown, but the third dimension may be deep enough that we will still want to study computational complexity on 3-d arrays.

Another possibility is to use 2-dimensional cellular automata. Then heat generation increases proportionally with surface area, and we should be able to cool the array by conventional means. Unfortunately, the argument for space-time universality is no longer as persuasive. Any conventional device can be simulated on a 3-d array with no more than a linear slowdown, but not necessarily on a 2-d array.

How much slower is a 2-d array than a 3-d array? We can map a cube r units on a side onto a square $r^{1.5}$ units on a side in such a way that adjacent cells in the 3-d array are separated by no more than $\sqrt[2]{r}$ cells in the 2-d array. Then every step of the 3-d array can be simulated in $\sqrt[2]{r}$ steps on the plane, so the slowdown factor can be no more than $\sqrt[2]{r}$. Assuming that all the processors actually play a role in computing the answer, the time on the 3-d array (T_3) must be at least r, so $T_2 \leq T_3^{1.5}$. This means the 2-d array is not too far from the optimum. In the rest of this thesis, we treat the dimension d of the array as a parameter, either 2 or 3, and give the running time of algorithms in terms of d. Typically, the difference is just that the 2-d algorithm has a factor of $\sqrt[2]{n}$ where the 3-d array has $\sqrt[3]{n}$.

The preceding analysis is for simulating a full 3-d array. If we are simulating a real scalable physical device, that device must satisfy the same limits on heat dissipation as the cellular automaton. Therefore, we can presume that no more than $O(r^2)$ of the space in the device is used by gates or wires in *any* cube with side r within the device. The general 3-d to 2-d embedding above could be used, inflating a volume of radius r into a planar region of radius $r^{1.5}$. It seems likely that a better bound than $r^{1.5}$ can be achieved when we know the density of the device is low. We conjecture, but have not been able to prove, that any structure with no more than r^2 density in any region of diameter r can be mapped onto the plane with only a constant expansion. More formally:

Conjecture 1 *Let A be a set of points in $\mathbb{Z}^3$ such that for any cubical region $S = \{a+1, a+2, \ldots a+r\} \times \{b+1, b+2, \ldots b+r\} \times \{c+1, c+2, \ldots c+r\}$, $|A \cap S| \leq r^2$. Then there is a mapping of $f : A \longrightarrow \mathbb{Z}^2$ such that if $x, y \in A$ and $\|x - y\| = 1$ then $\|f(x) - f(y)\| < c$ for some constant c not depending on A.*

In that case, the 2-dimensional array would once again be as fast, within a constant factor, as any device we could build.

Reversible computation

Despite the foregoing, arbitrarily large 3-dimensional arrays may be possible. Rolf Landauer first pointed out that energy dissipation is only theoretically necessary when a gate performs some irreversible operation [Lan61]. Fredkin proposed a kind of gate, now commonly referred to as a *Fredkin gate*, which can reversibly perform the same functions as the standard irreversible NAND or NOR gates. Since it cannot destroy information, it has 3 inputs and 3 outputs—a control signal which passes through unchanged, and 2 other inputs which are conditionally swapped depending on the value of the control line. Several techniques for building computers out of reversible gates have been proposed and investigated, including reversible cellular automata [Mar91, Tof77].

Reversible logic is not just a theoretical possibility—it can be demonstrated with current technology. Circuits have been built which dissipate much less heat than conventional CMOS, though still well above the theoretical limit of kT. In fact the technique was independently invented in the electrical engineering field in the 1970s. At that time the dominant technology was NMOS, which dissipates much more power than CMOS. Reversible circuits, called "hot-clock NMOS," were designed to reduce that power consumption to levels more comparable with CMOS. More recently, researchers at MIT have used the same techniques to greatly reduce power consumption in CMOS circuits [YK94].

In any case, reversible computation is far from being understood from the point of view of algorithms and complexity. Reversible instruction sets have been designed [Hal94], but programming a reversible computer introduces a large number of constraints to ensure that information is never destroyed. Irreversible operations may be introduced to provide for necessary forgetting of state. The forgetting would presumably be done only at the periphery of the array, where the heat can be dissipated. Fredkin [FT82] showed that any computation can be done with only enough irreversible steps to copy over the input and output, by running the program forward (keeping *all* intermediate results), saving a copy of the output, then running it backward to consume all the intermediate results. Unfortunately, this may add greatly to the space and time required by the program. It is likely that the most efficient algorithms in such a model will be quite different from those we now use.

All these considerations leave us in an uncertain realm between 2 and 3 dimensions. Fortunately, the difference is not too great, either in the design of algorithms or in their efficiency. If we design algorithms capable of working in both models, we cannot go far wrong.

3.3.2 Quantum computing

In contrast to the problem of heat, which implies that we may not be able to actually build a machine as powerful as a 3-dimensional cellular automaton, some of the features of quantum mechanics suggest that we might be able to build a computer *more* powerful. The cellular automaton model is based on the fact that all interaction is local, while quantum mechanics seems to have certain unavoidable non-local effects. The classic demonstration of this is the Einstein-Podolsky-Rosen (EPR) paradox, in which a pair of particles is created having correlated, but as-yet uncertain spins. When one particle is measured, the other's spin is determined, if it is measured by a spin axis 90 degrees rotated from the first. If the two particles are widely separated and the two measurements are nearly simultaneous, this apparently requires information to be communicated faster than the speed of light, which under special relativity implies that causality is violated. Einstein, in fact, designed this experiment to show that there must be a flaw in quantum mechanics, but the paradoxical effects have withstood the test of time, and Bell's theorem [GHSZ90] has shown that the faster-than-light communication is unavoidable. This seems to make it impossible to simulate quantum mechanics on a CA with only constant slowdown.

The question then becomes whether quantum effects can be used to any advantage in computing. Fortunately, there seems to be no way to use the non-local effects to actually transmit information faster than the speed of light—if there were, it could be exploited to send information into the past, and create macroscopic violations of causality and accompanying paradoxes of self-reference. There is mounting evidence, however, that quantum mechanics can provide more efficient solutions to certain problems than any computer built out of conventional logic.

Shor [Sho94] has proved, for example, that factoring and discrete logarithm can be done in polynomial time in the commonly accepted model of quantum computing. At the moment it appears very unlikely that either problem can be solved in polynomial time on a standard Turing machine. It would be even more convincing if an *NP*-hard problem could be solved in polynomial time on a quantum computer, but this has not been done. Grover [Gro96] made some progress in this area by showing that any search problem can be solved on a quantum computer with a $\sqrt[2]{n}$ speedup over a sequential algorithm. This is no faster than can be achieved by a 2-dimensional CA, but the computation is performed on a single quantum computer, leaving open the possibility of even faster algorithms on a parallel quantum computer.

Quantum difficulties

We have been arguing that it is the right time in history to adopt cellular automata as our standard computation model. Could it be instead time to leapfrog ahead to quantum computation? Probably no researcher in the field would say so now, since no quantum computers have yet been built. The possibility of building one is much more in the realm of science fiction than a cellular automaton was when von Neumann first proposed the model. There are formidable practical and theoretical difficulties to overcome. In standard digital circuits, the signal is regenerated and cleaned of noise after every operation. In a quantum computer, there is no apparent way to do this—making a "clean" version essentially means making a measurement and thereby collapsing the wave function. Designing a memory system for a quantum computer is also unsolved.

It is still very much an open question whether these problems can be overcome. If they are not, then our arguments for the optimality of cellular automata will still be valid. If quantum computation does become a reality, however, there is some chance that a combination of quantum computation and cellular automata may be more powerful than either technique alone. Indeed, some of the work on design of quantum computers has focussed on cellular automata because of their simplicity and regularity [BJ97, Bia94].

Chapter 4

Cell architecture

> The code which he suggests is however very contrary to the line of development here, and much more in the American tradition of solving one's difficulties by means of much equipment rather than thought.
>
> — Alan Turing, on Wilkes' proposal for a pilot architecture for the ACE computer

Theory tells us that the particular cells used in the array don't really matter, as long as they are space-time universal. Any other cell design can be emulated within a constant factor of the same speed. But the long-term goal is to start using cellular arrays in standard computers. To do so requires that we pay attention to practical matters. Constant factors can be quite important in the short-term viability of a machine architecture, as we have seen—many supercomputers have been overtaken by Moore's law before they really could show their potential.

The overarching goal, encompassing both architecture and software, is to produce a parallel machine that can be competitive with standard von Neumann machines in all areas and clearly superior in many. That goal is almost inevitable in the long run as features shrink to nanometer sizes. At that level, even such simple cellular automata as Conway's game of life will be usable. But with the technology available in the next few years, we will have to balance the efficiency of such well-developed structures as ALUs and memories with the benefits of simplicity.

The major issues for design of the cells include:

- How wide should the data paths be between cells? Thinking of the cells as programmable logic gates suggests one or two bits. Thinking of them as small computers suggests 16 to 64 bits. Which model is best?

- How much state (i.e. memory) should a cell have? Memory arrays are the most area-efficient use of silicon in current technology. It would be wasteful not to take advantage of that. However too much memory is what's responsible for the von Neumann bottleneck. Also, the amount of memory per cell divided by the I/O bandwidth of a cell determine the time it takes to move data on and off a cell, which is a critical factor in many algorithms.

- Should they be synchronous (clocked) or asynchronous? The cellular automaton model is synchronous, but in real implementations, the clock can become a global signal, and hence a long communication wire, a feature we are trying to avoid.

- Should they be 2- or 3-dimensional? Current VLSI and circuit-board technology is planar. If we ignore that, we may have a system which cannot be manufactured. On the other hand, if we design now for planar technology, we may find our algorithms hobbled when the third dimension is added.

- Should there be non-local connectivity? Even at a clock speed of 600MHz, like the fastest commercially available processor, a signal can propagate about one foot in a clock cycle. The propagation speed in a cellular array clocked at that speed would be much less, assuming the cells are much smaller than one foot apart. Perhaps a compromise would be useful, providing some channels which span several cells.

- How can cells be packaged? Presumably many cells will fit on a single chip, but the number of I/O pins at the edge of the block will become quite large. If the cells are to be connected in a 3-dimensional grid, the problem becomes significantly worse.

- How can heat be dissipated? This is particularly a problem for 3-dimensional arrays, but even 2-d arrays can have a much higher concentration of active logic than current computers, and heat dissipation already requires special techniques on processors like the Pentium.

- How can cells be initialized (programmed)? There must be some way to "bootload" a program or operating system into the array. And if we want to have a robust operating system running on the array, the OS must have a way to arrest and reload a section of the array containing a program which has stopped responding. Ideally this process would not require a global reset or long wires—it should be local, like other communication.

In answering these questions, we have produced two prototype designs. The first, somewhat idealistic, single-bit design was realized as a project chip. Although the cell proved to be too simple for the goal of general-purpose computing, it provided some valuable lessons, which were incorporated into the second prototype. That design, with much larger cells, has served as a platform for implementing software. We have built a simulator for it, but have not yet implemented it in hardware.

4.1 The first prototype

The most radical answer to the question of memory is that all memory should be active. The von Neumann bottleneck is the (low) ratio of active logic to passive logic in the system. Memory is, by its nature, the chief culprit—only a single item from the memory is being fetched or stored at a time while the rest of the array is idle. So to definitively eliminate the bottleneck, we can make every bit of memory capable of doing useful computation.

This is the answer used by our first prototype. The grid is 2-dimensional, with each cell connected to its neighbors in the four canonical directions. The data paths are one bit wide between neighbors, but are bidirectional. The cells can be programmed to perform only a small number of logical operations, but the set is universal in the sense that any circuit can be embedded in the array.

The complexity of an individual cell is not much different from the game of life. If molecular-scale machines, "computing crystals," can ever be realized, then they will likely have such a simple structure.

4.1.1 Functional description

Each cell can be programmed to perform one of the four functions shown in figure 4.1. The NAND function (N) is sufficient by itself to implement any circuit. In addition to its logical applications, a NAND with a constant 1 can be used for routing signals, to turn a corner while inverting. The XNOR function (X) is difficult to construct out of NAND cells, and it makes construction of adder-like circuits simpler. Like the NAND, it can be used to make a signal turn a corner; unlike the NAND, it doesn't invert the signal in the process. The One function (1) presents a constant 1 on all of its outputs. The Pass function (P) connects the input and output lines of the cell above to the cell below, and similarly for the left and right neighbors. This allows construction of "wires" to relay signals, and also gives signals the ability to cross over one another when the circuit cannot be realized by a planar graph.

The operation of the cells is *asynchronous*—there is no global clock to synchronize operation of adjacent cells. Instead, changes of the input signals result in changes in the output after a small delay, depending only on the speed of the inter-

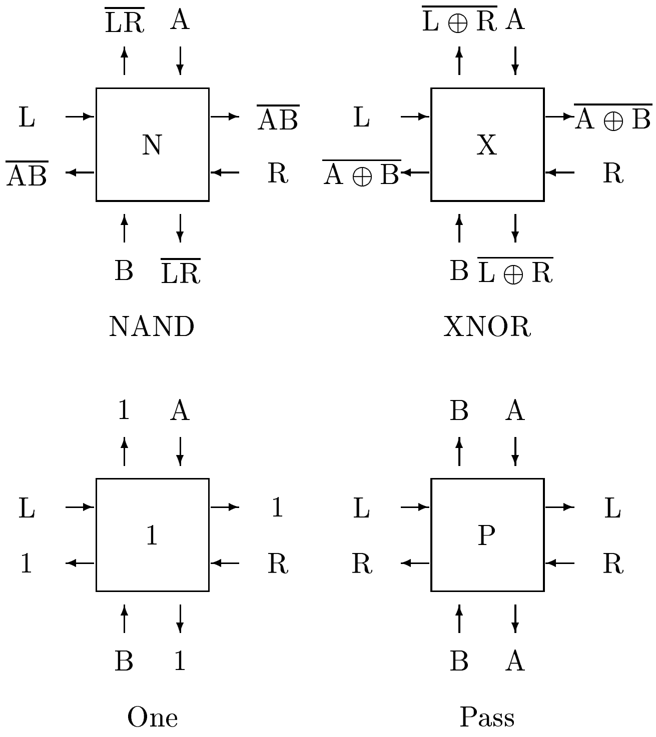


Figure 4.1: The four cell functions

nal logic. This is in contrast to the cellular automaton paradigm, in which signals change only on a global clock tick. We wanted to see if reasonable systems could be built without the need for a global clock, which in itself may be seen as a form of non-local communication. Also, we wanted to program at the level of logic circuits, which aren't inherently synchronous.

Programming of the cells, unfortunately, is a non-local operation. Each cell maintains a 2-bit latch holding the function the cell performs. These latches may be changed for an entire row of the array at once.

Although the cells incorporate latches (for the function memory), when they are programmed they function only as memoryless logic gates, and it takes several of them operating together to provide the feedback required for memory. Figure 4.2 shows how to program a single-bit flip-flop.

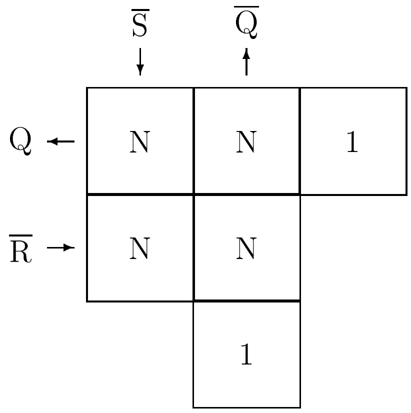


Figure 4.2: An SR flip-flop

4.1.2 Implementation

This cell design was implemented in 1991 as a CMOS project chip fabricated by MOSIS. MOSIS provides a service for VLSI prototypes in which several designs are packed onto a single chip, then sent to a foundry as a unit to save cost. This leaves each design with only a relatively small area, which for us meant a small array. Our chip had a die size of just under 2mm on a side, and we packed an 8×12 array of cells into this space, using 2 micron technology (i.e. the minimum feature size is 2 microns). Each cell required 92 transistors, in an area of 286×410 microns. The gate delays for the 4 available gates ranged from 2 to 5 nanoseconds, which is only worse than the bare logic gates by a factor of 2 to 3 for this technology.

4.1.3 Lessons

The primary lesson from this early design is that overly simple cell designs result in inefficient use of silicon. A single cell is comparable to a single logic gate in a hard-wired circuit, which (in CMOS) requires only 4 transistors. But the cell required 92 transistors to implement, an expansion factor of 20. But the expansion factor is worse than that: to provide a flip-flop requires 4 NAND cells in a square, together with two cells set to output a logic 1. The logic-1 cells could perhaps be shared with other flip-flops, but the bare minimum for providing a single bit of memory is 4 cells, and 368 transistors. In a dynamic RAM, only a single transistor is required for one bit of memory; in a static RAM only 5 or 6. Using today's technology, too much efficiency is thrown away. Memory in some form is going to be present in every circuit, every algorithm, we implement on an array. There must be a better balance between the relative efficiency of memory and computation.

The simplicity of the cells also precluded any attempt at bringing programming into the cellular automaton paradigm. Indeed, it is reasonably hard to think of *any* way to make programming a nearest-neighbor operation safely; with such minimal cells nearest-neighbor programming would have dominated the complexity of the cells.

Despite the inefficient use of silicon, the design revealed quite conclusively the benefits of using highly regular structures. These benefits are, of course, well-known for memory arrays, but not for active logic. This is evident in comparing the density of transistors for this array with state-of-the-art microprocessors like the DEC Alpha or Intel's P6 (Pentium II). Because of the differences in fabrication technology (the Alpha was implemented originally in .75-micron CMOS, the P6 in .35-micron, and our array in 2-micron), we will express all areas in units of f^2 where f is the minimum feature size. In this measure, the Alpha chip used $247f^2$ per transistor, the P6 used $220f^2$, and our chip used only $88f^2$. This does not show our superiority in VLSI design—very much the reverse is true—but it does show the great advantage of the cellular array structure.

4.2 Design requirements

The current cell design is based on the lessons learned from the single-bit prototype. It is intended to make much more efficient use of current technology. The cells have full ALUs and a small memory and the data paths are wider. The details of the design are the result of a number of (not always compatible) design goals:

- Synchronous operation. When doing operations at the level of arithmetic rather than logic, a shared clock greatly simplifies the communication between neighbors. This is, of course, a feature shared by the cellular automaton abstraction, as well as by systolic arrays, but it apparently requires a form of global communication for the clock signal. This proves less problematic than it first appears, however, since the clock signal only needs to be approximately synchronized between neighbors, not perfectly synchronized over the entire array.

- Strictly nearest-neighbor communication, including programming. Shutting down the entire operation of an array to change the program in one part is not practical if the array is to be used as a general-purpose computer. Global signals could become a significant bottleneck in initializing large arrays. But it is not immediately apparent how to allow an array to be "bootloaded" using only local signals.

