Highly Fault-Tolerant Parallel Computation

Extended Abstract*

Daniel A. Spielman†
Department of Mathematics, M.I.T.
Cambridge, MA 02139
spielman@math.mit.edu

Abstract

We re-introduce the coded model *of fault-tolerant computation in which the input and output of a computational device are treated as words in an error-correcting code. A computational device correctly computes a function in the coded model if its input and output, once decoded, are a valid input and output of the function. In the coded model, it is reasonable to hope to simulate all computational devices by devices whose size is greater by a constant factor but which are exponentially reliable even if each of their components can fail with some constant probability.*

We consider fine-grained parallel computations in which each processor has a constant probability of producing the wrong output at each time step. We show that any parallel computation that runs for time t on w processors can be performed reliably on a faulty machine in the coded model using $w \log^{O(1)} w$ processors and time $t \log^{O(1)} w$. The failure probability of the computation will be at most $t \cdot exp(-w^{1/4})$.

The codes used to communicate with our fault-tolerant machines are generalized Reed-Solomon codes and can thus be encoded and decoded in $O\left(n \log^{O(1)} n\right)$ sequential time and are independent of the machine they are used to communicate with.

We also show how coded computation can be used to self-correct many linear functions in parallel with arbitrarily small overhead.

1. Introduction

When the function of a processor is critical, it is common to use three in place of the one so that even if one fails, its instructions will be overridden by the other two. Similarly, one could protect against the failure of k processors by following the instructions of a majority of $2k+1$. But, what if these processors are components of a large parallel machine? Is it necessary to replicate each processor $2k+1$ times to insure against the loss of any k? We show that, in many cases, the answer is *no*. At the cost of a polylogarithmic increase in size and a polylogarithmic slow-down, fine grained parallel machines can be made to tolerate the failure of a constant fraction of their processors with a probability of failure exponentially small in their number of processors. This should be contrasted with the naive scheme in which the probability of failure is exponentially small in k.

In 1952, Von Neumann [vN56] introduced the study of computation by circuits with faulty gates. His argument (later made rigorous by Dobrushin and Ortyukov [DO77]) demonstrated that a circuit of m gates could be reliably simulated by a circuit of $O(m \log m)$ gates, even if each gate were allowed to fail with constant probability. He considered circuits with one output and considered the computation reliable if the simulating circuit would produce the correct output with probability some constant close to one.

His result seems optimal: if any gate can fail with constant probability, then there is a constant probability that the output gate will fail; similarly, unless each input is replicated at least a logarithmic number of times, then there is a constant probability that some input will be misread (for a rigorous proof of this assertion, see [Gál91, RS91]). However, this argument only concerns the problem of *communicating* with the circuit,

*Errata to this paper will be available at `http://www-math.mit.edu/~spielman`.

†Supported in part by an NSF postdoc.

not *computing* with the circuit. Both obstacles disappear if one allows the inputs and outputs of a circuit to be treated as words in an error-correcting code. If the input to a circuit is encoded by a good error-correcting code and if the output is treated as a partially-corrupted word and then decoded, it seems reasonable to simulate any circuit by one that is larger by only a constant factor but which can tolerate with exponentially high probability the random corruption of a constant fraction of its gates. We achieve exponentially reliable simulations whose blow-up is polylogarithmic for a large family of circuits.

We produce encoding and decoding functions, E and D, such that for any parallel machine M with w processors that runs for time t, we can build a fault-tolerant parallel machine M' with $w \log^{O(1)} w$ processors that runs for time $t \log^{O(1)} w$ such that

$$\text{Prob}[D(M'(E(x))) = M(x)] > 1 - t \cdot 2^{-w^{1/4}},$$

even if each processor of M' is allowed to fail independently with probability ϵ, for some small $\epsilon > 0$.

If we consider the problem of simulating one circuit by another, without placing restrictions on the depth of the simulating circuit, then we can obtain more reliable computations. For any leveled circuit C with t levels and at most w gates on any level, we can build a fault-tolerant circuit C' of size $tw \log^{O(1)} w$ such that

$$\text{Prob}[D(C'(E(x))) = C(x)] > 1 - t \cdot 2^{-w^{1-1/\log\log w}},$$

even if each gate of C' is allowed to fail independently with probability ϵ.

