# Chapter 1

# Introduction

It takes all the running *you* can do,

to keep in the same place.

If you want to get somewhere else,

you must run at least twice as fast as that!

—Lewis Carroll

After briefly examining the challenges facing designing ever more powerful computers, we discuss the important issues in parallel processing and outline solutions. An overview of the book is also given.

## 1.1 The von Neumann Machine Paradigm

The past five decades have witnessed the birth of the first electronic computer [257] and the rapid growth of the computing industry to exceed $1,000 billion (annual revenue) in the U.S. alone [162]. The demand for high-performance machines is further powered by the advent of many crucial problems whose solutions require enormous computing power: environmental issues, search for cures for diseases, accurate and timely weather forecasting, to mention just a few [271]. Moreover, although unrelenting decrease in the feature size continues to improve the computing capability per chip, turning that into a corresponding increase in computing performance is a major challenge [152, 316]. All these factors point toward the necessity of sustained innovation in the design of computers.

That the **von Neumann machine paradigm** [37], the conceptual framework for most computers, considered at the system level will impede further performance gains is not hard to see. Input and output excluded, the von Neumann machine conceptually consists of a processing unit, a memory storing both programs and data, and a wire that connects the two. In an execution cycle the processing unit fetches from memory an instruction, and decodes and executes it, which may cause additional data movements in the wire, all in *serial* manner. Clearly, no matter how fast the processor and the memory are, the wire connecting the two sets the limit to the overall performance, as the machine can be only as fast as the wire can deliver information. The von Neumann machine moreover makes inefficient use of its resources, since only *one* location in memory is accessible in a cycle with the rest remaining idle [155]. Backus called this wire the **von Neumann bottleneck** [36].

The von Neumann bottleneck is of architectural origin [11]. With the cost of processing logic dropping rapidly, the advent of VLSI, and the development of automated design tools, the von Neumann machine paradigm becomes outdated for designing high-speed computers [43, 151], and major performance improvements must now come from architectural designs capable of **parallel processing**. Today, all high-performance computers use some kind of parallel processing, and their designs are extremely diverse to satisfy their equally diverse goals [43]. They include pipelining (superscalar and superpipeline), VLIW ("Very Long Instruction Word"), and so forth [152, 342]. But parallelism in improving uniprocessor performance is restricted by such factors as fundamental physical limits, diminishing returns, source level parallelism, and compiler technology, whereas parallel architectures promise an essentially open-ended scope of performance [316].

## 1.2 Issues in Parallel Processing

Six important issues in parallel processing are identified and discussed in this book: (i) network topology, (ii) interprocessor communication, (iii) fault tolerance, (iv) simulation of ideal parallel computation models, (v) asynchronism without compromising efficiency and with low sensitivity to variations in component speed, and (vi) on-line maintenance. They are treated briefly in the following paragraphs.

### Issue 1: network topology

All physical devices have natural limits imposed by the speed of light and materials, and they have to be extremely tiny to be fast (as it takes a signal at least $10^{-9}$ second to travel one foot), which is expensive and has serious reliability problems to tackle [316, 347]. Instead of relying solely on fast gates and small dimensions to reduce delays, parallel processing attempts to speed up computation by replicating the logic [186, 340]. With more than one processor, the immediate question is how to connect them to achieve the desired speed-up in a way that is both economical and physically feasible, since the network can easily dominate the hardware cost and program execution time.

Interconnection networks for parallel computers are surveyed in Chapter 3. Discussed there are important issues such as diameter, easiness of control, routing, tolerance for faults, and cost considerations. Concise as the coverage necessarily is, it is relatively complete and many important concepts are defined.

### Issue 2: fast interprocessor communication

Once connected to become a parallel computer, processors communicate *via* routing messages through wires or their equivalents. [1] Since signal-propagation time between widely separated modules can easily dominate the delay due to device-switching time, even if logic signals travel at the speed of light [340], the paramount issue here is how to minimize such delays, which is formalized as the **parallel routing problem**.

We first summarize our approach to, our results for, and previous work on, the parallel routing problem in Chapter 4. Then, in Chapter 5, fast, space-efficient ran-

[1] If processors do not communicate between themselves, they do not need to be connected in the first place. For example, today, 500 microprocessors together can easily carry out more than 2,500 million floating point operations per second (Mflops). But, that alone does not a Cray Y-MP make; Cray Y-MP's total peak performance is rated at 2,670 Mflops [163] (peak performance, cynics say, is nothing but "the maximum performance that the manufacturer guarantees no program will exceed" [44, 153] or the "speed of a computer when not running any software" [334, p. 40]). See [129, Table 1] and [163, Tables 1 and 2] for the (claimed) performance of several supercomputers from the U.S. and Japan. See also [312] for a brief history of supercomputers and [44, 247, 251, 327, 334, 355, 386] for an up-to-date account of fast computers.

domized routing algorithms for the hypercube and the de Bruijn networks are described and analyzed. Although these are randomized algorithms, each fails to route successfully with extremely small probability. Take the algorithm for the hypercube network, FSRA ("Fault-Tolerant Subcube Routing Algorithm"), as an example. Let $N$ denote the size of the network. This algorithm runs in $2 \log N + 1$ time[2] without queueing delay and uses only constant size buffers. Its probability of unsuccessful routing is at most $N^{-2.419 \log N + 1.5}$, which for all practical purposes is zero. This result also solves Rabin's conjecture [285].