- Identical cells. This is a rather obvious requirement, but it has the subtle consequence that there is no built-in addressing. For those who have seen architectures fade away because they had insufficient address bits, this may seem welcome, but it also raises the question of how the array can be navigated. In this case, we can easily follow our philosophy of replacing hardware with software by programming cells to count off in order from a designated corner. A program thus devises its own address space, which can be expanded without hardware modification when needed.

- Fast propagation. Since non-local communication needs to go through all the intermediate cells, we want that communication to propagate as fast as possible. In a synchronous system, the maximum speed (the "speed of light" of the system) is one cell per clock cycle. The design should be able to attain this speed—i.e. in a single clock cycle, a new item appearing at an input register should be able to be copied to an output register. This is not generally possible in machines with deep pipelines, since the data only moves through a single pipeline stage in one cycle. Pipelining is a form of hidden parallelism in conventional computer designs, and in this design we are trying to make the parallelism explicit and visible to the programmer. Each cell should be able to implement a pipeline *stage*, so the programmer can devise new pipelines when the algorithm requires, as in systolic arrays. To that end, we would like the 1 cell/cycle propagation speed to apply to arithmetic and logical operations as well: a cell should be able to add together the inputs from two of its neighbors and pass the result to a third, all in a single clock cycle.

- Fast clock speed. Obviously the previous goal can be met by slowing down the clock sufficiently, but we want the design to be simple enough that the propagation speed can be maintained even with state-of-the-art clock speeds, currently 200-300MHz. This requires that the logic depth (the depth of the tree of logic gates) of the input-ALU-output data path be comparable to the depth of a single pipeline stage in a processor like the Intel P6.

- Balanced memory and logic. Too much memory for each cell reintroduces the von Neumann bottleneck, as well as effectively slowing down the network bandwidth when measured per word of memory. This affects the efficiency of such basic building blocks as sorting (see section 5.3). But too little memory per cell means that the array will be much more inefficient than RAM in holding a given size of problem, so that much smaller problems will fit. This could more than outweigh the performance gain of parallelism. A reasonable compromise is to allocate about 50% of the area to memory and 50% to logic.

In this way, we are guaranteed not to lose more than a factor of 2 in efficiency over a pure memory or pure active-logic approach.

- Reduced logic. Although we have abandoned the simplicity of the single-bit design, we still want to keep each cell as simple as possible. In the spirit of RISC, we want each cell to be a "reduced-logic computer," capable of fast operation, but taking as little real estate as possible. The smaller the cell, the more of them we can fit in an array, so the more power there is. Similarly, if we use approximately equal area for the processor and memory, then the simpler we can make the processor, the less memory per node there needs to be, so the less the von Neumann bottleneck affects us. The reason we call this reduced logic rather than RISC is that RISC processors typically use quite deep pipelines and very complex logic to achieve high-speed operation. We want to use something more like microcode as the instruction set, a very low-level language that's easy to decode and fast to execute.

4.3 Functional cell design

Working with those design goals constrained the architecture of the cell in unusual ways. What emerged from trying to satisfy those goals concurrently looks very different from current RISC or CISC machines, but remarkably similar to the microcode architecture of machines built in the 1970s. The reason is the same—at that time, logic was expensive, and the use of microcode permitted a rich instruction set with a minimum of decoding and sequencing logic. Two kinds of microcode were common: vertical, which is highly encoded, uses short instruction words, and is very sequential; and horizontal, which uses very wide instructions and is highly parallel. We use an instruction set reminiscent of horizontal microcode, with 64-bit instructions.

The other main features of the cell are 32-bit data paths, a simple ALU, a 1-kilobyte static RAM (which provides a good balance to the size of the other logic), and a full crossbar switch between the data pathways (which allows us to meet the requirement of fast propagation). We will describe these features more fully below.

The design is specified for a 3-dimensional array, though the 2-dimensional subset is all that we could plausibly implement with current technology. We wanted to gain experience with the 3-dimensional array and its algorithms just because that is the harder problem. Specialization of algorithms to 2 dimensions is trivial, while generalizing from 2 dimensions to 3 is not, and the same is true of the instruction set. Also, having 2 neighbors in each of the 3 dimensions gives a total of 6 inputs and 6 outputs, which puts much more pressure on the instruction encoding than 4 of each (though there is a way to make 3-dimensional arrays using only 4 neighbors—see section 4.4.2).

The cell, as specified, is extremely simple to implement, and so meets the goal of being a "reduced-logic computer." One measure of this is the fact that the core routine of the simulator, simulating a single step on a single cell of the array, takes only about 200 lines of C code.

4.3.1 Data paths

In order to provide completely flexible and fast interconnections between the various I/O paths and the functional units of the processor, the heart of the cell is a crossbar switch. This switch connects the inputs from the 6 canonical directions, the ALU output and the memory bus with the 6 outputs, the memory bus, the 2 ALU inputs and a special "branch register" (BR). The switch does not limit the possible parallelism of communications, which has proved to be a very useful feature. The ability to simultaneously forward a value from east to west and one from north to south allows packets to cross with no loss of bandwidth, for example, and can also help speed up computation, for example in matrix algorithms. A block diagram of the cell is shown in figure 4.3.

Most of the data paths are 33 bits wide instead of 32. The extra bit allows for out-of-band signalling. In the output of the ALU the extra bit is the carry bit, but in the 6 directional I/O channels it is simply propagated from the source of the value. The memory only provides 32 bits, so on memory reads the extra bit is always zero. This extra bit can be tested on any of the 6 directional input channels, which provides a good way to notify a neighbor of something out of the ordinary—the end of a sequence of values, a request for attention from the operating system, or a count running out. In order to fill up a memory at full speed (one word per clock cycle) for example, the ALU is needed to compute the next address, so one of the neighbors can cooperate by maintaining the count, setting the extra bit as a signal when the buffer is full.

4.3.2 ALU

The computational capability of the cell consists of a simple 32-bit arithmetic logic unit. Standard ALUs provide very efficient use of silicon, since much of the circuitry is used for almost all the functions. We used a fairly basic set of instructions for this design: move, complement, increment, negate, add, subtract, and, or, exclusive or, add-with-carry. We also provided full shift and rotate. These are very useful for addressing, especially in the absence of byte addressing. Like some early computers and microinstruction architectures, there is only one adder, so memory indexing

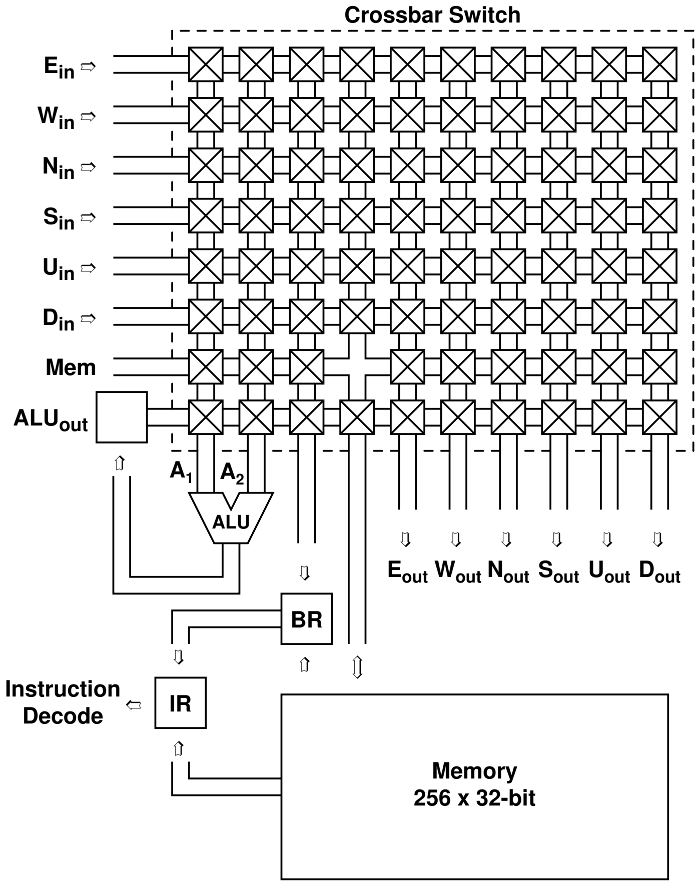


Figure 4.3: Block diagram of a cell.

must be done by arithmetic on instructions using the ALU. Since the ALU must complete each operation in half a clock cycle, the shift operations are fairly costly in area, requiring a barrel-shifter. Another instruction, MOVC, is provided to set the carry bit on a piece of data. This provides an easy way to signal another cell, for example to request service from the operating system.

Notably absent from this list are multiply and divide and floating point. These would greatly increase the size of the cell while decreasing the speed. They were left out of many early RISC chips as well, but later added. The overall philosophy of this project requires that the hardware only provide the basic computational primitives used by all programs—anything special-purpose should be programmed. Multiplication and floating point operations, when needed, should be done by explicit use of parallelism. This can be done very efficiently for multiplication and division (even in high precision—see chapter 6). It would be an excellent exercise to write fast parallel floating-point routines for the array.

4.3.3 Memory

The memory array is a 2-ported 1024-byte memory organized as 256 32-bit words, or, for instruction fetches, as 128 64-bit words. The dual ports are necessary to allow an instruction fetch (read-only) and a read or write by the program on every clock cycle. To implement a dual-port static RAM in CMOS requires 7 transistors if one of the ports is read-only, so the entire 8K-bit memory requires 56K transistors, plus a few more for address decode. This compares with only about 20K transistors necessary for the ALU, the crossbar switch, latches, instruction decode and control. This is a fairly good balance, since the memory array can be packed much more densely due to its regularity and straightforward connectivity.

Of course, SRAM is already an order of magnitude less dense than DRAM (dynamic RAM) which uses only a single transistor per bit. It would be reasonable to balance the logic with a much larger DRAM array, or a small SRAM and larger DRAM, and so be within a factor of two of the best density that RAM can achieve. Unfortunately, that density comes at a price—because the DRAM stores its charge in a small capacitor, much smaller than the capacitance of the wire it's connected to, the difference between a logic 0 and a logic 1 is only a tiny fraction of a volt on the output wire. This signal has to be boosted with analog circuits called sense-amplifiers. The sense-amps add a great deal to the size of the circuit, and may not be economical (in chip real estate) for small DRAM arrays. In addition, DRAM is inherently slower than SRAM and does not adapt well to dual-port configurations. Finally, the fabrication of high-density DRAM is different from other logic, and marrying the two is difficult (though two chip manufacturers, Micron Technologies and Cirrus Semiconductor, have announced plans to introduce DRAM arrays with custom logic on-chip).

For these reasons, and because I lack expertise with DRAM, I didn't consider it. It would, however, be a fruitful direction for further research. The IRAM project at Berkeley [Pat95] is pursuing similar research, investigating the idea of adding a processor to each DRAM chip.

4.3.4 Instruction set

Each instruction has to specify the action of the crossbar switch, the ALU function, the memory address, whether it must be read or written, and any conditional branching. The format we use is designed to be extremely easy to decode—any signal needed by the control can be easily generated from a few bits of the instruction.

As a result, the instructions are quite wide, like horizontal microinstructions—in this case 64 bits. When the instructions are fetched from memory, all 64 bits are loaded (since the memory array's natural aspect ratio is nearly square, it will likely be 64 or 128 bits wide internally). If the instruction is loaded from the ALU output or from one of the directional inputs, only the even word is loaded, and the odd word remains unchanged.

	31 29	28 26	25 23	22 20	19 17	16	15 12	11	10	9	8	7 0
Even word	E_o route	N_o route	U_o route	Mem route	BR route	BL	Branch	0	1	BE	BN	Mem Addr
Odd word	W_o route	S_o route	D_o route	A_1 route	A_2 route	0	ALU func	0	0	0	0	Next instruction Addr

Figure 4.4: Instruction format.

The format of an instruction is shown in figure 4.4. About half of the space is taken up by fields to control the crossbar switch. There are 10 destinations: 6 directions, 2 ALU inputs, the branch register, and the memory bus. Each of the destinations has a routing field in the instruction word. This specifies which of the 8 possible sources are connected to it. The coding of the routing fields is shown in table 4.1. Since there is only one memory bus, which is bidirectional, it does not make sense to connect memory to memory—the "Mem" routing for the memory bus is reserved to indicate a memory read. If the routing is anything else, a memory write is performed.

There is no adder for the instruction address, so a separate field is used to hold the next instruction address. Conditional branches are accomplished through the branch register (BR), which must be loaded in advance of the branch; in this way, delayed branches aren't necessary. If the branch condition specified in the instruction is met, the contents of the BR are used as the next instruction. If not, the instruction fetched from the "next instruction addr" is used. The BR is loaded just like the instruction fetch—if it is loaded from memory, all 64 bits are loaded; if from the ALU or one of the cell inputs, only the even word is loaded. It is only loaded, however, if the BL bit is set; otherwise it remains unchanged. The coding of the ALU and Branch fields is shown in tables 4.2 and 4.3.

The BE and BN bits are used for forcing a neighboring cell into a known state—their function is described in section 4.3.5 below.

We use a very simple assembly language, allowing instructions of the form:

ADD Ei,Mem; Eo←Ao; Bnz; go(Loop)

The first operation specifies the ALU function, the two fields after it give the routing for the two ALU inputs, after that follow fields in any order giving routing for other

Table 4.1: Routing codes.

E_i	0
W_i	1
N_i	2
S_i	3
U_i	4
D_i	5
A_o	6
Mem	7

Table 4.2: ALU functions.

MOV	0
COM	1
INC	2
NEG	3
ADD	4
SUB	5
AND	6
OR	7
XOR	8
ASHIFT	9
LSHIFT	10
ROT	11
ADDC	12
MOVC	13

Table 4.3: Branch codes.

no branch	0
BRA	1
BPL	2
BMI	3
BC	4
BNC	5
BZ	6
BNZ	7
BCE	8
BCW	9
BCN	10
BCS	11
BCU	12
BCD	13

destinations (any left out default to Ei), the next instruction address (if omitted, the next address), and the branch condition (defaulting to "never branch").

4.3.5 Programming

To handle programming within the paradigm of nearest-neighbor communication poses some problems. Since we cannot guarantee that one cell will not somehow run amok, there must be a way to forcibly reload it. Even if we could always guarantee correct operation, when the array is first powered up, the cells may be in some indeterminate state and need to be forcibly bootloaded. If this forced reload is not to be a global operation, it must be done by the cell's neighbors. But we are now on dangerous ground, since a malicious or malfunctioning piece of code in some cell could then start reprogramming all those around it, and eventually take over the entire machine. What would distinguish some virus, trying to take over the operating system, from the OS itself, trying to contain the virus? Conceivably there could be some supervisor-mode bit in each cell which would give that cell the ability to reload its neighbors and grant them the same privilege. But then power-up would still be problematic, as some of these supervisor bits might be left on. Worse, if a cosmic ray turned on the supervisor bit in one cell, it could then corrupt the entire array.

We adopted a more elegant solution, at the expense of breaking some symmetry. We require two canonically chosen neighbors—the west and south—to collaborate in rebooting a cell. Each has a bit which can be set in the instruction word to request a reload of its east or north neighbor. These are the bits labelled BE and BN in figure 4.4. If only one of the two neighbors sets its bit, nothing happens. If both do, the instruction register (the even word only) of the cell being rebooted is set from the west input, i.e. from one of the two cells requesting the reload. This cell, executing the instruction word from its west neighbor, can now propagate the signal to its east and north neighbors. Figure 4.5 shows a region in the process of being rebooted after three clock cycles: the instruction word I which is being passed along sets both BE (boot-east) and BN (boot-north) and copies the west input to the east output. However, the bootload can propagate only to cells that are directly east and directly north of cells that are cooperating.

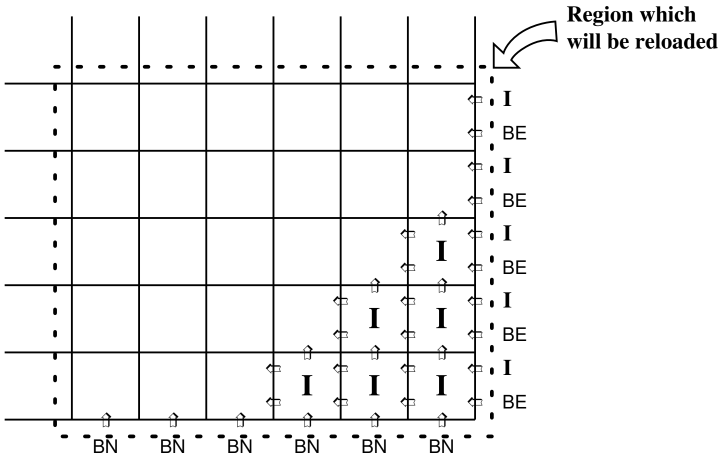


Figure 4.5: A region being rebooted.

If programs inhabit rectangular regions in the array, they can only forcibly reload cells within the rectangle. The operating system, which controls the periphery of the region, cannot be affected by reload signals from within, but it can reload the program inside the rectangle when necessary.

Rebooting an entire computer is straightforward—the reload sequence sets *all* the boot bits on the periphery of the array. The inputs on the entire west face of the array are set to a (constant) instruction, and the south input, say, of the cell in the west-south-down corner is given a sequence of instructions from ROM. These instructions tell it how to initialize the rest of the array. The boot bits must remain on long enough for the entire array to be initialized to some receptive state. This takes time at least equal to the sum of the dimensions of the array.

In three dimensions, we could require *three* neighbors to collaborate in order to reboot a cell, say west, south and down. Nothing significant seems to be gained by this, however, other than more flexibility (which we are unlikely to use) in the shapes of regions that can be allocated to processes. So we opted for the simple 2-dimensional solution.

4.4 Related work

Many mesh-connected computers have been designed over the 40-odd year history of parallel computation, starting from the ILLIAC IV in the mid-1960s [Hor82]. Many of them have been special-purpose machines for some application like vision or graphics; others have been SIMD supercomputers, like the Maspar or Goodyear's Staran; and some have been MIMD supercomputers. A group at MIT has produced a series of machines specifically designed for simulating cellular automata, the CAM series, currently represented by the CAM-8 [TM87, Mar96]. The CAM machines have proven particularly useful in simulations of physical systems.

Systolic arrays [KL78], introduced by Kung in the mid 1970s, are much like cellular automata sharing a global clock, and are designed to be pipeline elements, like the prototype CA we describe above. Many arrays have been designed for special purposes, frequently one-dimensional. Special-purpose two-dimensional arrays have also been used for problems which have a natural 2-d structure like matrix algorithms, and fully general systolic processors have also been designed, like the iWarp [Kun90]. These lose some of the simplicity of the earlier designs, containing, for example, communication routing hardware.