E and D encode and decode the concatenation of a generalized Reed-Solomon code with a repetition code, and can thus be computed by circuits of size $O(n \log^2 n)$. In addition to having encoded inputs and outputs, each stage of our computation is a codeword which is transformed into a codeword representing the next stage of the computation. Because we use the same codes for all computations, we guarantee that the computation occurs in the simulating machine rather than in the encoding or decoding operations. Moreover, we obtain a model of computation in which the output of one machine can be used as the input of another, without the interference of any encoder or decoder. Thus, one should think of the encoded inputs and outputs as merely being an alternative communication format.[1] There is no need to decode the communications until one needs to transform the encoded information into an unencodable action.

In his original paper, von Neumann [vN56] suggested that coding theory should somehow be applied to constructing fault-tolerant circuits. Elias [Eli58] introduced a notion of coding for fault-tolerant parallel computation that would allow one to compute many instances of one linear function in the presense of noise. However, his model was overly restrictive, allowing one to prove negative results for general computation [Eli58, Win62, Ahl84]. In Section 8, we present an application of Elias's ideas to self-correcting linear functions (as defined in [BLR90]). The model that we present here was essentially introduced by Taylor [Tay68]. Taylor used Gallager's low density parity check codes to construct memories that were stable even in the presense of faults. He then demonstrated that any linear function could be computed with high reliability with gates that fail with constant probability (his results for general computation were in error—see [Pip90]). For a more complete survey of work on fault-tolerant computation, we direct the reader to [Pip90].

The techniques that we use in our construction are derived from those used to construct small probabilistically checkable proofs in [BFL91, BFLS91, Sud92, BF93, PS94]. Other advances in the development of probabilistically checkable proofs geared at decreasing the number of bits read or random bits used do not seem to help us in our construction. Gàl and Szegedy [GS95] make an interesting connection between probabilistically checkable proofs and fault-tolerant circuits that is very different from the one we make here—in their constructions, all the computation occurs in the encoding and the work of the fault-tolerant circuit is devoted to obtaining a 0/1 output.

In Section 3, we define our coded model of computation. We review von Neumann's construction in Section 4. In Section 5, we describe the polynomial codes that we will use to encode the inputs, outputs, and each stage of our computations. In Section 6, we show how fault-tolerant computations can be performed on polynomial codes. In Section 7, we define our model of parallel computation, describe how general parallel computations can be encoded by polynomial codes, and describe our constructions. In Section 8, we provide a simple example of how ideas of coded computation can be used to make the computation of linear functions more reliable by demonstrating that self-correction of linear functions can be performed in parallel with very little overhead. We conclude with a discussion of how our work might

[1] As Taylor [Tay68] points out, *uncoded* communication should be considered unusual since almost all data communication requires the use of error-correcting codes of some type.

be extended.

2. Notation

If S and T are sets, we write S^T to denote the set of $|T|$-tuples of elements of S indexed by elements of T (*i.e.*, the set of functions from T to S). For d an integer, S^d denotes the set of d-tuples of elements of S.

The parameters of a circuit that we measure are its *height* and its *width*. These are defined by assigning a *level* to each gate in a circuit as follows: the inputs to the circuit lie on level zero, and the level of a gate is one greater than the maximum of the levels of its inputs. The *height* of the circuit is the maximum level of any gate. The *width* is the maximum over i of the number of wires that go from gates on level i or less to gates on level $i+1$ or more. Essentially, width measures how much space is required to evaluate a circuit. Note that the width of a circuit is always at least its number of inputs.

3. Coded Computation

In this section, we define a coded model of fault-tolerant computation. We begin with a definition that captures the notion of *coding*.

Definition 1. A pair of functions E, D are an *encoding-decoding pair* if there exists a function l such that $E : \{0,1\}^n \to \{0,1\}^{l(n)}$ and $D : \{0,1\}^{l(n)} \to \{0,1\}^n \cup \{?\}$ such that $D(E(\vec{a})) = \vec{a}$ for all $\vec{a} \in \{0,1\}^n$.

Of course, one can make similar definitions with larger alphabets. While it is not explicit in this definition, we will usually require that E and D be encoding and decoding functions for an error-correcting code.