Two numerical examples can illustrate how improbable FSRA may fail to route successfully. The probability of unsuccessful routing in a $1,024$-node hypercube network is less than $4.9 \cdot 10^{-69}$. As another example, a $2^{16}$-node hypercube network, the size of a full-size Connection Machine CM-2 [349], would have an unsuccessful routing with probability at most $6.4 \cdot 10^{-180}$.

### Issue 3: fault tolerance without high cost

There are no fault-free devices in the real world [326]. Even highly reliable components make transient faults due to reasons such as thermal fluctuations [245]. With large numbers of components, as parallel computers almost by definition must be, some component will go wrong with non-negligible probability according to simple statistical principles. Indeed, it has been surmised that "the practical size limit of [parallel] machines may well depend on reliability — the mean time between failure and mean time to repair must be small enough to permit the system to compute for a useful fraction of the time" [316]. This has serious consequences for parallel computers. To save a computation from errors, not only the processor that experienced faults has to take actions, *all* other processors receiving data from it must also take actions, then processors that received data from these processors, too, etc. This wave of corrective actions was dubbed the "**domino effect**" by Randell [292]. Since this effect can occur even if there is only one transient fault at only one site during the whole computation, clearly it is important to confine all errors to their originating places [382].

We will show that the fast routing schemes developed in Chapter 5 for the hypercube and the de Bruijn networks tolerate random link failures. For example, the one for the hypercube network (FSRA, that is) tolerates $\Theta(N)$ random link failures with high probability. A voting scheme is also developed in Subsection 2.6 to mask out node failures. This scheme can be easily incorporated into FSRA to create regions of fault containment, as shown in the final chapter.

[2] All logs are to the base 2 throughout this book unless noted otherwise.

### Issue 4: simulation of ideal parallel computation models by feasible parallel computers

Communication has long been abstracted out of computation models, which traditionally focus on *logical* operations. The research on parallel processing, however, demonstrates clearly that communication is an inseparable part of computation [245].

In Chapter 6 parallel computation models are surveyed. At the end of that chapter, we briefly show how the PRAM ("Parallel Random Access Machine") model — the most popular theoretical model — can be simulated on the hypercube network with a slowdown of $O(\log N)$ with probability tending to one as $N$, the number of processors, or the number of PRAM instructions approaches infinity. Our main focus there is a class of PRAM programs whose efficient simulation does not require the expensive hashing of the address space. We show that such PRAM programs can be simulated with a slowdown of $8 \log N$ with almost certainty. Both simulation schemes are IDA-based and fault-tolerant.

### Issue 5: asynchronism without compromising speed and low communication complexity, but with low sensitivity to variations in link or processor speed

It is desirable that a parallel computer has no global control, which may become a communication bottleneck and a single point of failure. Global clocks in particular are also difficult to implement due to the problem of clock skews and delays in large systems [119]. It is moreover desirable that the execution time is not too sensitive to variations in component speeds, which may be caused by a wide variety of reasons such as differences in wire length or computing power (as in heterogeneous systems [192]), statistical phenomena, variations in electrical characteristics, and processors diverted to run diagnostics.

In Chapter 7 we address both problems. First we show that our routing schemes can run on asynchronous networks without loss of efficiency in either time or communication complexity, defined as the number of **synchronization messages** used to simulate the global clock pulses. Then we show that FSRA is not sensitive to variations in component speed in that if a link or a processor is slowed by an amount of $\Delta t > 0$, the run-time of FSRA will skew by only $O(\Delta t)$, which is *linear* in $\Delta t$ and *independent* of the size of the network. As a consequence, increase in the machine size will not affect the sensitivity to component speed.

### Issue 6: efficient, on-line maintenance

With fault tolerance comes redundancy [326], which should be exploitable to make maintenance less disrupting to the user. We hence propose a novel on-line mainte-

nance and repair concept.

In Chapter 8 we show that wires of the hypercube network can be partitioned into 352 sets of roughly equal sizes such that those in the same set can be disabled simultaneously *without* disrupting the ongoing computation or degrading the routing performance much, if FSRA is used. This partition can also be computed locally and efficiently. As a result, efficient, on-line wire testing and replacement on the hypercube network can be realized. Furthermore, the maintenance procedure can be completed in 352 cycles, *independent* of the number of processors.