At a level closer to hardware, field-programmable gate arrays (FPGAs) have been quite successful both in research and commercially. First introduced by Xilinx in 1986 and now available from many vendors, they are similar to the first prototype cellular array in that they allow software-reconfigurable logic, though the communication is not always nearest-neighbor. The name "reconfigurable computing" has been adopted for the applications of FPGAs; they have had some success in breaking through the wall between hardware and software, one of the goals of this work. We will describe one FPGA-based system below.

Indeed, there are so many cellular-automaton-like computers in existence that the question naturally arises why I did not use one of them as my experimental platform. In some cases, it was because I was unaware of the research as I began my own; in others (the supercomputers) because the individual processors were too large, so certain problems would not arise (like being unable to store the entire program in one cell) that must arise in cheap arrays; in others because the systems are too special-purpose and are incapable by themselves of supporting today's rich and complex operating environments.

We will describe two projects, PAM and NuMesh, which have special relevance to our work.

4.4.1 PAM—programmable active memory

A group at DEC's Paris research laboratory (now closed, unfortunately) investigated, starting in the late 1980s, what they called Programmable Active Memory (PAM). Their designs, called the Perle-0 [BRV89] and later the DECPeRLe-1 [VBR+96] are remarkably similar to our first prototype—each cell in the array has 4 inputs and 4 outputs for its north, south, east and west neighbors, and a function memory which determined how each of the outputs was computed from the inputs. There is also a single output bit which is fed as an input to the same cell on the next clock cycle. The function memory saves 5 entire truth tables for the 5 bits, so there are 160 bits of memory per cell. The system is synchronous—outputs change only on the clock cycle. Both generations of the PAM were implemented using FPGAs: the Perle-0 used one of the very first such arrays from Xilinx.

Although the PAM is conceptualized as a full mesh, at chip boundaries the current implementation connects only some subset of the boundary cells, due to the limited pinout of the chip. Programmers and compilers for the PAM must simply work around this limitation.

The PAM was not intended to take on general computational tasks, but it was intended as a universal hardware accelerator, a reconfigurable circuit to replace special-purpose hardware. And it showed considerable success in that arena, having been used for more than a dozen applications, ranging from calorimeters for high-energy physics to stereo vision and RSA cryptography [VBR+96]. The resulting solutions still require the expense of an add-in card, but they convey the advantages of a software implementation—flexibility, ease of prototyping and debugging—with the high performance of hardware. PAM has convincingly shown the viability of "soft" hardware.

4.4.2 NuMesh

NuMesh is a project of MIT's computer architecture group. Its goal is high-bandwidth systolic communication in a thoroughly modular system. Meshes are intended to be as easy to extend as LEGO-brick structures. The packaging and interconnect technology supports this by allowing the basic modules to be simply plugged together in 2-d and 3-d meshes.

A node in the NuMesh [WAD+93] has 4 bidirectional communication links to neighbors in the mesh. A communication processor on each node allows forwarding of data without any involvement by the main processor. This processor is based on a finite-state machine and is specialized for statically compiled patterns of communication. The main processor (either a Sparc chip or a TI digital signal processor) can also access the communication links through FIFOs, though only reading from the north-south line and writing to the east-west line; this may require signals to be redirected by the FSM.

One of the unique features of NuMesh is its support for both 2 and 3 dimensional arrays with the same node architecture. This at first seems impossible, since only 4 neighbor links were provided. The three-dimensional array uses an ingenious 3-d interconnection scheme, the diamond lattice, which is illustrated in figure 4.6. Such a geometry is very appealing for the cellular array we described above, since it cuts down significantly on the complexity of the crossbar switch. The only drawback is that the array may be less adaptable to highly regular computations. The RSA algorithm presented in chapter 6, for example, is very well suited to a simple grid.

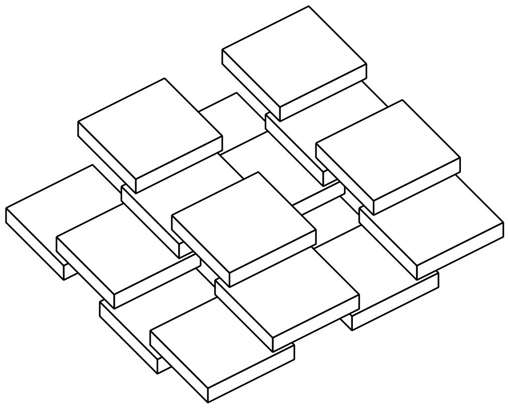


Figure 4.6: Diamond lattice interconnection.

4.5 Packaging

To actually build a computer based on this architecture, we would need to face the problem of chip interconnection. The data paths between the processors are quite modest within the chip, but on the edge of the chip, the macroscopic size of the pins severely limits the number of connections. The planar nature of VLSI would naturally restrict us to a 2-dimensional array. Using the same technology as the Pentium II, we could fit at least a 10×10 array of processors on a chip. Each edge would have 10 33-bit input channels and 10 33-bit output channels, for a total of $4 \times 66 \times 10 = 2640$ pins, not counting a number of pins for power, ground, and clock. This is an order of magnitude more than the available pinout.

One solution, used by the PAM project, is to restrict off-chip connections to some subset of the boundary cells. This would, unfortunately, complicate the development of algorithms, but might be necessary. Another possibility is building larger arrays by making larger chips, up to the size of an entire wafer. Wafer-scale integration, once a hot topic, foundered for lack of a clear application with enough volume to justify the expense. With cellular arrays, the larger arrays could be as much a commodity as large DRAMs are. But this still leaves formidable problems, such as low yield and the necessity of bypassing bad cells in the array.

It may also be possible to develop different packaging and interconnect technology that takes advantage of the fact that the wire runs are short and low-capacitance. If chip interconnections were done by photographic processes and vapor deposition, the bonding sites could be made much smaller.

Perhaps the most plausible solution given current technology is to partially serialize the communication between the chips using an extremely fast clock rate. This takes advantage of the physical proximity of the neighbors. A similar technology, called RAMBUS, is being used now for high-performance memory systems. RAMBUS uses only a byte-wide data path, but up to a 1GHz clock, a speed made possible by short and low-capacitance wires and straight runs. This keeps wire length the same and minimizes clock skew. The same features are present to an even greater degree on the cellular array, so fast serial communication should be possible.

It is a pleasant diversion to speculate about three-dimensional interconnection, but any practical solution would require novel chip fabrication technology. We should note here that Seymour Cray, in the Cray-3 computer, stacked chips (the actual die, not the package) up to 4 high. To keep the profile low, he had the backs of the dice milled down from their original thickness. All interconnection was still provided at the boundaries of the chips, of course.

Chapter 5

Programming

> If the hill will not come to Mahomet, Mahomet will go to the hill
>
> — Francis Bacon, in the essay "Of Boldness"

The single largest challenge for parallel supercomputer systems up to the present has been the difficulty of programming them. As we pointed out earlier, some of the difficulty has been in the extremely complex problems supercomputers were used to solve, and some has resulted from the legacy of large, unwieldy, and poorly-understood codes which purported to solve them.

But parallel programming seems to be inherently more difficult than sequential programming, even for quite simple problems. Consider the famous maximal independent set problem—to find a set of vertices in a graph which are independent (no two of them share an edge) and which is maximal (every other vertex in the graph has an edge to some vertex in the set). This requires hardly any thought to do efficiently on a sequential machine, yet Michael Luby's elegant parallel algorithm is far from obvious.

On a cellular array, it is not even clear what a program is. Does each processor have a program? If so, the programs can't be very big, since the cells have very limited memory. Does the program reside in a large memory constructed from a number of cells, from which it is disseminated? Or is the program compiled into a circuit, as if by a silicon compiler, and laid out in cellular space? How is a subroutine called? Does the code get sent out to the data, or is the data passed to the code? Or is the code permanently interspersed with the data, ready to be used at any time? The choices seem almost dizzying. The aim of this chapter is to make some sense of those choices. We will discuss several of the styles of parallel programming that are common now and show which are the most natural on CAs. We will show what constitutes a program for a CA by giving the details of loading a program, and we will survey some of the known algorithms for CAs.

5.1 Parallel programming styles

To a considerable extent, the range of choices on the cellular array stems from the universality of the CA. It can emulate any other parallel architecture, so any parallel programming paradigm is available. This may seem like a clear benefit, since we can use whatever architecture works best for a particular application. Or it can seem like a drawback, since the difficulty of programming seems to be proportional to the number of decisions that must be made.

Here we consider the different styles of programming implied by the common parallel architectures, and their implications for cellular arrays.

5.1.1 Dataflow

Dataflow computers [AB93] are parallel machines built to directly execute data flow graphs, which are generalizations of expression trees, allowing cycles (for loops), multiple uses of a value, and procedure calls. They are based on the observation that a great deal of parallelism is present in most programs as they are written, which the compiler somewhat arbitrarily resolves by the order in which it emits code. In a dataflow computer, an operation is ready to execute as soon as the values it depends on have been computed, and it will then be executed by the next available processor.

Dataflow computers at first required special languages which made the data flow graphs particularly easy to compute. These avoided certain language features, like multiple assignments, aliasing, and pointers, that can make it difficult to figure out (for a compiler or a human reader) where a particular variable's current value came from. More sophisticated compilers eventually were constructed that do an excellent job of data flow analysis even for standard languages like Fortran. The compiler technology that grew out of the work on dataflow computers eventually undermined the machines themselves, as compilers using dataflow concepts were successfully applied to other parallel architectures, and even to code rearrangement for sequential machines.

Data flow graphs have been applied to regular problems, both by hand (for systolic array design, see section 5.1.4) and automatically (for detecting dependencies in array code). The techniques of program analysis that dataflow computers inspired will likely be an integral part of compilers for any future parallel languages.

5.1.2 SIMD

SIMD (Single instruction-stream, multiple data-stream) machines provide an array of processors each of which execute the very same sequence of instructions in lock-step, but using their own local data. The instructions are broadcast from a host computer, a standard sequential machine. Examples of SIMD include the Illiac IV, the Staran and MPP from Goodyear Aerospace, the MASPAR, and various generations of the Connection Machine. In highly regular problems like matrix arithmetic, most data is undergoing the same operation and the problem maps very well onto a SIMD machine. The architecture is particularly well suited to "data-parallel" language constructs: performing a particular piece of code on each element of a large dataset. The processor array can be thought of as a smart memory, where the data lives, and it is the instructions which are sent out to operate on the data.

To handle conditional execution within the code sequence, these machines usually have a way for certain machines to conditionally disable themselves so they do not execute instructions from an IF-statement whose condition wasn't met by their data.

In compensation for having extremely simple logic and small memories, SIMD machines usually usually have a much larger number of processors than other contemporary machines; however they have fallen out of favor recently due to the low price of off-the-shelf microprocessors.

The processors can communicate data with each other through some communication network, whose characteristics vary according to the model. The MASPAR, for example, uses a mesh interconnect, while the CM-5 used a fat-tree.

5.1.3 MIMD

MIMD (Multiple instruction-steam, multiple data-stream) machines provides processors which are much like fully functional sequential computers. The processors execute completely independent instruction streams. The cellular array described in chapter 4 is a MIMD machine, though it is very atypical. Usually, MIMD machines have enough local memory that each processor can keep a full copy of the running program locally, whereas the cellular array, like many of the SIMD machines, has a much smaller local memory.

Existing MIMD machines also have many different communication schemes, which is the chief architectural difference between them. The two main camps are shared-memory multiprocessors and message-passing machines.

Shared-memory

Shared memory multiprocessors are MIMD machines which communicate through a large shared memory. In contrast with SIMD machines, in which instructions are sent to the data, in shared-memory systems the instructions generally stay put, while the data is constantly shipped around. The PRAM model is an abstraction of this architecture. The machines are a natural outgrowth of the small-scale parallelism used for mainframe time-sharing systems in the 1970s, and increasingly used for microprocessor-based servers today. In these machines, a few (2 to 8) processors are connected to a central memory. In extending this to highly parallel machines, many new techniques had to be invented to prevent the memory from becoming a bottleneck.

The benefit of the shared-memory architecture is that it allows relatively easy and fine-grained sharing of data, and relatively easy adaptation of sequential programs. The disadvantage, however, is that the performance gains are often modest unless much more attention is paid to data layout and locality, which often requires detailed knowledge of the communication and memory systems—features the architecture is trying to hide.

Message-passing

In message-passing MIMD machines, all processors have a local memory and communicate by explicit I/O operations through the interconnection network. The overhead for sending or receiving a message is generally higher than for a shared-memory system, but the transfer rate (once the message begins transmission) is generally much higher. This rewards bulk communication—relatively infrequent messages containing a lot of data. Programs usually don't take advantage of the specific interconnection network. This pattern of infrequent high-volume communication is called "coarse-grained" parallelism, and is captured very well in two abstract models, BSP and LogP (see section 3.2.3).

5.1.4 Systolic

Systolic arrays, like SIMD machines, support fine-grained parallelism. Some systolic arrays are, in fact, SIMD machines, but others allow local programming, and others have completely hardwired control. The processing elements are generally connected in a linear array or a 2-dimensional mesh. Each element can send a data value to its neighbor (or neighbors) on each step of a global clock. The data values thus circulate through the array in a way similar to the circulation of the blood through the human body. Systolic arrays have been used as a structured technique to design hardware solutions to certain computationally-intensive problems and as a general machine architecture for implementing fine-grained parallel algorithms (the iWarp machine, for example [Kun90]). They have proven particularly successful in matrix algorithms and signal processing [Kun88b].

Of all the architectures we consider, systolic arrays are the most like the CA architecture we describe here. Indeed, cellular arrays can be thought of as a kind of systolic array. This means that a large body of algorithms for computationally difficult problems can be imported directly for the machines we describe. Perhaps more important, there is a well-developed methodology for transforming highly-structured problems, through their dependence graphs, into systolic algorithms [Kun88b].

5.1.5 Cellular arrays

Programs for cellular arrays can be written in any of these styles. Which is best? Once again, the presence of too many different models of parallel programming leads to confusion. But we can make certain observations. Anything written for the systolic array style will work very efficiently on a CA. SIMD algorithms will also translate quite efficiently as long as there is not too much dependence on low-diameter communication networks like hypercubes. If a program is written to use only mesh-interconnection (like the MASPAR), it should be possible to run it as efficiently on a CA. Although SIMD machines seem to rely on instantaneous global communication of the instruction word, this can be simulated by retiming with a loss of only a factor of 2 in efficiency.

Even though a cellular array is a kind of MIMD computer, it is slightly harder to transport a MIMD program to run on a CA. To do so, we will generally need to allocate small clusters of cells to act as the processors of the MIMD computer. This is because the individual processors of most MIMD systems have memories significantly larger than our cells have. Each cluster will need to be large enough to hold the processor's local data and program.

The communication architecture of the MIMD system may be very different from the local interconnect of a CA, but few applications depend on the low-level details of the network. High-level abstractions, like BSP, can be implemented efficiently on the mesh using well-known packet routing algorithms (see section 5.3).

The natural programming style of a CA is somewhere between these two. The code present on an individual processor need not be limited to a single instruction, as SIMD would. That forces too many inefficiencies when processors aren't acting in perfect lock-step. But to try to disseminate too large a program into the data forces the program to span several cells, which introduces its own inefficiencies. This can be productive up to a point, but the fastest algorithms will have only small code-bursts (around half the memory of a single cell) to accelerate some data-parallel operation or operations. A subsequent phase of the main program may distribute different code. The document formatting problem in chapter 7 uses this technique—only simple self-contained operations are performed on all the document in parallel, but the main program (which might be a word processor, for example) could be extremely complex.

The (large) main program may have only one copy, and be executed sequentially, but any portion which affects all the data will be executed in a data-parallel fashion by sending code out into the portion of the array containing the data. Zippel argues for a similar approach to add data-parallelism to sequential machines through a simple SIMD coprocessor called the data-structure accelerator [Zip91].

5.2 What is a program?

After considering all the different styles of parallel computer, the entire concept of a program can seem a little vague on the cellular array. To provide a more solid footing, we will now describe the programming concretely.

When a program is started on the cellular array, a region of the array is forced into a receptive state using the boot bits. Then a stream of words is introduced into the corner of the region. That stream of words is the program.

The beginning of the stream is a loader, which is loaded into all the processors. The loader we have written next computes the coordinates of the processor, a word for each dimension. Then it broadcasts the rest of the words to every cell in the region as a series of blocks. There is a count at the beginning of each block, but the operating system also sets a flag bit at the end of the block so the loader doesn't have to use its ALU to keep count (the ALU is instead used to increment the address to store the next word). At the end of each block is an instruction to execute—this can return to the loader, decide if the next block belongs in this cell (and ignore it if not), or start up the program.

The loading sequence is designed to allow the program to be injected into the array one word per clock cycle. Although program loading is sequential, at 200MHz a 1Mb program would take only a little more than a millisecond to load. The coordinates in each cell are used to decide which code should be stored in which cell. For a program in which all cells have the same structure except the edges, a simple comparison is enough to decide whether to load or ignore the next block. If the cells all have the same code, but each receives a different piece of initial data, then the data can be sent in a stream and each processor counts down the list according to its coordinates, and then seizes the value belonging to it. Both of these techniques are used in the RSA program (see chapter 6).

To tile the array with a cluster of cells, as might be needed to simulate a MIMD computer, the coordinate in each dimension must be reduced modulo the period of the tiling. This could be done by making the processors count off modulo the tile size, which would involve no division. Unfortunately, in our prototype cell it would require a conditional branch, so the count cannot propagate at the full speed of one cell per clock cycle. This would force the entire rest of the loading sequence to also slow down to half speed, so the division is actually preferable. Fortunately, despite the absence of hardware multiply/divide, reduction modulo a small constant is still a fast operation (much faster than general division). If we want to reduce j modulo the constant n, we can use the iteration

$$j := (j \bmod 2^i) + (2^i \bmod n) \cdot \left\lfloor \frac{j}{2^i} \right\rfloor$$

This halves the number of bits on each iteration, involving only the operations add, shift, logical AND, and possibly multiplication by a small constant. The iteration for $n = 3$, for example, is just to add the top half of the number to the bottom half.

Finally, the most familiar form of loading (for sequential computers) is non-replicated, where each cell gets a different portion of the program. This is very easily handled by labeling each block with the coordinates of its destination cell.

5.3 Algorithms

The literature contains a great number of parallel algorithms which will run on CAs. Some of them come from work on systolic arrays, others from mesh-connected MIMD machines. The existence of a large body of basic algorithms is essential to the success of any machine in the marketplace, so we will summarize some of the results here.

Sorting and packet routing

The problems of sorting and packet routing are intimately connected and are essential subroutines in many other algorithms. Their importance for CA algorithms comes from the fact that data must have the correct geometry to permit local communication. Sorting and routing are general-purpose tools to take data in any order, and rearrange it into the geometry required by the algorithm.