Definition 2. Let E, D be an encoding-decoding pair. A parallel machine M' (ϵ, δ, E, D)-*simulates* a machine M if

$$\text{Prob}[D(M'(E(\vec{a}))) = M(\vec{a})] > 1 - \delta$$

for all inputs $\vec{a}$, even if each processor of M' has some probability less than ϵ of producing the wrong output at each time step. Similarly, a circuit C' (ϵ, δ, E, D)-*simulates* a circuit C if

$$\text{Prob}[D(C'(E(\vec{a}))) = C(\vec{a})] > 1 - \delta$$

for all inputs $\vec{a}$, even if each wire of C' has some probability less than ϵ of producing the wrong output. The *blow-up* of the simulation is the number of gates in C' divided by the number of gates in C.

Remark 3. *When we say that each wire or processor has some probability less than ϵ of failing, we mean that the failures are consistent with Pippenger's* ϵ-admissible failure model [Pip89]. *Essentially, Pippenger's model assumes that, when it comes time for a signal to traverse a wire, there is a function that that takes the state of every previously computed wire in the circuit as input and outputs a number ϵ' between 0 and ϵ. The value of the wire is then flipped with probability ϵ'. If the value of the wire is flipped, then we say that the wire has failed. For simplicity, the reader might want to consider the weaker model in which each wire is assigned some probability less than ϵ of failing, independent of the computation being performed.*

Remark 4. *When a processor fails, we will assume that its output can be arbitrarily bad. We can make this assumption because the processors that appear in our constructions will only output one bit at a time.*

For an encoded computation to be meaningful, we need to be sure that the computation actually occurs in the machine M' rather than in the encoding or decoding operations. One way to guarantee this is to require that the encoding and decoding functions be computed by circuits smaller than any known circuit computing the same function as M. This condition is usually satisfied by our construction because our encoding and decoding functions can be computed in time $n \log^{O(1)} n$.

Our encoding and decoding functions satisfy a much stronger condition: *we use the same encoding and decoding functions in all simulations.* That is, the encoding and decoding only depend on the number of processors in the machine to be simulated. Such a system has the advantage that it allows for a consistent system of computation: the output of one of our fault-tolerant machines can be used as the input to another without any encoding or decoding needed to make the transition. Thus, we can consider a world in which the input and output of every computational device is encoded by an error-correcting code. Communications only need be decoded when they are to be realized as action rather than computation.

On the other hand, one could reasonably discuss coded fault-tolerant computation in which the encoding and decoding function do depend on the device simulated and even encode and decode different error-correcting codes, so long as the encoding and decoding circuits are smaller than the device to be simulated.

4. Local Coding

Von Neumann constructed fault-tolerant versions of ordinary circuits by adjusting the circuits locally. He replaced each wire in a circuit with a *bundle* of r wires. During fault-free computation each wire in a bundle would carry the same value. During fault-tolerant computation, the value of a wire is represented by the majority of the wires in its corresponding bundle. This majority was arranged to be strict, so that a $1-\epsilon$ fraction of the wires in a bundle should all carry the same value, for some small ϵ. Where two wires in the original circuit would meet at a gate, two bundles of wires in the fault-tolerant circuit meet at a collection of gates, each of which would act on one wire from each bundle. If a $1-\epsilon$ fraction of the wires in each incoming bundle all carried the same value, then at least a $1-2\epsilon$ fraction of the wires in the outgoing bundle would carry the same value. To boost this majority back to $1-\epsilon$, von Neumann fed such a bundle into an *amplifier* that would boost the agreement of the wires to at least a $1-\epsilon$ fraction. This amplifier could be constructed from $O(r)$ gates and wires. Dobrushin and Ortyukov [DO77] proved that, even if each wire is allowed to fail with some small constant probability, the probability that any bundle would fail to represent its intended fault-free value is $2^{-O(r)}$. The fault-tolerant circuit was then capped with a device that would reliably compute one wire representing the value of the output bundle. Thus, a circuit of w wires could be made fault-tolerant at a cost of $O(\log w)$ blow-up.