The idea of space-efficient **information dispersal**, pioneered by Rabin [285], is vital to results in this book. An information dispersal algorithm (IDA), parametrized by $n$ and $m$ for $m \leq n$, is an algorithm which breaks any given piece of information $F$ into $n$ pieces, each only one $m^{\text{th}}$ the length of $F$, so that any $m$ of them suffice to reconstruct $F$. Efficient IDAs with ideas from the theory of error-correcting codes, especially those related to Reed-Solomon codes, are presented in Chapter 2.

With these results, we are only a step shy of a parallel computing system design. We take that extra step in Chapter 9, where a fault-tolerant hypercube parallel computer is sketched. That design, called the **hypercube parallel computer** (HPC), uses replication of program execution and employs FSRA as the routing scheme.

Fault-tolerant IDA-based routing schemes depend heavily on finding node-disjoint paths in the network. The classic result of Menger [72], linking connectivity with the number of node-disjoint paths, is not strong enough because the *lengths* of these paths — which determine the efficiency of routing — are left out. Such motivations lead to graph-theoretical concepts that take into account path lengths. The performance of the standard two-phase fault-tolerant IDA-based routing scheme due to Rabin [285] can now be expressed in a general form when applied to **node-symmetric** graphs, to whose class the hypercube belongs.

## 1.3 Overview of the Book

The concepts of information dispersal and an efficient FFT-based IDA are presented in Chapter 2. The application of IDA to voting is also developed there. Chapter 3 briefly surveys interconnection networks (**Issue 1**). From there on, our main focus will be the hypercube network, though routing schemes for the de Bruijn network will also be presented. We present fast, fault-tolerant routing algorithms for these two networks in Chapter 5 (**Issues 2** and **3**), after the preview in Chapter 4. A general formulation of two-phase IDA-based routing and its connection to graph theory are also covered. The simulation of PRAMs is addressed in Chapter 6 (**Issue 4**), where a

class of PRAM programs is treated in detail since such programs may be more amiable to general-purpose simulation in practice. In Chapter 7 we prove that our routing schemes can run on asynchronous networks without loss of efficiency. Furthermore, we show that FSRA has low sensitivity to variations in link and processor speeds (**Issue 5**). In Chapter 8 it is proved that FSRA allows efficient on-line maintenance (**Issue 6**). Finally, we propose a fault-tolerant parallel computer in Chapter 9. We remark that **Issues 5** and **6** have not been addressed analytically before, to the author's best knowledge. A more technical summary of this book can be found in the preface.

Due to limited space, this book cannot discuss all relevant issues concerning parallel processing. Fortunately, there are excellent papers and books to fill that void: architecture [11, 110, 120, 127, 153, 341], memory hierarchy [161], program transformation [185, 341], synchronization [35, 103, 104, 109, 128, 196, 293], VLSI [194, 210, 221, 245, 331, 352, 360], cache coherence [19, 109, 383], experience of using and building parallel computers [22, 91, 123, 167, 168, 315, 382], data flow architecture [96], limitation in speedup [13, 121, 144, 180], performance analysis [60, 86, 98, 99, 182, 204, 264, 311], and the possible impact of optical technology [54, 107, 113, 141, 154].

Finally, some words on history. John von Neumann is himself a pioneer in parallel processing [21, pp. 30, 41, and 275]; the suffix "bottleneck" applies only to the von Neumann machine paradigm. On the other hand, some prominent researchers like Hennessy and Patterson believe the term, von Neumann machine, "gives too much credit to von Neumann, who wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines" [153, pp. 23–24]. There are also disputes about whether Atanasoff at Iowa State University in the early 1940s built the first electronic computer before Eckert and Mauchly [153, 257].

# Chapter 2

# Information Dispersal

Ten times thyself were happier than you are,

If ten of thine ten times reconfigured thee.

—Shakespeare

The concept of information dispersal is introduced, and an information dispersal algorithm (IDA) based on polynomial evaluation and interpolation is described. This scheme can take advantage of the Fast Fourier Transform (FFT) algorithm under certain circumstances. Several variations on this algorithm are also explored. An IDA-based software voting method is presented as an application and analyzed under a random fault model.