The lower bound for both sorting and routing is clearly on the order of the diameter of the portion of the array containing the data ($\Omega(\sqrt[d]{n})$ on a d-dimensional array), since a data value will, in the worst case, have to travel from one corner of the array to the diagonally opposite corner, requiring a minimum of $d\sqrt[d]{n}$ steps.

This lower bound can be met, either for packet routing or sorting, within a constant factor [Kun88a].

These results are for arrays in which one value is stored in each cell. With the array of chapter 4, we can store up to 256 values in a single cell. In that case, the I/O bandwidth of each cell becomes a limiting factor. Consider a d-dimensional array with r cells on a side, and k values held in each cell. Sorting the whole array is called k-k sorting, or k-k routing if the destinations are known in advance. If we split the array down the middle, there are $kr^d/2$ values on each side. In the worst case, all of them must migrate to the other side, across a boundary consisting of r^{d-1} nodes. Since only one value can cross at each cell in each time step, this will require at least $kr/2$ steps for either sorting or routing. That bound has been achieved (with the addition of lower-order terms) both for sorting [Sib94] and for routing [Kun93] on the 2-dimensional mesh. Several algorithms with near-optimal performance generalize to higher dimensions as well [Sib95].

Matrix algorithms

The 2-dimensional CA is particularly well-suited to matrix arithmetic. Two $n \times n$ matrices can be multiplied in an obvious way by an $n \times n$ array of processors in $O(n)$ time, an algorithm which executes only $O(n^3)$ operations, which matches the work performed by the usual sequential algorithm. LU decomposition can be performed by Gaussian elimination in $O(n)$ time also (with a larger constant factor), but only without pivoting, which reduces the numerical stability in many cases [Lei92]. Unfortunately, determining the complete pivoting sequence for Gaussian elimination is ¶-complete, which is compelling evidence that it is an inherently sequential process [Vav89]. Fortunately, other factorizations work very well on the mesh. QR factorization, which does not involve pivoting, can also be performed in $O(n)$ time on an $n \times n$ array, and can, like LU decomposition, be used for solving systems of linear equations. The same applies to Cholesky factorization of symmetric matrices [GvL89].

Since these matrix algorithms are work-efficient (each of them performs no more operations than the normal sequential algorithm), we cannot hope to do better on a 3-dimensional array unless we use more than $O(n^2)$ processors, i.e. more processors than the size of the data. If we do, we can multiply two $n \times n$ matrices in time $O(n^{3/4})$ times on an $n^{3/4} \times n^{3/4} \times n^{3/4}$ array [Lei92].

Signal processing

The single most important algorithm for signal processing is discrete Fourier transform (DFT). Although the Fast Fourier Transform (FFT) algorithm is not suitable for a CA, a slight variation works well. Instead of subdividing the vector into two interleaved subvectors and recursively computing the FFT of both, we can subdivide it into r subvectors, then compute the DFT of these. If the CA is 2-dimensional, the subvectors should be approximately r elements, and their DFT can be computed by a straightforward convolution algorithm. If the CA is 3-dimensional, we make one more subdivision and a recursive call. The size of the original vector, in this case, should be of the form r^3 instead of 2^n as is required by FFT.

Many techniques for filtering have been implemented with systolic arrays and allow fast parallel implementations on a CA. These include finite impulse response (FIR) filters, equivalent to non-recursive convolution, and infinite impulse response (IIR) filters, or recursive filtering [Kun88b, Rob87]. Other techniques specifically applicable to images have been well-known for cellular automata [JD84] and systolic arrays [Kun88b]. They include edge detection, line and curve-detection, pattern matching, image segmentation, chromosome recognition and cell counting.

Graphics

Graphics is an area hungry for processing power. Much of the demand for faster processors in the PC world is driven by applications like virtual reality, games, and MPEG-encoded video. If parallelism is to deliver high performance anywhere, graphics is the area to do it. Given the importance of graphics, we could have devoted all the research to it. The only reason we did not is because graphics parallelizes so well. Graphical applications are extremely well-suited to cellular automata because of the obvious mapping of two-dimensional processor arrays onto the image plane.

Simple graphics rendering is inherently parallel, since each region can be painted independently of the others. Many 3-dimensional rendering systems eliminate hidden surfaces by sorting the surface patches by depth, then rendering them in back-to-front order. This process algorithmically depends on nothing more than sorting, first in the x and y planes to distribute patches to regions, then in the z plane to make nearby objects occlude more distant ones. All of these can be performed efficiently on a CA. More expensive algorithms like ray-tracing have also been implemented on mesh-connected parallel computers, achieving real-time performance [SK93, LRN93].

Video decoding and processing also have a great deal of inherent parallelism and should adapt well to the CA. MPEG-2 [HPN97] encoding uses the discrete cosine transform (DCT) for image compression, which is similar to the DFT. Although possessing as much inherent parallelism as DFT, it may not benefit much because the image blocks it acts on are small, 8×8. Each of these blocks is compressed separately, however, so these blocks can be compressed or restored in parallel. At a higher level, the motion compensation of MPEG uses “macroblocks” of 16×16 pixels to make predictive compression more effective; these can also be restored in parallel. And at a still higher level, full video frames can be processed in parallel—a subset of the frames (the P-pictures) must be decoded in order, since each depends on the previous one; in between each pair are a number of frames called B-pictures which depend only on the two nearest P-pictures but are independent of each other. These B-pictures can also be processed in parallel.

In the field of computational geometry, many classic problems can be solved with efficient mesh algorithms. For sets of points in the plane, the convex hull of the set, the nearest neighbors of all the points, and the Voronoi diagram of the set can all be computed in $O(\sqrt[2]{n})$ time on a 2-dimensional mesh. Algorithms are also known for triangulation of a polygon, detecting intersections in a set of line segments, and for locating the region of a planar subdivision in which a point lies [AL93].

Chapter 6

RSA Encryption

The control of large numbers is possible, and like unto that of small numbers, if we subdivide them.

— Sun Tzu, Sun Tzu Ping Fa

This chapter shows how a cellular array can support a very fast implementation of RSA encryption. We chose this problem for two reasons: it is a very time-consuming operation that is becoming increasingly common; and it is a good test case to evaluate whether general-purpose parallelism can replace special-purpose hardware accelerators.

Public-key cryptography has recently moved from an academic sideline to take center-stage in the world of electronic commerce. Every major system for public-key cryptography has a very high computational cost, in contrast to the many quite speedy secret-key systems available. Despite this drawback, public-key systems like RSA have enormous benefits, and they are likely to become increasingly widely used; fast implementations are therefore extremely desirable.

Software implementations of RSA have been getting significantly faster, primarily because of the faster clock speed and wider ALUs available on newer microprocessors. Unfortunately, Moore's law is also helping the codebreakers, perhaps even more than it helps encryption, since factoring algorithms can use parallelism very effectively, as Lenstra and others have so convincingly demonstrated by breaking the original RSA challenge message [Len94]. In addition, internet traffic, and presumably also secure traffic, is increasing even faster than computer speeds, so we need to find other avenues to speed up RSA.

There have been several hardware implementations of RSA giving significantly higher performance than software. All these implementations use some form of parallelism, even if only in the form of very wide ALUs. We have made the claim that general-purpose parallelism can replace special-purpose hardware accelerators, and that we gain several advantages in the process: economies of scale, simpler architecture and system design, and more flexible allocation of resources. The RSA problem is a good test case for this claim.

There are several other obvious candidates to serve as test cases: graphics accelerators, signal processors, network interfaces, and so forth. But these are all fairly easy—the cellular array architecture is very well suited to graphics and video algorithms, which map well onto nearest-neighbor grids, and network interfaces or modems already use fairly general-purpose CPUs. RSA seems like a more challenging problem—its need for high-precision arithmetic appears to favor special-purpose hardware. We do not know of any previous parallel software implementation.

6.1 Opportunities for parallelism

Recall the basic operations of RSA [RSA78]: the public key (e, m) consists of an exponent and a modulus, and the encryption operation transforms a message $t < m$ into $t^e \bmod m$. The private key (d, m) uses the same modulus but a different exponent; the operation is the same, modular exponentiation. The modulus m is chosen to be the product of two large prime numbers p and q, and e is usually chosen to be 3 or some other small number, so that public-key operations are reasonably fast (about $O(n^2)$ operations, where n is the size of the modulus). The private exponent d is of the same order as m, so private key operations require $O(n^3)$ time. Because of the great speed disparity, I will focus on speeding up private-key operations.

Given a message t, a modulus m, and an exponent $e = \sum_{i=1}^{n} e_i 2^i$, the basic modular exponentiation algorithm used by RSA loops over each bit e_i of the exponent:

1. $x := t$
2. $v := 1$
3. for $i := 1$ to n
4. if $e_i = 1$ then $v := vx \bmod m$
5. $x := x^2 \bmod m$

The final value of v is the result, $t^e \bmod m$.

There does not seem to be any way to perform modular exponentiation faster than the method of repeated squaring, so there appears to be an inherent sequentiality to the main loop of RSA. Nevertheless, there are four clear opportunities for parallelism in this basic algorithm:

- If there is a stream of messages to be decrypted (or signed), each of these operations can be performed independently on a different processor (or set of processors). This will not speed up the elapsed time for a single private-key operation, but it can greatly speed up the overall throughput.

- Step 5, squaring, can be performed in parallel with the multiplication in step 4 from the previous iteration, saving some 33%. Note that this is only possible with the loop running from low-order exponent bits to high-order. It is more common to scan the exponent in high-to-low order, but doing so removes some inherent parallelism.

- For private-key operations, we may assume that the factors p and q of m are known. The modular exponentiation can be performed separately and in parallel mod p and mod q, and then the two results can be combined by the Chinese remainder theorem. This is often done in fast software implementations, since two half-size modular exponentiations are still four times as fast as one full-size exponentiation.

- Finally, the high-precision multiplications in steps 4 and 5 are slow ($O(n^2)$) operations. Using standard long-multiplication with 32-bit words and a 512-bit modulus, there are 16 words in each number and $16^2 = 256$ 32×32 bit multiplies to perform. These multiplies can all be performed and the results summed in parallel. Note that although there are asymptotically faster multiplication algorithms, including FFT and Karatsuba [Knu91a], they do not appear to be competitive until much larger numbers are used than RSA [Koç94]. Unlike the other methods, parallel multiply requires an architecture that supports fine-grained parallelism—it is not practical if the communication overhead between processors is much larger than the multiplication time.

A good parallel implementation of RSA will certainly incorporate the first three of these ideas. The first, performing several RSA operations independently on different processors, is the most scalable, and arguably the most important. But we would still like to speed up the response time of a single operation, for which the other three techniques are needed. Performing squaring in parallel with multiplication and exponentiating modulo the two prime factors together gain only a factor of 3. The rest of this chapter investigates variations on the last idea—using parallelism to speed up multiplication.

6.2 Parallel modular multiplication

The operation we must perform in RSA is not simple multiple-precision multiplication but *modular* multiplication: $xy \bmod m$. The sequential implementation of this operation generally requires about twice the time of simple multiplication, since each $1 \times n$ multiplication step is followed by a $1 \times n$ modular reduction step. This interleaved method is inherently sequential and has no obvious analogue in the parallel multiplication described above. There are parallel algorithms for division and remainder that are approximately as efficient as multiplication and in fact use multiplication as a subroutine [Lei92]. To divide two numbers one can use Newton's method to form an approximate reciprocal of the denominator and multiply it by the numerator. In the case of RSA, the reciprocal approximation need be computed only once. The algorithm for modular multiplication then becomes:

$$\begin{aligned} &s := xy \\ &q := m^{-1}s \\ &r := s - qm \end{aligned}$$

The intermediate products s and qm require twice as much space as the modulus. The quotient q is only approximate, so a correction step is normally necessary, but the remainder will still be in the same residue class mod m, so we can accommodate the possible error by allowing an extra one or two bits of precision for the intermediate results and only doing the full reduction on the last step.

This is indeed of the same order of complexity as multiplication, but is slower by a factor of 3, since each step requires a high-precision multiplication (on a sequential machine, this would be a factor of 2, since we need only half the bits of q and qm, but on a parallel machine it is almost as fast to compute the entire product). Furthermore, each multiplication depends on the results of the previous one, so they must be done sequentially. My first implementation used this method, and it can be competitive with hardware implementations, but we can do better. The tools we need are Montgomery's method for modular multiplication (used in many software implementations of RSA) and residue number systems (also called modular arithmetic) where high-precision numbers are represented by their residues modulo a collection of small relatively-prime numbers. Neither of these techniques individually seems to help, but when combined, and with the help of a novel method for converting one residue number system to another, there is a surprising synergy. The final implementation is very close to three times as fast as the first—almost the speed of a single high-precision multiplication.

6.3 Montgomery's method

The purpose of the modular reduction after each multiply is to avoid the exponential growth in the size of the intermediate results, adding on a multiple of the modulus in order to zero the high-order bits of the product. Montgomery [Mon85] used an idea originally due to Hensel at the turn of the century, which is to zero instead the *low-order* bits of the product, and shift the remaining bits down. The residue classes are still represented uniquely, though it requires a final modular multiplication by a constant to restore the result to its normal representation.

Let $w = 2^n$ be at least $4m$, and choose m' such that $m'm \equiv -1 \pmod{w}$. Montgomery's method for modular multiplication works as follows:

$$\begin{aligned}
&s := xy \\
&q := m's \bmod w \\
&r := (s + qm)/w
\end{aligned}$$

Once again, the intermediate products s and qm have twice as many digits as m, x, y, or r. Since $qm \equiv -s \pmod{w}$, $(s + qm)$ will always be a multiple of w. And it is easy to verify that if both x and y are less than $w/2$ then r will be also.

Since w is a power of two, reduction modulo w and division by w are trivial operations, but it seems we haven't really made any progress, since this still requires three multiplications, just like the first method. But now we translate this method into a new representation.

6.4 Residue number systems

In a residue number system (RNS), a high-precision integer is represented by its residues modulo some set of integers. Multiplication, addition, and subtraction can all be performed element-wise, but comparison and division are in general difficult. Our plan is to implement Montgomery's method in residue arithmetic. Notice that Montgomery's method doesn't depend on w being a power of 2, it simply makes it easy to reduce mod w and divide by w. In residue arithmetic we will choose w to be a product of a subset of the moduli.

Let $v_1, v_2, \ldots, v_n, w_1, w_2, \ldots, w_n$ be relatively prime, and let $v = v_1 v_2 \cdots v_n$ and $w = w_1 w_2 \cdots w_n$. Given a number $x < v$ we can represent it by its residues: $[x_{v_1}, x_{v_2}, \ldots, x_{v_n}]$ where $x_{v_i} = x \bmod v_i$, and similarly for w. We will choose enough moduli so that $v > 4m$ and $w > 4m$: then either the v_i or the w_i can represent a reduced product, and we can use both together to represent a full product. Now the three multiplies are much simpler elementwise operations (they might be, say, 16×16 bit operations, reduced mod a 16-bit number). We have two more troublesome operations, though—reducing mod w in the second step ($q = m's \bmod w$), and dividing the remainder by w in the final step. But these can be performed using the operation of *converting* a number in one RNS (say the w_j) to another (the v_i). Denote this operation by $[x_{v_1}, x_{v_2}, \ldots, x_{v_n}] \leftarrow [x_{w_1}, x_{w_2}, \ldots, x_{w_n}]$. Assume we have precomputed the residue representation mod the w_j of $m' = [m'_{w_1}, m'_{w_2}, \ldots, m'_{w_n}]$ and similarly the representations mod the v_i of m and w^{-1} (the inverse of w exists mod v_i because w and v are relatively prime). Here are the steps for doing a modular multiplication of two numbers represented mod the w_j and producing the result represented mod the v_i:

Input: $[x_{w_1}, x_{w_2}, \ldots, x_{w_n}], [y_{w_1}, y_{w_2}, \ldots, y_{w_n}]$
Output: $[r_{v_1}, r_{v_2}, \ldots, r_{v_n}]$ where $r = xy \bmod m$

1. $[x_{v_1}, x_{v_2}, \ldots, x_{v_n}] \leftarrow [x_{w_1}, x_{w_2}, \ldots, x_{w_n}]$
2. $[y_{v_1}, y_{v_2}, \ldots, y_{v_n}] \leftarrow [y_{w_1}, y_{w_2}, \ldots, y_{w_n}]$
3. $s_{w_j} := x_{w_j} y_{w_j} \bmod w_j$ for $1 \leq j \leq n$
4. $s_{v_i} := x_{v_i} y_{v_i} \bmod v_i$ for $1 \leq i \leq n$
5. $q_{w_j} := s_{w_j} m'_{w_j} \bmod w_j$ for $1 \leq j \leq n$
6. $[q_{v_1}, q_{v_2}, \ldots, q_{v_n}] \leftarrow [q_{w_1}, q_{w_2}, \ldots, q_{w_n}]$
7. $r_{v_i} := (s_{v_i} + q_{v_i} m_{v_i}) w_{v_i}^{-1} \bmod v_i$ for $1 \leq i \leq n$

Since step 3 computes a representation of $s \bmod w$ and step 4 computes $s \bmod v$, if we combined the two with the Chinese remainder theorem, we would have the correct value of $s \bmod vw$, which is just s. It is also easy to see that steps 5 and 6 produce the same value of q as the standard Montgomery's method. The number r represented by the r_{v_i} in step 7 has the properties that (1) $r < v$, (2) $rw \equiv s + qm \pmod{v}$ and (3) trivially $rw \equiv 0 \pmod{w}$. Since $s + qm \equiv 0 \pmod{w}$, (2) and (3) together give us that $rw = s + qm$, so r is the correct quotient $(s + qm)/w$.

We are left with the problem that the algorithm started with the inputs mod the w_j and ended with a result represented mod the v_i. But we don't need to convert back again—alternate iterations of the main exponentiation loop can switch back and forth between w and v. Several other ways that the algorithm seems cumbersome also turn out to be easier than they look. For example, the two RNS conversions in steps 1 and 2 are never both necessary—for squaring they are both the same number, and the other modular multiply is by a number that is also being squared, so its RNS conversion is available for free. Although there are still two global operations (the RNS conversions) instead of three for the straightforward modular multiply, these two are independent and can be performed in parallel. And finally, the RNS conversion algorithm allows us to fold the multiplies in steps 5 and 7 into the conversion process without any penalty, as we will see shortly.

6.5 Residue conversion

Converting from one residue number system to another was discussed by Hitz and Kaltofen [HK95] in the context of division in a RNS. Unfortunately, their algorithm is both fairly complex and computationally costly, essentially forming the full multiple-precision value (in a mixed-radix system) of the number represented by the residues, then converting that to the output RNS. I describe a new algorithm here that is both faster and simpler.