The constructions of [vN56, DO77] were probabilistic. Using explicit constructions of expander graphs, Pippenger [Pip85] made these constructions explicit. We summarize Pippenger's theorem for later use:

Theorem 5. *There is a constant $\epsilon_P > 0$ such that, for all circuits C, there is a means for replacing each wire in C with a bundle of $O(r)$ wires and an amplifier consisting of a circuit of size $O(r)$ so that the probability that any bundle in the circuit fails to represent its intended value is at most $w2^{-r}$. The blow-up of such a simulation is $O(r)$.*

One can view this scheme as encoding its inputs with a *repetition code*—one in which each symbol is repeated many times.

This scheme can be directly applied to fine-grained parallel computations: Each constant-bit processor can be replaced by a circuit of constant size. We can then apply the theorem, obtain a new circuit, and now view each gate in the new circuit as a (very) small processor.

5. Polynomial Coding

In this section, we define the generalized Reed-Solomon codes that we will compose with repetition codes to encode the inputs and outputs of our computations. The state of each stage of our computations will also be encoded by such a code. We point out that these codes can be encoded and decoded efficiently and develop the terminology that we will use to discuss these codes.

Definition 6. Let $\mathcal{F}$ be a field and let $\mathcal{H} \subset \mathcal{F}$. We define the code $C_{\mathcal{H},\mathcal{F}}$ by defining its encoding function. The *encoding function*

$$E_{\mathcal{H},\mathcal{F}} : \mathcal{F}^{\mathcal{H}} \to \mathcal{F}^{\mathcal{F}}$$

acts by treating its input as an $\mathcal{F}$-valued function on $\mathcal{H}$, finding the unique degree $(|\mathcal{H}|-1)$ polynomial that interpolates this function, and writing the values of this polynomial at every point of $\mathcal{F}$. The code $C_{\mathcal{H},\mathcal{F}}$ is the image of $\mathcal{F}^{\mathcal{H}}$ under $E_{\mathcal{H},\mathcal{F}}$. The *decoding function*

$$D_{\mathcal{H},\mathcal{F}} : \mathcal{F}^{\mathcal{F}} \to \mathcal{F}^{\mathcal{H}} \cup \{?\}$$

takes a word $\vec{a} = (a_1, \ldots, a_{|\mathcal{F}|})$ as input. If there is a codeword $\vec{b}$ of $C_{\mathcal{H},\mathcal{F}}$ that differs from $\vec{a}$ in fewer than $(|\mathcal{F}| - |\mathcal{H}|)/2$ places, then $D_{\mathcal{H},\mathcal{F}}$ outputs $E^{-1}_{\mathcal{H},\mathcal{F}}(\vec{b})$; otherwise, it outputs "?".

The code $C_{\mathcal{H},\mathcal{F}}$ is usually called an *extended Reed-Solomon code* (see [MS77, vL92]). To make sense of the definition of the decoding function, one needs to observe that distinct codewords differ in at least $|\mathcal{F}|-|\mathcal{H}|$ places, so there is no ambiguity as to the correct output of the decoding function.

Theorem 7 (Justesen, Sarwate). *The encoding and decoding functions $E_{\mathcal{H},\mathcal{F}}$ and $D_{\mathcal{H},\mathcal{F}}$ can be computed by circuits of size $|\mathcal{F}| \log^{O(1)} |\mathcal{F}|$.*

Proof: The encoding function is just polynomial interpolation, for which efficient circuits are presented in [AHU74] and [JáJ92]. Justesen [Jus76] and Sarwate [Sar77] demonstrate that efficient algorithms for computing the Half-GCD of two polynomials can be used to decode Reed-Solomon and Goppa codes. They relied on [AHU74] for their computation of the Half-GCD in time $O(n \log^2 n)$. While the proof in [AHU74] was faulty, it has been corrected (see [Str83] and [BCS96, Chapter 3, Sections 1 and 2]). □

A close relative of the decoding function is the *error-correction function*. Where the decoding function outputs a vector in $\mathcal{F}^{\mathcal{H}}$, an error-correction function outputs a codeword close to the input word. That is, the error-correction function

$$D^k_{\mathcal{H},\mathcal{F}} : \mathcal{F}^{\mathcal{F}} \to \mathcal{F}^{\mathcal{F}} \cup \{?\}$$

maps its input to a codeword of $C_{\mathcal{H},\mathcal{F}}$ that differs in at most k places, if such a codeword exists. If no such codeword exists, the function outputs "?". The parameter k is included in this definition because the complexity of computing the error-correction function depends on k. In particular, for $k = O(\sqrt{|\mathcal{F}|})$, there is a randomized algorithm of Kaltofen and Pan [KP94] from which we can build a circuit that computes $D^k_{\mathcal{H},\mathcal{F}}$ and which has polylogarithmic depth and almost linear size.