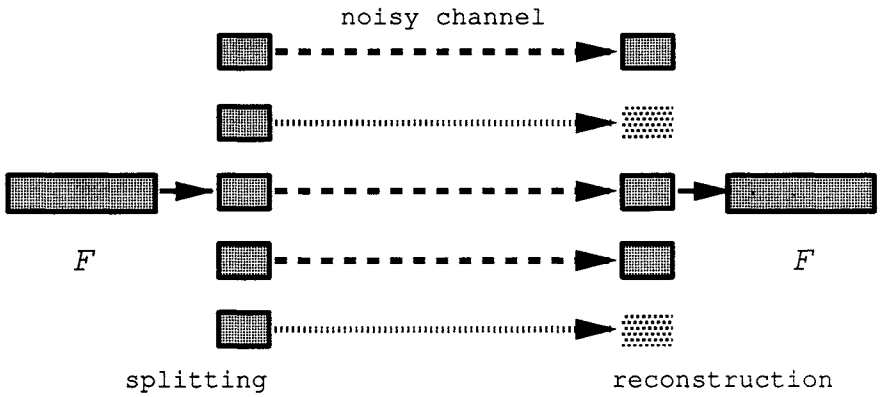


Figure 2.1: THE WORKING OF IDA WITH $n = 5$ AND $m = 3$. The original information $F$ is first split into five pieces and two of them are subsequently unavailable. However, as long as at least three pieces are accessible, as in the case above, $F$ can be reconstructed.

## 2.1 Introduction

An **information dispersal algorithm** (IDA) is any efficient method that can split a given string of bits $F$ into $n$ **pieces** in such a way that any $m$ of them suffice to reconstruct the original $F$. Each piece, moreover, is of size $|F|/m$, where $|F|$ is the size — measured as the number of characters throughout this book — of $F$; the pieces, therefore, total $(n/m)\,|F|$ characters. The two parameters $n$ and $m$ can be any two integers as long as $m \leq n$. See Figure 2.1 for illustration. The study of space-efficient information dispersal with application to fault-tolerant interprocessor communication is due to Rabin [285]. Subsection 2.5.4 reviews related work.

The value of $m$ can be adjusted to meet the desired combination of the reliability level and the total space used by the $n$ pieces. With $m$ approaching $n$, less space is consumed by the pieces; on the other hand, with $m$ approaching one, more pieces can afford to be lost since less of them are needed to reconstruct $F$. Hence, given $n$, we can fine-tune $m$ to strike a balance between two seemingly diametrically opposed goals: efficient use of space and high degree of reconstructibility.

In this chapter we introduce an efficient IDA based on the Vandermonde matrix where the splitting operation is **polynomial evaluation** and the reconstruction op-

eration **polynomial interpolation**. When certain conditions are satisfied, we can even apply the **Fast Fourier Transform** (FFT). Our scheme is asymptotically more efficient than Rabin's original IDA [285] (see Subsection 2.5.2).

Information dispersal can be viewed from the error-correcting codes standpoint [63, 228, 238, 272]: one wants to decode the code words from errors [285]. As we are mainly concerned with a particular kind of errors, **erasures**, our approach takes the more limited *interpolation* standpoint: one wants enough sample values to interpolate the function representing the original information. The error-correcting codes viewpoint is explored in Subsection 2.5.3.

One natural application of IDA is the dispersal of files in a computer system such as the Redundant Arrays of Inexpensive Disks (RAID) systems [58, 153, 265, 372]. Instead of keeping only one copy, we may use IDA to split a file into smaller files and disperse them across the network. This increases the availability of a file, since it is accessible as long as at least $m$ of the deposit sites are up. This IDA-based scheme, furthermore, saves disk space in comparison with the more common replication scheme where each file is copied in its entirety. Other applications can be found in [286].

In this book, IDA is employed to achieve fast, fault-tolerant parallel routing, where packets are first split into pieces by IDA before being routed in parallel to their destinations (see Chapter 5). Since each packet can afford to lose up to $n - m$ of its pieces during routing, loss of information due to buffer overflow and/or link failures can be tolerated. Allowing pieces to be lost also eliminates queueing delay.

IDA can also be used to **filter** out polluted data, thus erecting some defense against error propagation and the "domino effect" (see Chapter 9). The algorithm FILTERING for this purpose and its analysis is presented in Section 2.6. In FILTERING, voting is applied to *pieces* instead of the complete results; hence, messages are shorter, implying faster communication. FILTERING works roughly as follows. If a processor detects that its own pieces do not agree with, say, the majority of the received pieces, it may safely declare that its result is wrong and then take corrective steps. On the other hand, a processor whose pieces are supported by the majority of the received pieces can treat as **suspects** those processors having sent conflicting pieces. This voting process can be invoked before each communication round to prevent innocent processors from accepting invalid data, thus creating regions of fault containment (see Chapter 9).

This chapter is organized as follows. First we introduce basic concepts from number theory and algebra needed for later developments. Polynomial evaluation and interpolation, the cornerstone of our IDA, are then discussed as well as their relation with the FFT. The general approach for IDA is presented in Section 2.4, and our specific implementation as well as its variations in Section 2.5. Application to voting