The starting point for this algorithm is also an application of the Chinese remainder theorem to generate the large integer represented by the residues. But as we go, we reduce that integer mod each of the moduli in the output RNS. It will turn out that almost all the work can be done on small integers. By the CRT, the number represented by $[x_{w_1}, x_{w_2}, \ldots, x_{w_n}]$ is:

$$x = \left(\sum_{j=1}^{n} c_{w_j} x_{w_j} \right) \bmod w$$

for easily computed constants c_{w_j}. For now, ignore the final reduction of this sum mod w, and let x' be the unreduced sum. We can compute the value of $x' \bmod v_i$ as a simple inner product of small integers modulo a small integer:

$$x' \bmod v_i = \left(\sum_{j=1}^{n} (c_{w_j} \bmod v_i) x_{w_j} \right) \bmod v_i$$

But we cannot ignore the mod w. To account for it, we have to subtract $w \lfloor x'/w \rfloor$ from x' or subtract this amount mod v_i from each x_{v_i}. Notice that $\lfloor x'/w \rfloor$ is not very large—it is bounded by the sum of the w_j since $c_{w_j}/w < 1$, so the necessary adjustment is also a small integer. Although the exact value of x'/w may be difficult to compute, we can get an approximation of its value with a fractional inner product like the one above. With, say, 18-bit moduli and a 32-bit approximation, we would obtain the correct value of the integer part almost always. Since we can distinguish the cases in which our approximation might be wrong, we can handle it as an exceptional case and compute the exact value. This will occur so rarely (about 1 time out of 64) that the average run-time will be affected insignificantly.

We can now combine the various multiplications by constants into this residue conversion process. If we wish to multiply x by some constant k mod w before converting to v, we can simply use the coefficients kc_{w_j} mod w instead of c_{w_j} in the above. Note that this affects the computation of $\lfloor x'/w \rfloor$, since the x' uses the new coefficients. To multiply by some other constant l mod v *after* converting to v, we can multiply l mod v_i into every coefficient of the inner product to compute x_{v_i} (reducing the coefficient modulo v_i) and into the adjustment coefficient w mod v_i that we multiply by $\lfloor x'/w \rfloor$. The full formula, then, for converting from x represented in the w_j system to $y = l(kx \bmod w) \bmod v$ represented in the v_i system is this:

$$a = \left\lfloor \sum_{j=1}^{n} x_{w_j} \frac{kc_{w_j} \bmod w}{w} \right\rfloor$$

$$y_{v_i} = \left(\sum_{j=1}^{n} x_{w_j} l(kc_{w_j} \bmod w) \right) - la \bmod v_i$$

One last detail: this allows all the multiplications by constants to be folded into the conversion except one: multiplying s by w^{-1} in the v_i residue system in step 7 of the algorithm. But if w^{-1} has a square root mod v, we can fold that into the conversions of x and y in steps 1 and 2, and when we take the product (or square) in step 4, the product s will effectively have been multiplied by w^{-1}. An easy way to ensure that w^{-1} has a square root is to make all the moduli $w_1, w_2 \ldots, w_n$ squares, which is what our implementation does.

6.6 Implementation

In the CA implementation, we use 16 moduli in each v and w, with all the moduli less than 2^{17} for a 256-bit modulus. Since each prime factor is 256 bits, the key length is 512 bits. Each full 256-bit modular multiply requires only 56 cycles, so a full decryption takes about 15,000 cycles (only 150 microseconds with a 100MHz clock). This is somewhat better than the fastest reported hardware implementation [SV93], which required about 34,000 cycles on a 40MHz clock, or about 850 microseconds. That implementation uses an $O(n^2)$ algorithm, so our parallel algorithm, which is $O(n^{1.5})$, will have an increasing advantage as key sizes grow to 1024 bits or more.

Since the target machine (described in chapter 4) has no hardware multiply, we used a novel table-lookup scheme for the multiplications in the RNS conversion process. Each 3 bits of x_{w_j} are shifted in by a processor on the edge, combined with an appropriate load instruction, and the result sent out along the row of processors. Each of these processors executes the instruction, which fetches the product of those 3 bits with its (constant) coefficient in an 8-word table. There are only 6 such lookups per multiply, so we have space for a different table for each of the 6 positions, with the product pre-shifted and reduced mod v_i. The final sum for the entire inner product is at most $6 \cdot 16 v_i$, so fully reducing it mod v_i can be done very quickly with a table lookup on the high order bits.

The approximate value of x/w is computed similarly, using tables during the multiplication. Rather than have a complex handler for the exceptional case that the result is too close to an integer to call, we compute the fraction with full precision (down to an error of $1/w$), but propagate the carry without waiting for the lower-order results whenever we can. We thus gain in simplicity and speed, but pay a modest penalty in processors whose computation is rarely needed.

Chapter 7

Document formatting

> We choose to go to the moon in this decade and do the other things not because they are easy, but because they are hard, because that goal will serve to organize and measure the best of our energies and skills...
>
> — John F. Kennedy, speech at Rice University, 1962

For parallel computers to achieve the commodity status enjoyed by Intel-based PCs now, they must deliver computational power that makes the machines appealing to office or home users. The simple fact that graphics can be much faster may be enough to sell machines. But we would also like everyday application programs to be able to benefit from the parallelism.

We chose document formatting as a common application to parallelize. There are certainly many other candidates—spreadsheets, which first started the infiltration of PCs into offices, or databases, which place some of the greatest demands on performance. But the only slow operation for spreadsheets is recalculation, and there is a quite obvious way to parallelize it. And parallel techniques are already well-established for database systems to the point that commercial database systems like Oracle are often run on highly parallel machines like the IBM SP/2. In contrast, document formatting seems at first glance to be highly sequential. In short, we chose it because it was a challenge.

It is also appropriate symbolically—word processing was one of the applications that helped establish personal computers as a commodity, the way typewriters had been before them. To create a parallel word processor is an homage to the ordinary, imagining a future in which parallel computers are as ordinary as personal computers are now.

These were the reasons we chose the problem, but there is one more feature that makes it a particularly good example: the way in which geometry enters into the algorithm in a fundamental way. There is no way to achieve as efficient an algorithm without careful attention to the spatial layout of the data, as we shall see.

Document formatting captures the bulk of the computationally difficult work of a word processor. Operations like global search-and-replace, while they may be time consuming on sequential machines, can be easily parallelized; most other operations are local, and their cost doesn't depend on the size of the document. Once the layout of the characters on the page has been determined, graphical rendering of a document—bit-mapping it for a laser printer, for example—also becomes a local operation, and hence easily parallelizable. Our problem is to decide, given the text and its formatting information, exactly where the words are placed within the lines and pages.

The algorithm we develop for the problem uses a similar divide-and-conquer technique as the classic *parallel prefix* algorithm. Instead of simply adding together numbers to combine the partial results, we must compose functions, represented by a more complex data structure. Sorting is the basis for the function-composition algorithm. We show how, by laying out the data following a space-filling curve, the algorithm's running time can match the lower-bound.

7.1 The Problem

Document layout can be reduced to a very simple mathematical problem. All words and pictures on the page can be represented as rectangles with a particular width and height. Our task is to pack these rectangles horizontally into fixed-width lines, starting a new line whenever the next word would exceed the margin. Then we give each completed line a height, the maximum height of any word on it, and pack the lines onto pages in the same way.

This leaves out many subtleties—changing the margins, hyphenation, justification, footnotes—but while they add complexity, they do not change the mathematical character of the problem. They can be accommodated quite easily by the final algorithm. Other features—paragraph breaks and page breaks—actually make the problem easier, since they provide a reference point in the middle of the document where the horizontal position or the vertical position is known; in any case, they are nearly equivalent to a word as wide as an entire line (for a paragraph break) or a line as tall as a page (for page breaks).

The width and height of the rectangles are typically represented as integers in some device-independent units. We will represent them as real numbers, normalized to the width of the line or the length of the page. In this way, a line needs to be broken if the next word would make its total length more than 1, and similarly for the page.

The line-filling and page-filling problems have the same mathematical structure. We are given a sequence of n items numbered 1 to n, each having a size s_i, $0 < s_i \leq 1$. We wish to pack them into unit-size *bins* (lines or pages) sequentially, and assign them a position within their bin. We define the position inductively as follows:

$$
\begin{aligned}
p_1 &= s_1 \\
p_{i+1} &= \begin{cases} p_i + s_{i+1} & \text{if } \lceil p_i + s_{i+1} \rceil = \lceil p_i \rceil \\ \lceil p_i \rceil + s_{i+1} & \text{otherwise} \end{cases}
\end{aligned}
$$

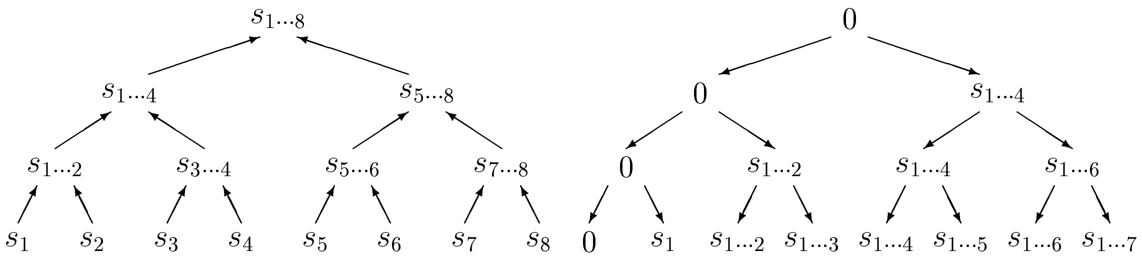
The fractional part of p_i is the ending position of item i within its bin and the integer part is its bin number. We want a fast algorithm to find all the positions in parallel.

On the face of it, this looks quite hard. There is no obvious way to divide up the problem and solve each part separately—we don't know where in a line to begin filling until we have finished laying out the previous lines. It seems as if we have to process the lines one at a time, in order, which would restrict us to a completely sequential algorithm.

7.2 Parallel prefix

If we increase the bin size to the point that there is only one bin, then the recurrence becomes simply $p_{i+1} = p_i + s_i$. This is the classic *prefix sums* problem, for which there are very efficient parallel algorithms. The first parallel circuit for the problem was developed in the early 1960s by Ofman [Ofm63], and many researchers have investigated it and its variants, see Blelloch's article [Ble93] for more history. The problem is in *NC*, and can also be solved in $O(\sqrt[d]{n})$ time on a d-dimensional mesh [Lei92]. The operation in the recurrence need not be addition, but can be any associative operation. Blelloch shows several quite diverse problems that can be framed as a general prefix sum problem, including regular expression searching and lexical analysis [Ble93].

The classic parallel prefix algorithm uses a binary tree to compute the sum of all the items in $\log(n)$ stages. This is the *upward sweep*. Then, in the downward sweep, the sum of all the items to the left of each node is passed down to the node—it can pass this down unchanged to its left subtree, and pass that plus the sum of the left subtree to the right subtree. Figure 7.1 illustrates the upward and downward sweeps.



(a) Upward sweep (b) Downward sweep

Figure 7.1: Parallel prefix algorithm.

It is tempting to try to use the same technique for the layout problem above. We can reformulate the document formatting problem as a recurrence by defining a new operator $\boxplus$, where

$$x \boxplus y = \begin{cases} x + y & \text{if } \lceil x + y \rceil = \lceil x \rceil \\ \lceil x \rceil + y & \text{otherwise} \end{cases}$$

Then we want to find the sequence $p_1, p_2, \ldots, p_n$ defined by

$$\begin{aligned} p_1 &= s_1 \\ p_{i+1} &= p_i \boxplus s_{i+1} \end{aligned}$$

Unfortunately, the operation $\boxplus$ defined this way is not associative, and that is essential for the parallel prefix algorithm. For example

$$\begin{aligned} (.5 \boxplus .4) \boxplus .3 &= 1.3 \\ .5 \boxplus (.4 \boxplus .3) &= 1.7 \neq (.5 \boxplus .4) \boxplus .3 \end{aligned}$$

In parallel prefix, the contribution of any subsequence of the data can be reduced to a single number, the sum of everything in the subsequence. That is not true for the document formatting problem—a subsequence can have a complex effect on the position of later items. Our task is essentially to carry enough information to make $\boxplus$ associative, but a small enough amount that it's still fast to compute $\boxplus$.

To produce an algorithm for the cellular array, we will first develop an *NC* algorithm, with no regard for spatial layout and communication costs, then lay out the data so that it can be implemented efficiently on a cellular array.

7.3 An NC algorithm

In document formatting, the effect of a section of the sequence of s_i on later sections cannot be reduced to a single number as it can in prefix sums. Instead it acts as a complicated function from the position at the beginning of the sequence to the position after the sequence. We can define these position functions precisely. For each item i, let f_i be the function

$$f_i(x) = x \boxplus s_i = \begin{cases} x + s_i & \text{if } \lceil x + s_i \rceil = \lceil x \rceil \\ \lceil x \rceil + s_i & \text{otherwise} \end{cases}$$

Finite compositions of such functions represent the action of a sequence of items. If $i \geq j$, let

$$f_{i,j}(x) = f_i(f_{i-1}(\cdots f_{j+1}(f_j(x))\cdots)) = f_i \circ f_{i-1} \circ \cdots \circ f_{j+1} \circ f_j(x)$$

Then $p_i = f_{i-1,1}(0)$. Function composition is associative, so the sequence may be divided into pieces in any way: if $i \geq j > k$ then $f_{i,k} = f_{i,j} \circ f_{j+1,k}$. It is easy to verify that $f_{i,j}(x+1) = f_{i,j}(x) + 1$ for any position function $f_{i,j}$, so a function is completely specified by its value on the interval $(0, 1]$. Including one but not zero in the interval is slightly unusual, but it fits well with the structure of the position functions.

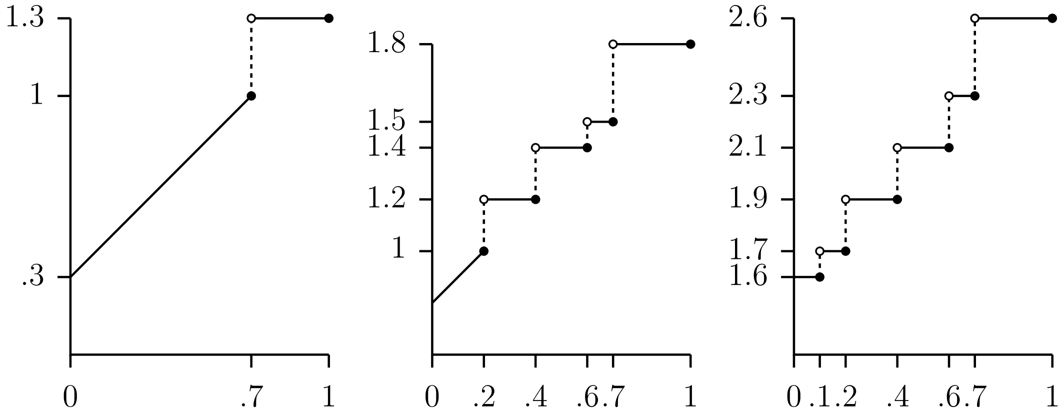
Within that interval, the form of the position function depends on whether the sequence of sizes totals to more than one. Let s be the total of all the sizes in the sequence. If $s \geq 1$ then $f_{i,j}(x)$ will be a monotonically non-decreasing step function, having a constant value on half-open intervals $(a, b]$ (figure 7.2(c) shows an example). If $s < 1$ then an initial interval will have slope 1 ($f_{i,j}(x) = s + x$ for $x \in (0, 1 - s\,]$), after which $f_{i,j}$ will be a step function (see figure 7.2(a) and 7.2(b)).

The overall idea of the parallel algorithm is that a position function can be represented by the list of its constant intervals, which will have a length bounded by the number of items. Composing two position functions can be done by sorting the two lists together, the outer function keyed by the x coordinate where the step occurs, and the inner function (representing the first part of the sequence) keyed by the fractional part of the y coordinate. Sorting is a very efficient parallel operation, and there are only $O(\log n)$ stages of composition needed to find all the position functions.

7.3.1 Representing position functions

We will represent a position function by a base $a_{\min}$ and a sequence of ordered pairs

$$(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$$



(a) A single value (.3) (b) .3, .1, .2, .2 (c) .3, .1, .2, .2, .1, .3, .2, .1

Figure 7.2: Position functions.

where $0 \le a_{\min} \le a_1 < a_2 < \cdots < a_k < 1$. This represents the function

$$f(x) = \begin{cases} x + 1 - a_{\min} & \text{if } x \le a_{\min} \\ b_k - 1 & \text{if } a_{\min} < x \le a_1 \\ b_k & \text{if } x > a_k \\ b_i & \text{if } a_i < x \le a_{i+1} \end{cases}$$

when $0 < x \le 1$. For values outside that range, f can be computed by $f(x) = f(\mathrm{fp}(x)) + \mathrm{ip}(x)$ where

$$\begin{aligned} \mathrm{ip}(x) &= \lceil x \rceil - 1 \\ \mathrm{fp}(x) &= x - \mathrm{ip}(x) \end{aligned}$$

Note that $x = \mathrm{ip}(x) + \mathrm{fp}(x)$, $\mathrm{ip}(x) \in \mathbb{Z}$ and $\mathrm{fp}(x) \in (0, 1]$.

There is some redundancy in this representation, since $a_{\min}$ is either zero (if there is no region of slope 1) or is equal to a_1.

The position function in figure 7.2(b) can be represented by $a_{\min} = .2$ and the ordered pairs

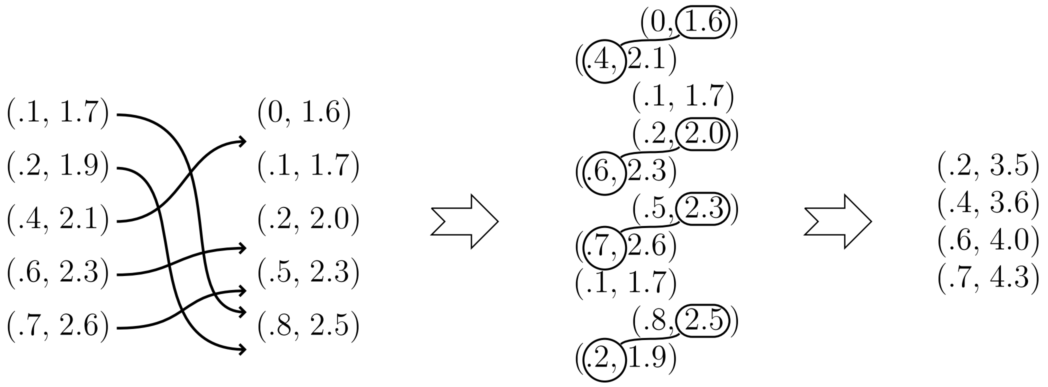
$$(.2, 1.2), (.4, 1.4), (.6, 1.5), (.7, 1.8)$$

while that in figure 7.2(c) would have $a_{\min} = 0$ and the pairs

$$(.1, 1.7), (.2, 1.9), (.4, 2.1), (.6, 2.3), (.7, 2.6).$$

7.3.2 Composing position functions

If f and g are two position functions represented by $a_{\min}, (a_1, b_1), \ldots, (a_m, b_m)$ and $c_{\min}, (c_1, d_1), \ldots, (c_n, d_n)$ respectively, we want a fast algorithm to find the list of pairs representing their composition $g \circ f$. In the usual case, when f represents a sequence of total size greater than 1, the composition can be represented by the list $(a_i, g(b_i))$. Translating each b_i value by g is also straightforward, since it just requires finding the point in the (c_j, d_j) sequence for which $c_j < \mathrm{fp}(b_i) \leq c_{j+1}$. This can be accomplished by sorting the two lists together. Figure 7.3(a) shows an



(a) Sort (b) Find transitions (c) Combine

Figure 7.3: Composing two position functions.

example of this process. The appropriate c_j value is the last preceding pair from g in the list. That pair may not be immediately adjacent in the list, but in that case a_{i-1} will also share the same output value (i.e. $g(b_{i-1}) = g(b_i)$), so we can simply omit a_i from the output function—only where a pair from f immediately follows a pair from g will we need to create a pair for the composition. This is illustrated in figure 7.3(b) and (c). The test for the immediately preceding pair in the sorted list must "wrap around" from the start of the list to the end, because $g(x) = d_n - 1$ if $c_{\min} < x \leq c_1$.