Theorem 8 (Kaltofen-Pan). *There is a randomized parallel algorithm that will solve a $k \times k$ Toeplitz system over any finite field with probability $1 - 1/k$ in time $\log^{O(1)} k$ using $k^2 \log^{O(1)} k$ processors.*

Lemma 9. *The function $D^k_{\mathcal{H},\mathcal{F}}$ can be computed by a randomized parallel algorithm that takes time $\log^{O(1)} |\mathcal{F}|$ on $(k^2 + |\mathcal{F}|) \log^{O(1)} |\mathcal{F}|$ processors, for $k < (|\mathcal{F}| - |\mathcal{H}|)/2$. The algorithm succeeds with probability $1 - 1/q$.*

Proof: Standard techniques for decoding Reed-Solomon codes (see [MS77, Chapter 8]), compute $D^k_{\mathcal{H},\mathcal{F}}$ by solving a $k \times k$ Toeplitz system, multiplying two polynomials over $\mathcal{F}$ modulo $(x^{|\mathcal{F}|} - x)$, and dividing two polynomials over $\mathcal{F}$. Using efficient algorithms for the Finite Fourier Transform, the multiplication and the division can be performed with $|\mathcal{F}| \log^{O(1)} |\mathcal{F}|$ processors in $\log^{O(1)} |\mathcal{F}|$ time [JáJ92]. To solve the Toeplitz system, we use the randomized algorithm of Theorem 8 that uses at most $k^2 \log^{O(1)} |\mathcal{F}|$ processors and takes time $\log^{O(1)} |\mathcal{F}|$. □

Our construction will use bivariate analogues of the codes $C_{\mathcal{H},\mathcal{F}}$.

Definition 10. Let $\mathcal{F}$ be a field and let $\mathcal{H} \subset \mathcal{F}$. The code $C_{\mathcal{H}^2,\mathcal{F}}$ is defined by its encoding function

$$E_{\mathcal{H}^2,\mathcal{F}} : \mathcal{F}^{\mathcal{H}^2} \to \mathcal{F}^{\mathcal{F}^2}$$

which treats its input as an $\mathcal{F}$-valued function on $\mathcal{H}^2$, finds the unique bivariate polynomial of degree at most $(|\mathcal{H}| - 1)$ in each variable that interpolates this function, and outputs the values of this polynomial at every point of $\mathcal{F}^2$. The decoding function

$$D_{\mathcal{H}^2,\mathcal{F}} : \mathcal{F}^{\mathcal{F}^2} \to \mathcal{F}^{\mathcal{H}^2} \cup \{?\}$$

takes a word $\vec{a} \in \mathcal{F}^{\mathcal{F}^2}$ as input. If there is a codeword $\vec{b}$ of $C_{\mathcal{H}^2,\mathcal{F}}$ that differs from $\vec{a}$ in fewer than $((|\mathcal{F}| - |\mathcal{H}|)/2)^2$ places, then $D_{\mathcal{H}^2,\mathcal{F}}$ outputs $E^{-1}_{\mathcal{H}^2,\mathcal{F}}(\vec{b})$; otherwise, it outputs "?".

The code $C_{\mathcal{H}^2,\mathcal{F}}$ is a *generalized Reed-Solomon code* (see [MS77, vL92]). These codes inspire the definition of the *maximum degree* of a multivariate polynomial to be the maximum of its degrees in each variable.