There are some subtleties to deal with if one or both of f and g start with a section of slope 1, as in figure 7.2(a) or (b). If $a_{\min} > 0$ then there is an interval in which f is not a step function, and $g \circ f$ will need more pairs in its representation than f had. When g has a pair (c_j, d_j) with $c_j > 1 - a_{\min}$, then there is a point in the region where f is varying continuously that maps onto c_j. This means a new step must be introduced at $c_j - 1 + a_{\min}$.

Algorithm Compose

Input: base $a_{\min}$ and pairs (a_i, b_i), $1 \leq i \leq m$ representing f
base $c_{\min}$ and pairs (c_i, d_i), $1 \leq i \leq n$ representing g
Output: base $p_{\min}$ and pairs (p_i, q_i), $1 \leq i \leq l$ representing $h = g \circ f$

Sort the (a_i, b_i) pairs keyed by $\mathrm{fp}(b_i)$ together with the (c_j, d_j) keyed by c_j
Broadcast $a_{\min}$, $c_{\min}$, and the first pair in sorted order
forall pairs (c_j, d_j)
    if (c_j, d_j) immediately followed by (a_i, b_i) **then**
        add $(a_i, \mathrm{ip}(b_i) + d_j)$ to h
    end if
    if (c_j, d_j) is last pair and (a_i, b_i) is first **then**
        add $(a_i, \mathrm{ip}(b_i) - 1 + d_j)$ to h (wraparound)
    end if
    if $c_j > 1 - a_{\min}$ **then**
        add $(c_j - 1 + a_{\min}, d_j)$ to h
    end if
end forall
forall pairs (a_i, b_i)
    if $\mathrm{fp}(b_i) \leq c_{\min}$ **then**
        add $(a_i, b_i + 1 - c_{\min})$ to h
    end if
end forall
$p_{\min} := \max(0, a_{\min} + c_{\min} - 1)$
Sort the pairs (p_k, q_k) of h into ascending order by p_k

Note that the last step (sorting the pairs) can usually be omitted since the next iteration will be another composition, which will start by sorting the pairs again.

7.3.3 Upward and downward sweeps

Like parallel prefix, the document-layout algorithm requires that we first propagate results up the tree to find the large-scale effect of a block, then send the information back down the tree to get the detailed position information. At stage k of the upward sweep, pairs of blocks of size 2^k are combined by the composition algorithm above to make blocks of size 2^{k+1}. The upward sweep for 8 items is shown in figure 7.4. Stage k of the downward sweep (with k running from $\log(n)$ to 1) starts knowing the starting position of each block of size 2^k and computes the position of each block of size 2^{k-1}. The starting position of the left subtree is the same as the starting position of the parent, and the starting position of the right subtree is found by applying the left subtree's position function to its starting position. This is shown in figure 7.5.

Since we need to use the position functions of the smaller aggregates during the downward sweep, we need to keep them around, which would require an additional

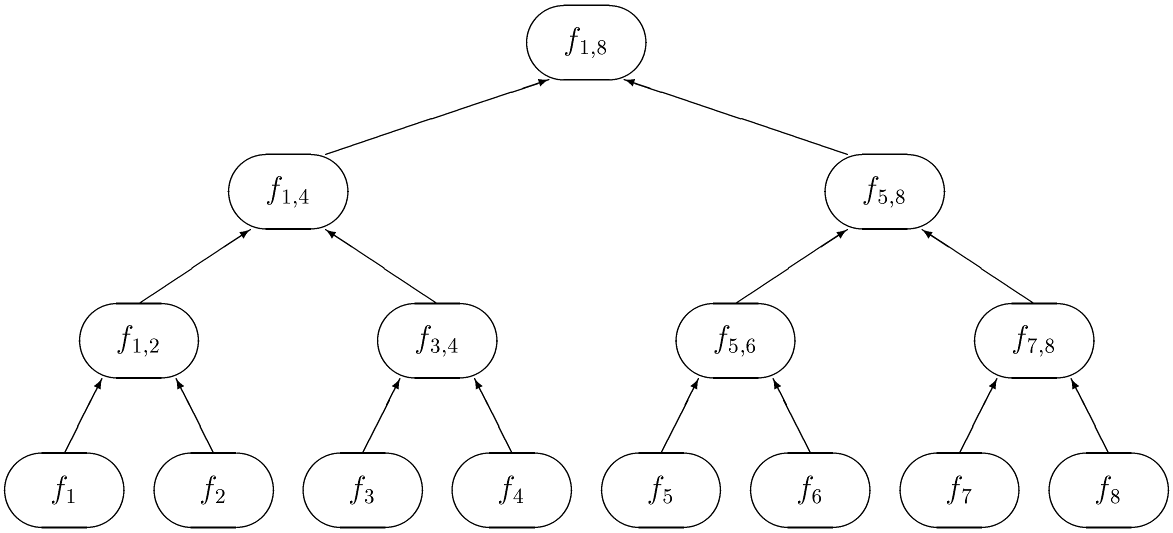


Figure 7.4: Upward sweep: merging position functions.

$O(n \log n)$ storage. We would like to avoid this in general, but particularly when we move to the CA, where the increased storage would slow down the algorithm by a factor of $\sqrt[d]{\log n}$. Fortunately, we can extract the position functions of the two subtrees from the combined function as long as we keep the original index of each component. The cost of extracting them is no more than the cost of combining them—one sort.

Each of the $\log n$ phases of the upward sweep requires a sort and some local computation. The cost of a phase is therefore $O(\log n)$ on a PRAM. Each phase of the downward sweep requires applying the position function, which can be performed in $O(\log n)$ time by binary search, and re-extracting the lower-level position function, which can also be done in $O(\log n)$ time using sorting. There are $\lceil \log n \rceil$ phases, so the total time required is $O(\log^2 n)$, using $O(n)$ processors. This shows that the document layout problem is in *NC*.

7.4 Embedding the algorithm on a CA

The *NC* algorithm above requires nothing but sorting and broadcasting, both of which can be performed in $O(\sqrt[d]{n})$ time on a d-dimensional CA, and some local communication, which requires constant time. Since these operations are done only once in each phase, and there are $O(\log(n))$ phases, the total running time of the algorithm will be $O(\log(n) \sqrt[d]{n})$. A lower bound for any algorithm is $\Omega(\sqrt[d]{n})$ time, since the position of a single word may be dependent on every input item occurring

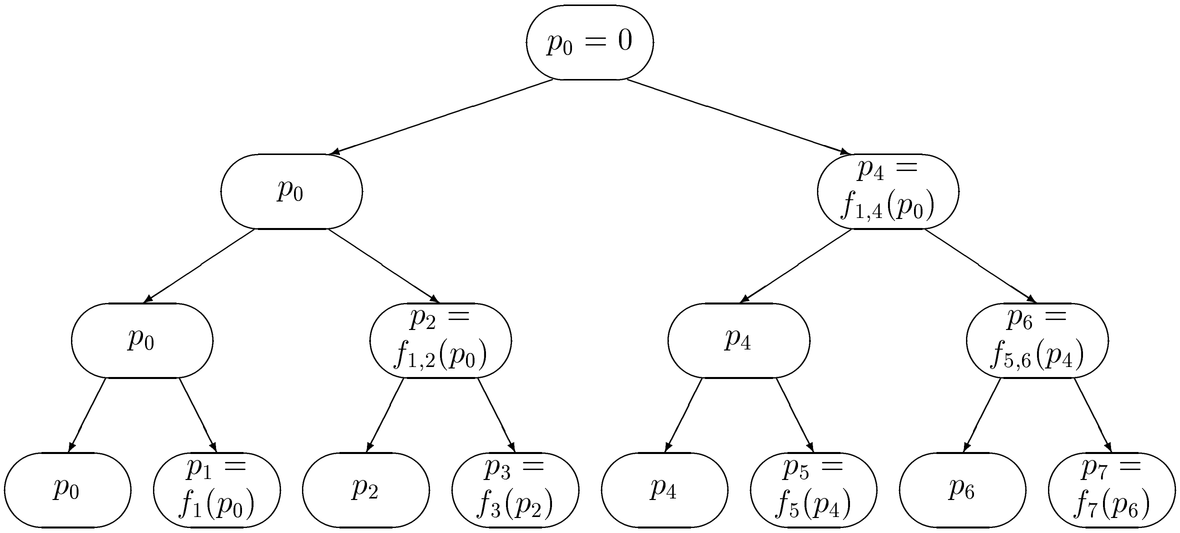


Figure 7.5: Downward sweep: computing positions.

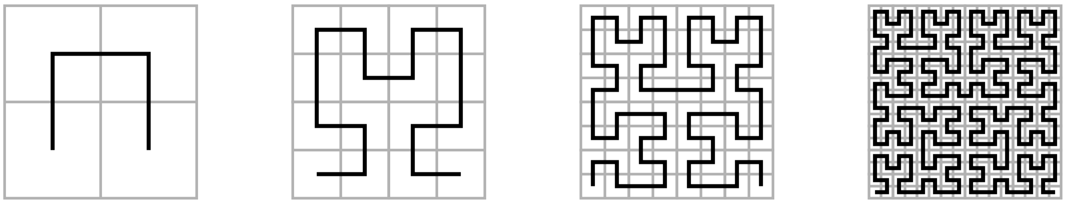
before it. Can we eliminate the $\log(n)$ factor, and achieve the lower bound? The embedding we just sketched for the CA paid no attention to data layout. If we are careful about how the data are arranged in space, we can solve the problem in $O(\sqrt[d]{n})$ time.

In the upward sweep, the data sets which are being merged start small, then double (or potentially double) each iteration until all the data may be involved. In the downward sweep, the pattern is reversed. The key to a fast algorithm on a CA is to make the small data sets go faster—if every composition of k items can be done in $O(\sqrt[d]{k})$ time, then the total running time of the algorithm will be

$$O(\sqrt[d]{n} + \sqrt[d]{n/2} + \sqrt[d]{n/4} + \cdots + \sqrt[d]{1}) = O(\sqrt[d]{n}).$$

7.4.1 Hilbert curves

The only requirement for keeping the compositions fast is that the k pairs being composed lie within a neighborhood of radius $O(\sqrt[d]{k})$. The standard orderings of data on a mesh, row-major order or snake-like row-major order, don't have this property. Hilbert's space-filling curve, or more precisely the n-th order approximations to the Hilbert curve [Hil91, Sag94] do, however. Figure 7.6 shows Hilbert curves of orders 1–4. Notice that each of these curves completely traverses the left half before entering the right half, completely traverses the bottom left quadrant before the top left, and so forth. This property gives us the necessary locality to do fast compositions.



(a) First order (b) Second order (c) Third order (d) Fourth order

Figure 7.6: First four approximations to the Hilbert curve.

One more detail: the usual recursive description of the Hilbert curve is a method for generating an *entire* order n curve from four copies of the curve of order $n-1$. In order to efficiently arrange the data into Hilbert curve order, or to sort them into this order, we must be able to locally map each item to its place in space. Given an integer i, we want to efficiently compute the (x, y) coordinates of the ith point along the Hilbert curve. The following simple algorithm does so:

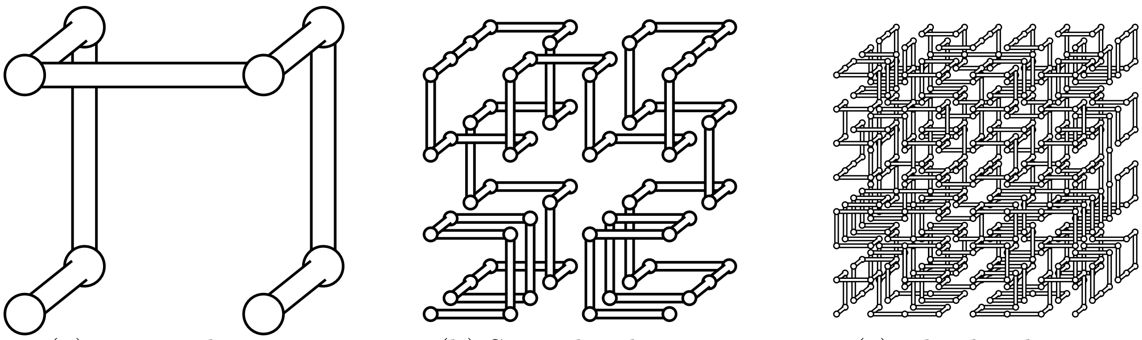
Input: n, the order of the Hilbert curve
$i = 2^{2n-1}b_{2n-1} + 2^{2n-2}b_{2n-2} + \cdots + 2b_1 + b_0$, the index and its bitstring representation
Output: (x, y), the coordinates of the point at index i

$(x, y) := (0, 0)$
for $j := 0$ **to** $n-1$
  $q := 2b_{2j+1} + b_{2j}$
  if $q = 0$ **then** $(x, y) := (y, x)$
  if $q = 1$ **then** $(x, y) := (x, y + 2^j)$
  if $q = 2$ **then** $(x, y) := (x + 2^j, y + 2^j)$
  if $q = 3$ **then** $(x, y) := (2^{j+1} - 1 - y, 2^j - 1 - x)$
end for

We can also sort in the Hilbert curve indexing just as efficiently as we can sort into row-major indexing. Sibeyn [Sib94] has shown how to modify mesh sorting algorithms for any index that has sufficient locality, in the sense that for most elements, the immediate successor of that element is adjacent in the array.

7.4.2 Hilbert curves in 3 dimensions

The Hilbert curve can also be generalized to 3 or more dimensions, which we need to achieve the optimal running time on a 3-dimensional CA. The generalization was independently discovered by several researchers [But69, Gil84, Jag90], each apparently unaware of the others' work.



(a) First order (b) Second order (c) Third order

Figure 7.7: A three-dimensional Hilbert curve.

Unlike the 2-dimensional Hilbert curve, many variations are possible in three dimensions. The first order curve must follow a 3-bit Gray code, and all such curves are equivalent except for rotations and reflections. For higher-order curves, each of the 8 points of the first-order curve are replaced by a lower-order curve whose orientation is not fully determined. In general, there will be two possibilities for the orientation, giving 256 possible curves. We chose one on the basis of symmetry and similarity to the 2-dimensional curve. It is illustrated in figure 7.7. This curve can be generated by the following mapping from an index to its (x, y, z) coordinates:

Input: n, the order of the Hilbert curve
$i = 2^{3n-1}b_{3n-1} + 2^{3n-2}b_{3n-2} + \cdots + 2b_1 + b_0$, the index and its bitstring representation
Output: (x, y, z), the coordinates of the point at index i

$(x, y, z) := (0, 0, 0)$
for $j := 0$ **to** $n - 1$
  $q := 4b_{3j+2} + 2b_{3j+1} + b_{3j}$
  if $q = 0$ **then** $(x, y, z) := (z, y, x)$
  if $q = 1$ **then** $(x, y, z) := (y, x, z + 2^j)$
  if $q = 2$ **then** $(x, y, z) := (y, x + 2^j, z + 2^j)$
  if $q = 3$ **then** $(x, y, z) := (x, 2^{j+1} - 1 - y, 2^j - 1 - z)$
  if $q = 4$ **then** $(x, y, z) := (x + 2^j, 2^{j+1} - 1 - y, 2^j - 1 - z)$
  if $q = 5$ **then** $(x, y, z) := (2^{j+1} - 1 - y, 2^{j+1} - 1 - x, z + 2^j)$
  if $q = 6$ **then** $(x, y, z) := (2^{j+1} - 1 - y, 2^j - 1 - x, z + 2^j)$
  if $q = 7$ **then** $(x, y, z) := (2^{j+1} - 1 - z, y, 2^j - 1 - x)$
end for

The 3-dimensional curve just described, like its 2-dimensional analogue, completely traverses one octant before entering the next one, and the same property applies recursively to the section of the curve within the octant. Sorting two regions together each of which contain k elements therefore requires $O(\sqrt[3]{n})$ time, and the entire document layout algorithm takes time proportional to

$$\begin{aligned}\sqrt[3]{n}+\sqrt[3]{n/2}+\sqrt[3]{n/4}+\cdots+\sqrt[3]{1} &< \sqrt[3]{n}(1+\sqrt[3]{1/2}+\sqrt[3]{1/4}+\sqrt[3]{1/8}+\cdots) \\ &= \sqrt[3]{n}(2+2\sqrt[3]{1/2}+2\sqrt[3]{1/4}) \\ &< 5\sqrt[3]{n}.\end{aligned}$$

Hilbert curves, in 2 or more dimensions, do not seem to have been used before in parallel algorithms, but they have been used extensively in data layout for sequential algorithms. Generally, they have been used for laying out 2 or 3-dimensional data, like an image, in a 1-dimensional storage medium like a disk or RAM [Sam90, Jag90]. We use them for the opposite purpose: laying out 1-dimensional data in a 2 or 3-dimensional array.

7.5 Other algorithms

The algorithm we just presented has the optimal worst-case running time, which is the usual standard by which algorithms are judged in computer science. But from an engineering point of view, there are some simple ad hoc algorithms which would work well enough in practice. They are based on the observation that normal documents are unlikely to be too complex.

In particular, the algorithm we presented can handle very non-local effects, in which a word inserted on the first line can affect the position of words on other lines throughout the entire document. But this can only happen if there are no paragraph breaks anywhere in the document. If we assume regular paragraph breaks, the line filling problem can be solved independently on each paragraph. Similarly, very long documents are likely to have designated page breaks—for chapters, sections, or figures—fairly regularly. A 200-page monolithic document with no forced page breaks would be exceedingly rare. Again, a simple algorithm could take advantage of those page breaks and do page filling for each chapter separately in parallel. Such a technique would be good enough for most purposes. This is the approach we were going to take before discovering the NC algorithm and its Hilbert-curve embedding.

Chapter 8

Operating Systems

Operating systems—Relics of the past?

— Bob Frankston, title of talk, 1995

In order to become acceptable as commodity computers, CAs must not only run everyday applications, they must also support everyday operating environments. This is not to say that must run Windows NT or Unix, but they must provide a robust and comfortable environment for the user to interact with these programs and for the programs to interact with each other.

Since in some ways operating systems act as giant subroutine libraries, we must expect that the structure of the operating system will have to change to reflect the fact that the very notion of a subroutine was tied to sequential machines, and quite different programming constructs are more natural on the cellular array.

We are fortunate however that the change in world-view required is not very great, and was already well underway in the operating system community with the emergence of client-server systems and message-based transactions instead of system calls. These concepts are already woven into the design of Windows NT, for example, and are very well developed in Plan 9, the new operating system from the Unix developers at Bell Labs.

One other conceptual shift is necessary to design an operating system for a CA: instead of the operating system being thought of as the kernel, it must be thought of as the *periphery*. It sits on the boundary of each program and mediates its interactions with the world, but only through communication. In the old paradigm, the kernel was like an extension of the machine, providing extra instructions and capabilities. Extending the instruction set with floating point operations or supporting virtual memory, for example, fit very nicely into that model. In a cellular array, there is not enough room on each machine to support virtual memory or an instruction interpreter. Using external communication to access operating system functionality has benefits in security, efficiency and elegance.

One operating system problem that is significantly harder on a CA than a conventional machine is allocation of space. In particular, the three-dimensional structure of the array makes packing that much harder than in a one-dimensional memory. Moreover, since the 1970s for mainframes and mid-1980s for microprocessors most computers have supported page tables, which relieved the operating system from allocating a process' memory contiguously. In the CA, the interconnections reflect physical proximity and cannot be remapped by software, which makes allocation harder.

In this chapter, we will sketch the design of an operating system for the cellular array based on the client-server model. We will present the results of some experiments with space allocation in 3 dimensions, showing that the problem is not intractable, and we will explore some of the substantial security benefits of making the OS the periphery.

8.1 The OS as the periphery

High-performance operating systems over the past decade or so have shown a trend towards smaller, more efficient kernels. Part of the motivation is greater reliability—bugs in the kernel code affect all processes, but if some of the functionality can be moved into user-mode processes, the effect of bugs is contained. Part is efficiency—a small, light-weight kernel can be coded more tightly, and thus shorten the length of the critical path for all I/O functions.

Other researchers, noting that in gigabit networks most of the delay comes from operating system overhead, find ways of bypassing the OS entirely for certain I/O operations. The U-Net project, for instance, allows user processes direct access to the network interface, but maintains safety by respecting memory protection in the interface card [vEBBV95].

These two trends intersect at a point where the operating system vanishes from the local machine. This is the fundamental idea behind the operating system presented here. A single cell of the array is too small to support any meaningful operating system, and a rigorous application of the principle of no unnecessary logic in each cell applies to software as well (since software takes memory space, which requires physical space to implement).

The keys to making this possible are the "boot" bits, described in section 4.3.5. They provide a way to forcibly reload a processor or group of processors without requiring any supervisor mode. Since the immediate neighbors in two canonical directions must collaborate to reload a cell, there is no danger of a malicious reboot signal propagating out of a rectangle enclosing the misbehaving cells. The operating system will give processes rectangular regions of the array and will continue to control cells on the boundary of that region. In this way, any other process will be insulated from malicious action, but the operating system will maintain the ability to forcibly terminate any process.

Avoiding any supervisor mode bits in the cells lessens the distinction between operating-system level code and user code. Virtual operating systems, able to run other operating systems below them as user programs, are very easy to implement on the cellular array. Perhaps more important, the distinction between pieces of the operating system and user programs is slight. In current operating systems, like Microsoft Windows 95 and Windows NT, more and more code must be written in protected mode, often by third-party suppliers, to support special I/O devices. This threatens to make the systems much less reliable, and it also places a large burden on developers. Since the code runs in such a different environment from normal programs, many software tools and libraries are unavailable, and debugging is much harder [DC97]. All these problems are artifacts of having supervisor mode, and we can neatly avoid them by keeping the operating system physically separated from application code.

When starting a process and ending (or suspending) a process, the operating system will use the boot bits. In between, the only interaction between the OS and the process will be communication at the boundary between them. We will now discuss the problem of turning all the operating system's functionality into communication.

8.2 The client/server model

The Internet is a good example of a very large and complex system with no overall operating system that provides a dizzying range of functionality. It makes one wonder why we need operating systems at all. In fact, the operating system we describe resembles the Internet's unenforced cooperation. Network services are accessed only through communications—clients (like web browsers) send request messages to servers (web sites) and receive the requested information back. Most operating system functions can be implemented by exchanging messages with servers.

This model is called the *client/server* model, and several recent operating systems have been based on it, perhaps most notably Microsoft's Windows NT [Cus93]. In a typical client/server system, a simple communication facility is built into the kernel, and most requests are simply passed along from one user-mode process to another (from the client to a server or vice-versa). Since these messages can easily be forwarded through a network, client/server is a very natural model for distributed systems. Services involving shared resources are particularly well suited to being implemented by servers, for example the file system, the network (with its protocol drivers) and the display (with its graphical user interface).

Traditionally, I/O drivers have *not* been implemented as servers, because user processes (for reasons of security) are not allowed direct access to the I/O devices. In a cellular array, we envision that the outer boundary of the array would be dedicated to I/O devices. Different sections of the boundary would be connected with different devices. In keeping with the goal of reducing special-purpose hardware, we want the array to take over as many of the functions of the interface logic as is feasible (using the array's parallelism instead of a digital signal-processing chip for driving a modem, for example). This is incompatible with multiplexing several I/O devices on a single boundary cell. The I/O driver can then reside in a small region of the array immediately adjacent to that section of the boundary that controls its device. Aside from their location, I/O drivers can then behave just like other servers.

8.2.1 Plan 9

Plan 9 is the newest operating system from a team at Bell Labs that includes some of the original Unix developers. Though it bears a superficial resemblance to Unix, the design is all new, based on three interrelated ideas: first, the intuitive, hierarchical name space commonly used for file systems has been extended to all system resources; second, these objects respond to read and write operations using a standard protocol; third, the name space visible to a program is dynamic and independent, so the same name may refer to different objects in different programs. Because the resources are accessed by reading and writing files (i.e. by passing messages), the services may not be local. It will not matter where a program is run except from the standpoint of communication overhead, since all the same resources can be accessed the same way even if the requests go through a network.

For our purposes, the Plan 9 design offers the clearest working example of how operating system interaction can be turned into communication. The motivation in Plan 9 was to allow the overall system to be completely distributed, "to build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes" [PPD+95]. Presenting a unified face to a decentralized system is a wonderful feature, and will certainly become important to CAs if they achieve commodity status, but for us the major attraction is that the Plan 9 design moves almost all the functionality out of the kernel.

In Plan 9, all the functionality outside of the kernel can be provided remotely through the network, and all this functionality can be provided on the CA through communication with the OS at the boundary of the program's region. This includes file systems, networks, devices like the mouse and keyboard, and the windowing system. We will now examine the remaining functions, those which *are* implemented in the kernel in Plan 9, to see how to provide them on the CA.

- Read and write. These operations are essentially provided directly by the hardware. Designated sites on the boundary of a process serve as channels to the OS. These channels act like network interfaces—any of them can accept data or control information for any logical channels the process has open.

- Open, create, and duplicate. These are the operations which provide the file handles over which the communication happens in Plan 9. They can be coded as special messages sent along the OS channels.

- Device interfaces. Except for their location at the very edge of the physical array, device drivers can look just like any other servers on the CA. The only special operating system support needed is a way to specify certain programs that must be loaded at certain locations at boot time.

- Process management. Much of process management is an attempt to simulate parallel execution on a sequential machine; thus primitives like "suspend" or "fork" are not necessary, or even meaningful. Executing another program (in a new space, unlike fork) is meaningful, and must become a message sent to the space allocator of the operating system. Exiting (the calling process) or terminating (another process) are requests for the OS to forcibly bootload on some running process.

- Paging and memory management. Instead of memory management, we have space management, a multi-dimensional version of the problem which is much more challenging. The concept of virtual memory, in particular, doesn't translate well to cellular arrays, which is a pity.

8.3 Communication

Fast communication is essential to an operating system for a very high-performance parallel machine. Since interprocess communication is the operating system's main job, it should be able to do an extremely good job. But how exactly?

The first observation we must make about interprocess communication is that although the throughput can be excellent, we cannot expect the latency to be low. If we must use a 2-dimensional array (a reasonable assumption, with current technology), then a million-processor array would be 1000×1000, and it may happen that a client and its server are near opposite corners, so any message between them will have to be forwarded nearly 2000 times to reach the destination. A round-trip would double that.

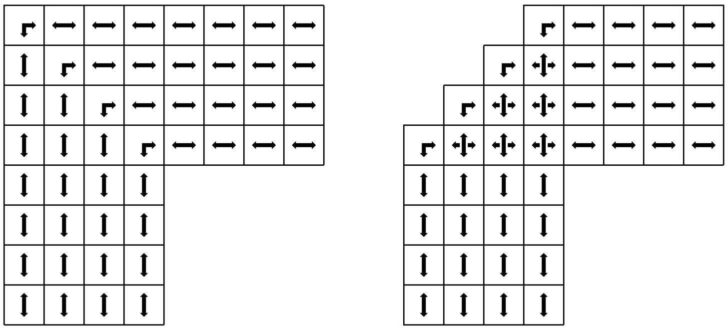
If we were to use a store-and-forward routing algorithm, the time would be prohibitive. Instead, when a path is established, each word is passed along the path immediately after it is received, like a bucket brigade. If the length of the path is l and the number of words in the message is s, the total time for the message to reach its destination is $l + s$ rather than ls for a store-and-forward system. This is often called *wormhole routing*, though the term technically means that the message starts even before the route is established. We will use the term *wormhole transport* for the method of sending the message independent of setting up the route.

In a planar array, communication paths between clients and servers cannot always be kept from crossing. The cell architecture we have chosen allows paths to cross through a cell without any degradation in performance by using a full crossbar switch. Without this feature, the wormhole transport would often have to block, and the entire communication mechanism would be much more prone to congestion.

Note that even the round-trip time of 4000 cycles is not prohibitive. At 200MHz, this is only about 20 microseconds, which is less than the overhead of making a system call on many current operating systems. The bandwidth available on the channel, assuming only one word is sent per clock cycle, is 800Mb/s. Since this is quite speedy compared to any I/O devices, a word-wide channel would be sufficient at the present. But we are trying to envision a computer with vastly greater throughput and processing power than current machines, so we need to think bigger.

Bigger, in this case, means wider. Wider data paths can carry more data in parallel and thus increase the bandwidth. The latency does not improve correspondingly. Wide data paths can be implemented by several parallel paths. Unfortunately, this opens the question of standardization—if there is more than one standard width for a communication interface, then how can we maintain interoperability? The problem is analogous to the different bus structures on the IBM PC. From the original 8-bit bus, to the 16-bit ISA bus, to the 32-bit PCI bus (and beyond), more I/O bandwidth has come through wider interfaces, and compatibility has suffered as a result. There is a software cure in the case of the CA operating system—we can define a standard way to multiplex and demultiplex from wide to narrow and vice-versa, as long as we restrict the widths of the data paths to powers of 2. This requires that the OS provide buffering, but since the data paths are cells with their own memory, there is quite a bit of buffer capacity already available.

Wide data paths introduce problems of signal timing. If the different words of the same signal don't travel the same length from the source to the destination, then the signals will be skewed in ways which depend on the actual path taken. This would necessitate complex buffering at the receiving end, which could neutralize the benefit of the wide data paths. The straightforward way of routing a signal round a corner shown in figure 8.1(a) has this problem. There is another way to route wide signals which doesn't change the timing. It is analogous to taking a piece of ribbon cable (in which all the wires are the same length) and folding it where necessary to make right-angle turns. The folded corner is shown in figure 8.1(b). It depends on the cell's cross-through routing capability. A 3-dimensional version of the folded corner can be built by stacking several layers of the 2-dimensional corner.



(a) Mitered corner (b) Folded corner

Figure 8.1: Routing wide signals around corners.

8.4 Resource allocation

Most resource allocation problems seem independent of the computer architecture we use. Disk allocation strategies are well understood, and we should be able to import the solutions directly to the CA. Allocation of network bandwidth is not perfectly understood in the presence of real-time data which needs quality-of-service guarantees, but any solutions should be readily implementable on a CA. The only area in which things are more challenging is in allocation of space in the processor array.

This problem is analogous to the memory allocation problem in a standard sequential machine: to provide facilities for allocation and deallocation of contiguous regions of memory on a dynamic basis as need arises, and to make efficient use of the memory.

Unlike memory allocation, however, in the cellular array the regions to be allocated and freed are multi-dimensional. The one-dimensional problem has been well studied [Knu91b, EGCL89], and quite efficient use of the space can be made, as long as the blocks being allocated are reasonably small compared with the total space. In two or more dimensions, however, the problem appears much harder. The bin-packing problem (also known as “stock-cutting” in the operations research community) is similar, but without deallocation, and with all the block sizes known in advance. Heuristic bin-packing algorithms have been used to allocate space in mesh processors for numerical simulations, but again the sizes are all known in advance and there is no deallocation [VH94]. Some theoretical work has been done in on-line packing, where the block sizes aren’t known in advance, and an on-line algorithm is known which uses within a constant factor of the best off-line algorithm [CvV93].

Apparently the only research done on the dynamic problem (i.e. in which blocks can also be deallocated) was by Coffman and Gilbert in the mid 1980s [CG84]. They investigated the problem of dynamically allocating and deallocating square regions in a rectangular space. They showed that if the squares could range up to $k \times k$ in size, then any first-fit algorithm might utilize only $O(1/\log(k))$ of the total space. They also gave a simple first-fit algorithm which matched this bound. They generalized their bounds to 3 or more dimensions, for which only the constant factor changed.

Our problem is slightly different than the one they investigated in that our regions are not necessarily square (or cubical). However, as a general rule they will be reasonably close to square, since elongated regions have long end-to-end communication time while getting no other benefits in return. Also, we are interested in average behavior, not worst-case. In the one-dimensional problem, dynamic storage allocation, all algorithms have extremely bad worst-case behavior, but many perform quite well in practice.

8.4.1 Experimenting with a simple allocator

In order to test whether allocation is at all feasible for the 3-dimensional array, we implemented a simple dynamic storage allocator and tested it under a simulated stream of requests. The simulations were in the style of Knuth’s simulations of dynamic memory allocation [Knu91b]. The algorithm we used is a somewhat naive best-fit system. The available free blocks are scanned in order of increasing *volume*. When a block is allocated from a larger block, the remaining volume is not rectangular. Since we keep a free list of rectangular blocks, this requires an arbitrary division of the remaining volume into 3 smaller rectangular blocks. We used the heuristic of subdividing first in the dimension with the largest remainder. In this way, we hoped to keep the free blocks as close to cubical as we could.

We simulated allocation on a $512 \times 512 \times 512$ array (this represents a 100 million processor system, which is large enough that the request sizes act like real numbers, not integers; smaller arrays result in less wasted space). The requests were for random-sized blocks, reasonably close to cubical (the ratio of the shortest edge to the longest was no more than 2). The longest edge was distributed logarithmically from 8 to 128 ($\frac{1}{4}$ the width of the total array). The residence times of each block were exponentially distributed with a mean of 1, and the arrival rate was varied to adjust the number of blocks and the storage density.

In the simulations, the maximum amount of the array that could be utilized without any requests being refused was about 30% (it varied from 28.5% to 31.6% in different simulations). This level of utilization was very stable, continuing for up to 150,000 allocations, so fragmentation was not steadily increasing. Allocating completely cubical blocks only increased this utilization by a few percent.

8.4.2 Wasted space

The level of utilization we achieved with the simple allocator described above may seem low—in memory allocation, 90% is more to be expected. The tests at least show that allocation is not going to be impossible, but it also raises the question of what to do with underutilized 70% of the array.

Part of the blame may belong to the simple allocator—there are several obvious improvements to try. First, the best-fit search could try to match one or two of the block's dimensions as closely as possible, and so try to lose less in irregularly-shaped remainders. Second, the irregular shapes need not be subdivided to put on the free list. Doing so made the programming easier, but if the irregular remainder were maintained as a unit, there would be more flexibility in subdividing it later. On the real array, there will be enough processing power in the free space of the array to perform quite complex searches, so the geometry need not be simplified for the sake of the program. Third, in the array the three axes are more or less equivalent, so a non-cubical request could conceivably be rotated to better match the available free blocks. This would require a quite sophisticated loader, however, which had the capability to re arrange the instruction words according to the rotation.

If we allow some allocations to fail and let the demand increase, the utilization of space goes up to around 60% In a real system, there will certainly be "greedy" requests that we simply can't satisfy. The real question is how to manage a greater demand for array space and time than we can satisfy. It is a familiar problem for a sequential machine—the very basis of processor scheduling and virtual memory is overcommitment. We need a way to deal with overcommitment of resources on the CA, which is the subject of the next section.

8.4.3 Quality of service

If at some time there is too much demand for space in the array, then we must somewhat degrade the performance of the programs that are running on it. But since we have no operating system running on each of the cells that make up a program, we have no convenient way to interrupt the program and capture its state. Capturing an instantaneous state is, in any case, exceptionally hard for a parallel program. So preempting a running program is not an option. Without preemption, however, the scheduling may not be fair—a process may have to wait an extremely long time for its chance to run.

The solution most in keeping with the overall philosophy (never do in hardware what can be done in software) is for programs to make contracts with the operating system for their time in the array. They may allocate large chunks to run in, but for a comparatively short time. After that time they must be ready to save their state and voluntarily terminate, or be forcibly terminated. They may continue running if no other process needs the space, but otherwise they will wait their next turn. The state-saving code would be very tedious to write into every program, but it would not be a burden if it were automatically generated by a compiler. If space could not be allocated for a program entering or re-entering the array, there could also be a choice of settling for one or more smaller fragments. Many algorithms would work reasonably well given smaller regions than they would optimally run in. The document formatting problem described in chapter 7, for example, would simply take 4 times as long on a region that could only hold 1/4 of the document—still much faster than sequential.

What we have just described could be called "best-effort" execution, and is similar to best-effort delivery of packets on the Internet. It is also the way almost all sequential operating systems work. Not all programs can operate correctly with no performance guarantees. Device drivers, signal processors, servers, and real-time video or audio players are among the programs that would likely demand quality-of-service guarantees. Quality of service guarantees originally come from similar applications of the Internet and the difficulty of implementing them with standard IP.

The CA is in a unique position to be able to offer quality-of-service guarantees. Long-term allocation might take longer to satisfy, or might be restricted to a smaller size, but nothing would then prevent the program from getting all the cycles it needed. Of course, the same guarantee would somehow have to be extended to the network for an application like real-time video; many researchers are working on adding quality-of-service to networks, often ignoring the need for such guarantees in the operating system.

8.5 Security

Security is one of the strongest areas of cellular array operating systems. The operating system as the periphery provides a physical barrier between a user program and the rest of the array. This quite naturally gives a level of security matching the most carefully engineered operating systems for uniprocessors. In some respects, the security level can be higher, since it is almost impossible in a conventional operating system to solve the confinement problem: using a subroutine on secure data without allowing it to leak any information to a third party. Butler Lampson examined the problem in [Lam73] and showed the near impossibility of eliminating covert channels, such as affecting the response time of the system. In a cellular array, all interaction with other parties can be entirely eliminated or carefully mediated, much like a network firewall mediates between local subnetworks and the Internet.

On a conventional microprocessor, the protection system imposes a high performance penalty for context switches and I/O operations due to the complex privilege modes and address translation. Since the goals of security and performance aren't completely compatible, many high-performance applications bypass the operating system completely or rely on modifications to the OS. In contrast, the security system on the CA adds essentially no overhead. Together with the extremely high performance of the CA, this means that the actual level of security should be substantially higher than what is common now.

8.5.1 Applications for security

There are some interesting applications for the increased security possible on the cellular array: specifically, the ability to run an untrusted subroutine securely. A *software agent* is a piece of code sent from one program (program A) to another (program B) with which it is interacting. The agent can then run locally as a subroutine of B and perform several interactions at lower cost than if A were sending a message for each interaction. Software agents are typically used over a network, where the messages have much higher overhead than they do on a single machine, though there are other uses.

The most familiar current use of software agents is in web browsers, in Java and Active X. They are also the key idea in *active messages*, which allow for faster network protocols by putting code bursts in the message headers [vECGS92]. The same idea has been used within a single machine to perform several I/O operations without repeatedly incurring the high startup cost. In the 1960s and 1970s, large mainframes typically had I/O processors which could execute chains of commands for such purposes as scatter/gather I/O. More recently, the same idea has been used to pass small programs to the operating system rather than making a series of requests. The motivation is that while throughput is increasing at about the same speed as CPU speed, response time is improving much more slowly, so many small requests—to the operating system or across the network—are becoming relatively more costly than one large request.

One of the principle barriers to widespread use of software agents is the difficulty of running them securely. Allowing unknown code to run on your computer is very much like bringing the Trojan horse through the city gates. Some of the solutions are language-based, like Java, in which the interpreter is believed to be secure. Bugs in the interpreter can result in security holes, however, and the interpretation layer lowers the performance. Operating system support for the confinement of the agent is somewhat more attractive, but there is still a performance penalty, and the confinement is less than complete. On a CA, however, a program may easily run an agent within its own region of the array with no need for support from the operating system and no fear of Trojan horse attacks. With such low-overhead security, it should be possible to run software agents even on critical machines like database servers, allowing novel applications like data mining and filtering to be performed where the data resides at a great savings of network bandwidth.

Chapter 9

Conclusions

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> — Winston Churchill, speech on the battle of Egypt, 1942

The goal of this work has been to show that massively parallel computers are well-suited to general-purpose computation, that they could become the standard for commodity computers. More particularly, we wanted to show that the cellular array is the best parallel architecture to supplant the von Neumann machines we now use, and that it can serve as a general computational substrate to replace the ad-hoc parallelism of special-purpose hardware.

It is quite easy to disagree with this vision—if everyone agreed, this wouldn't need to be written. A strong case needs to be made if we are going to make such a radical change.

To establish the feasibility of using a cellular array as the standard general-purpose computer, one must do the following:

1. Make a clear theoretical case that the CA architecture is the best for the task.
2. Establish its relationship to other parallel models.
3. Create a realistic prototype system.
4. Show that general-purpose parallelism is competitive with special-purpose hardware devices.
5. Show that everyday computational tasks can in fact be parallelized.

6. Show that CAs can support as robust and powerful operating systems as von Neumann machines do.

We have met each of those goals:

1. We have shown that under reasonable assumptions, the CA model is as good as any other parallel architecture that can be physically built, in the limit.
2. We have given emulation results for the RAM, PRAM, BSP and LogP models.
3. We developed a prototype of a realistic system under a simulator.
4. We have shown with the RSA algorithm that parallel software can not only be competitive with special-purpose hardware, but that it can have significant advantages.
5. We developed an optimal parallel algorithm for document layout, a challenging example of an everyday computational task.
6. We gave a paper design for a client/server operating system with as rich functionality as a standard sequential OS, and better security.

The work we have presented is only a small fraction of the results a number of other researchers have contributed to the area, whether in systolic arrays, mesh algorithms, or cellular automata. Despite this, we have together only scratched the surface of a large uncharted territory. The von Neumann architecture has absorbed the talents of thousands of researchers for the last 50 years, and duplicating that success will not happen overnight.

9.1 Future work

Here are a few of the areas it seems most profitable to tackle next:

- Implement a full prototype in silicon. This would not be too difficult a project, and would give an invaluable testbed to gain experience programming.
- Develop a programming language. Our standard programming languages perfectly reflect the cost model of standard sequential machines, and this allows us to write programs in C, Pascal, or Fortran and predict their performance fairly closely. Although there are several parallel programming languages, these tend to hide the communication costs just as previous abstract models of parallel computation did. We need a higher-level language that matches the architecture well, making communication costs and the geometry of data

explicit. One of the consequences of this would be that global variables cannot exist, or at least must be obviously more expensive to access.

- Find a niche. It is nearly impossible to enter the mass market by staging a revolution. To become widely used, CAs will need some specialty market to shelter their early growth. One plausible market is high-speed network routers, which are in high demand and which require very high performance. High-end graphics and multimedia are other possibilities.

This is usually a section for the author to list some loose ends to be cleaned up in the next couple of years. In our case, though, this entire work has been merely the prologue: the real future work is nothing less than starting to engineer computers and software for 15 or 20 years hence.

Bibliography

[AB93] Arvind and S. Brobst. The evolution of dataflow architectures: from static dataflow to p-risc. *International Journal of High Speed Computing*, 5(2):125–153, June 1993.

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, California, 1989.

[AI87] J. Albert and K. Culik II. A simple universal cellular automaton and its one-way and totalistic version. *Complex Systems*, 1(1):1–16, 1987.

[AL93] Selim G. Akl and Kelly A. Lyons. *Parallel Computational Geometry*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[Ass94] Semiconductor Industry Association. *National Technology Roadmap for Semiconductors*. Sematech, Inc., Austin, Texas, 1994. (http://www.sematech.org/public/roadmap/94e_rdmp.htm)

[Ban71] E. R. Banks. Information processing and transmission in cellular automata. Technical Report MIT/LCS/TR-81, MIT Laboratory for Computer Science, January 1971.

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your Mathematical Plays*, volume 2. Academic Press, New York, 1982.

[BHP+96] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. Bsp vs logp. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, pages 25–32, June 1996. (http://www.math.unipd.it/~capri/spaa96.ps.Z)

[Bia94] Michael Biafore. Cellular automata for nanometer-scale computation. *Physica D*, 70(4):415–433, February 1994.

[BJ97] S.C. Benjamin and N.F. Johnson. A possible nanometer-scale computing device based on an adding cellular automaton. *Applied Physics Letters*, 70(17):2321–2323, April 1997.

[Ble93] Guy E. Blelloch. Prefix sums and their applications. In John H. Reif, editor, *Synthesis of Parallel Algorithsm*, chapter 1, pages 35–60. Morgan Kaufman, 1993.

[BP95] Gianfranco Bilardi and Franco P. Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27(2):172–182, June 1995.
(`http://www.cs.brown.edu/publications/techreports/reports/CS-93-20.html`)

[BRV89] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to programmable active memories. In *Systolic Array Processors*, pages 301–309, June 1989.
(`ftp://ftp.digital.com/pub/DEC/PRL/research-reports/PRL-RR-3.pdf`)

[Bur70] Arthur W. Burks. *Essays on Cellular Automata.* University of Illinois Press, Urbana, 1970.

[But69] Arthur R. Butz. Convergence with Hilbert's space filling curve. *Journal of Computer and System Sciences*, 3(2):128–146, May 1969.

[CG84] E. G. Coffman and E. N. Gilbert. Dynamic, first-fit packings in two or more dimensions. *Information and Control*, 61(1):1–14, April 1984.

[CKCN93] James B. Cole, Rudolph A. Krutar, Dennis B. Creamer, and Susan K. Numrich. A cellular automaton methodology for solving the wave equation. In *Proceedings of the International Conference on Supercomputing*, pages 348–356. ACM, July 1993.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, July 1993.

[CM85] Bernard Chazelle and Louis Monier. A model of computation for vlsi with related complexity results. *Journal of the Association for Computing Machinery*, 32(3):573–588, July 1985.

[Cod68] E. F. Codd. *Cellular Automata.* Academic Press, New York, 1968.

[CR73] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of computer and systems sciences*, 7(4):354–375, 1973.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[CvV93] János Csirik and André van Vliet. An on-line algorithm for multidimensional bin packing. *Operations Research Letters*, 13(3):149–158, April 1993.

[DC97] Richard P. Draves and Scott M. Cutshall. Unifying the user and kernel environments. Technical Report MSR-TR-97-10, Microsoft Research, March 1997.
(`ftp://ftp.research.microsoft.com/pub/tr/tr-97-10.ps`)

[EGCL89] Jr. Ed G. Coffman and F. Thomson Leighton. A provably efficient algorithm for dynamic storage allocation. *Journal of computer and system sciences*, 38(1):2–35, February 1989.

[Fre90] Edward Fredkin. Digital mechanics. An informational process based on reversible universal cellular automata. *Physica D*, 45(1-3):254–270, September 1990.

[FS92] Yossee Feldman and Ehud Shapiro. Spatial machines: A more realistic approach to parallel computation. *Communications of the ACM*, 35(10):61–73, October 1992.

[FT82] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3/4):219–253, April 1982.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on the Theory of Computing (STOC)*, pages 114–118. ACM, May 1978.

[Gar95] Max Garzon. *Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks*. Springer-Verlag, Berlin, 1995.

[GHSZ90] D. Greenberg, M. Horne, A. Shimony, and A. Zeilinger. Bell's theorem without inequalities. *American Journal of Physics*, 58(12):1131–1143, December 1990.

[Gil84] William J. Gilbert. A cube-filling Hilbert curve. *Mathematical Intelligencer*, 6(3):78, 1984.
(`http://math.uwaterloo.ca/~wgilbert/Research/HilbertCurve/HilbertCurve.html`)

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 1996 ACM Symposium on the Theory of Computing (STOC)*, volume 28, pages 212–219. ACM, May 1996.

[GvL89] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 2 edition, 1989.

[Hal94] J. Storrs Hall. A reversible instruction set architecture and algorithms. In *Physics and Computation*, pages 128–134, November 1994.

[Hil91] David Hilbert. Über die stetige Abbildung einer Linie auf flächenstüch. *Mathematische Annalen*, 38:459–460, 1891.

[HK95] Markus A. Hitz and Erich Kaltofen. Integer division in residue number systems. *IEEE Transactions on Computers*, 44(8):983–989, August 1995.

[Hor82] R. Michael Hord. *The Illiac IV, the first Supercomputer*. Computer Science Press, Rockville, Maryland, 1982.

[HPN97] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video: an Introduction to MPEG-2*. Chapman & Hall, New York, 1997.

[Jag90] Hosagrahar V. Jagadish. Linear clustering of objects with multiple attributes. In *ACM SIGMOD International Conference on Management of Data*, pages 332–342, May 1990.

[JD84] Kendall Preston Jr. and Michael J. B. Duff. *Modern Cellular Automata*. Plenum Press, New York, 1984.

[KL78] H. T. Kung and Charles E. Leiserson. Systolic arrays (for vlsi). In Iain S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings, 1978*, pages 256–282. SIAM, November 1978.

[Knu91a] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, 1991.

[Knu91b] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1991.

[Koç94] Çetin Koç. High-speed RSA implementation. Technical Report TR-201, RSA Laboratories, November 1994.

[Kun88a] Manfred Kunde. Routing and sorting on mesh-connect arrays. In *Proceedings of the 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures*, pages 423–433, June 1988.

[Kun88b] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Kun90] H.T. Kung. iWarp multicomputer with an embedded switching network. *Microprocessors and Microsystems*, 14(1):59–60, January 1990.

[Kun93] Manfred Kunde. Block gossiping on grids and tori: Deterministic sorting and routing match the bisection bound. In *Proceedings of the First European Symposium on Algorithms (ESA)*, volume 726 of *LNCS*, pages 272–283. Springer-Verlag, October 1993.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[Len94] Arjen Lenstra. Factoring. In Gerard Tel and Paul Vitányi, editors, *Distributed Algorithms*, pages 28–38. Springer-Verlag, September 1994.

[LRN93] T. Y. Lee, C. S. Raghavendra, and J. B. Nicholas. A fully distributed parallel ray tracing scheme on the Delta Touchstone machine. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 129–134, July 1993.

[Mar91] Norman Margolus. Physics and computation. Technical Report MIT/LCS/TR-415, MIT LCS, 1991.

[Mar94] Bruno Martin. A universal cellular automaton in quasi-linear time and its S-m-n form. *Theoretical Computer Science*, 123(2):199–237, January 1994.

[Mar96] Norman Margolus. Large-scale logic-array computation. In *Proceedings of the SPIE—The International Society for Optical Engineering*, volume 2914, pages 341–352, November 1996.

[McC95] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 46–61. Springer-Verlag, 1995.
(http://www.comlab.ox.ac.uk/oucl/users/bill.mccoll/p9.ps.Z)

[Mer93] Ralph C. Merkle. Reversible electronic logic using switches. *Nanotechnology*, 4(1):21–40, January 1993.
(http://nano.xerox.com/nanotech/electroTextOnly.html)

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, April 1985.

[Moo95] Gordon E. Moore. Lithography and the future of Moore's law. In *Proceedings of the SPIE—the International Society for Optical Engineering*, volume 2440, pages 2–17, February 1995.

[Nor89] Mats G. Nordahl. Formal languages and finite cellular automata. *Complex Systems*, 3:63–78, 1989.

[Ofm63] Y. Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics Doklady*, 7(7):589–591, January 1963.

[Pat95] David A. Patterson. Microprocessors in 2020. *Scientific American*, 273:48–51, September 1995.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. In *Plan 9 Programmer's Manual*, volume 2. Bell Laboratories, Murray Hill, NJ, 1995.
(http://plan9.bell-labs.com/plan9/doc/9.html)

[Rei93] John H. Reif, editor. *Synthesis of Parallel Algorithsm*. Morgan Kaufman, San Mateo, California, 1993.

[Rob87] Yves Robert. Systolic algorithms and algorithms. In Yves Robert, Françoise Fogelman Soulié, and Maurice Tchuente, editors, *Automata Networks in Computer Science*, chapter 8, pages 187–228. Princeton University Press, Princeton, New Jersey, 1987.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sag94] Hans Sagan. *Space-filling Curves*. Springer-Verlag, New York, 1994.

[Sam90] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, Massachusetts, 1990.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 124–134. IEEE, November 1994.
(`http://www.cwi.nl/~berthiau/articles/shor.ps`)

[Sib94] Jop F. Sibeyn. Desnakification of mesh sorting algorithms. In *Proceedings of the 2nd European Symposium on Algorithms (ESA)*, volume 855 of *LNCS*, pages 377–390. Springer-Verlag, 1994.
(`http://www.mpi-sb.mpg.de/~jopsi/dps/si15.ps.gz`)

[Sib95] Jop F. Sibeyn. Overview of mesh results. Research Report MPI-I-95-1-018, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, July 1995.
(`http://www.mpi-sb.mpg.de/~jopsi/dps/si12.ps.gz`)

[SK93] Peter Schroder and W. Kruger. Data parallel volume-rendering algorithms for interactive visualization. *Visual Computer*, 9(8):405–416, August 1993.

[Ste81] Nancy B. Stern. *From ENIAC to UNIVAC: an Appraisal of the Eckert-Mauchly Computers*. Digital Press, Bedford, Massachusetts, 1981.

[SV93] Mark Shand and Jean Vuillemin. Fast implementations of RSA cryptography. In *Proceedings, 11th Symposium on Computer Arithmetic*, pages 252–259. IEEE, June 1993.

[TM87] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines—A New Environment for Modeling*. MIT Press, Cambridge, Massachusetts, 1987.

[Tof77] Tommaso Toffoli. Computation and construction universality of reversible cellular automata. *Journal of Computer Systems Science*, 15(2):213–231, October 1977.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[Vav89] Stephen A. Vavasis. Gaussian elmination with pivoting is p-complete. *SIAM Journal of Discrete Mathematics*, 2(3):413–423, August 1989.

[VBR+96] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, H.H. Touati, and P. Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.
(ftp://ftp.digital.com/pub/DEC/PRL/research-articles/VBRSTB941.pdf)

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: a user-level network interface for parallel and distributed computing. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[vECGS92] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266. ACM, May 1992.

[VH94] Ravi Varadarajan and Injae Hwang. An efficient dynamic processor allocation algorithm for adaptive mesh applications. Technical Report TR-94-004, Department of Computer and Information Science and Engineering, University of Florida, May 1994.
(ftp://ftp.cis.ufl.edu/cis/tech-reports/tr94/tr94-004.ps)

[vN66] John von Neumann. *Theory of Self-reproducing Automata*. University of Illinois Press, 1966.

[WAD+93] Steve Ward, Karim Abdalla, Rajeev Dujari, Michael Fetterman, Frank Honori, Ricardo Jenez, Philippe Laffont, Ken Mackenzie, Chris Metcalf, Milan Minsky, John Nguyen, John Pezaris, Gill Pratt, and Russell Tessier. The NuMesh: A modular, scalable communications substrate. In *Proceedings of the International Conference on Supercomputing*, July 1993.
(ftp://cag.lcs.mit.edu/pub/numesh/published/ics93.ps.Z)

[Wol86] Stephen Wolfram, editor. *Theory and Applications of Cellular Automata*. World Scientific, Singapore, 1986.

[YK94] S. G. Younis and T. F. Knight, Jr. Asymptotically zero energy split-level charge recovery logic. In *International Workshop on Low Power Design*, pages 177–182, 1994.

[Zip91] Richard Zippel. The data structure accelerator architecture. Technical Report TR 91-1256, Cornell University, December 1991.