We now describe standard algorithms for encoding and error-correcting the bivariate polynomial codes:

Bivariate encoding algorithm:
input: a word $(a_{x,y})_{(x,y)\in\mathcal{H}^2}$

- For each $y_0 \in \mathcal{H}$, compute
$$(b_{x,y_0})_{x\in\mathcal{F}} = E_{\mathcal{H},\mathcal{F}}((a_{x,y_0})_{x\in\mathcal{H}})$$
- For each $x_0 \in \mathcal{F}$, compute
$$(c_{x_0,y})_{y\in\mathcal{F}} = E_{\mathcal{H},\mathcal{F}}((b_{x_0,y})_{y\in\mathcal{H}})$$
- Output $(c_{x,y})_{(x,y)\in\mathcal{F}^2}$.

Bivariate error-correcting algorithm:
input: a word $(a_{x,y})_{(x,y)\in\mathcal{F}^2}$ and a parameter k

- For each $y_0 \in \mathcal{F}$, compute
$$(b_{x,y_0})_{x\in\mathcal{F}} = D^k_{\mathcal{H},\mathcal{F}}((a_{x,y_0})_{x\in\mathcal{F}})$$
- For each $x_0 \in \mathcal{F}$, compute
$$(c_{x_0,y})_{y\in\mathcal{F}} = D^k_{\mathcal{H},\mathcal{F}}((b_{x_0,y})_{y\in\mathcal{F}})$$
- Output $(c_{x,y})_{(x,y)\in\mathcal{F}^2}$.

We say that a word $(a_{x,y})_{(x,y)\in\mathcal{F}^2}$ *represents* a polynomial $p(\vec{x})$ if the word is the evaluation of $p(\vec{x})$ at each point of $\mathcal{F}^2$. That is, if $a_{x,y} = p(x,y)$ for all $(x,y) \in \mathcal{F}^2$. If $(a_{x,y})_{(x,y)\in\mathcal{F}^2}$ is a word that differs from the representation of a maximum-degree d polynomial $p(x,y)$ in at most an ϵ fraction of its entries and $\epsilon < ((|F| - |H| - 1)/2\,|\mathcal{F}|)^2$, then $p(x,y)$ is the only maximum-degree d polynomial whose representation is this close to $(a_{x,y})_{(x,y)\in\mathcal{F}^2}$, so we say that $(a_{x,y})_{(x,y)\in\mathcal{F}^2}$ *ϵ-represents* $p(x,y)$.

We also define a more restrictive notion: a word $(a_{x,y})_{(x,y)\in\mathcal{F}^2}$ *ϵ-column represents* a polynomial $p(x,y)$ if $a_{x,y}$ agrees with $p(x,y)$ in all but an ϵ fraction of the values of x.

6. Computation with polynomials

The fundamental operation of our fault-tolerant circuits will be to take representations of two maximum-degree $|\mathcal{H}|-1$ polynomials, A and B, that correspond to states of a computation and an *operation* polynomial $\phi(a,b)$ and produce a representation of the maximum-degree $|\mathcal{H}|-1$ polynomial C that interpolates the values of $\phi(A(x,y),B(x,y))$ for $(x,y)\in\mathcal{H}^2$.

We begin by observing that if $\vec{a}$ and $\vec{b}$ represent A and B, and if ϕ has degree c, then $(\phi(a(x,y),b(x,y)))_{(x,y)\in\mathcal{H}^2}$ represents the maximum-degree $c(|\mathcal{H}|-1)$ polynomial $\phi(A,B)$. Moreover, if $\vec{a}$ and $\vec{b}$ ϵ-represent A and B, then $(\phi(a(x,y),b(x,y)))_{(x,y)\in\mathcal{H}^2}$ 2ϵ-represents the maximum-degree $c(|\mathcal{H}|-1)$ polynomial $\phi(A,B)$. However, we want the word that represents the maximum degree $|\mathcal{H}|-1$ polynomial that interpolates $\phi(A,B)$ through $(x,y)\in\mathcal{H}^2$. To obtain such a word from $(\phi(a(x,y),b(x,y)))_{(x,y)\in\mathcal{H}^2}$, we use a process called *degree reduction*. We first explain degree reduction in one variable. (note that degree reduction is only reasonable if $c(|\mathcal{H}|-1)<|\mathcal{F}|\,(1-2\epsilon)$, but we will be sure that this always holds whenever we apply degree reduction.)

Univariate degree reduction takes a β-representation of a degree $d\geq|\mathcal{H}|$ polynomial P and produces the unique polynomial Q of degree at most $|\mathcal{H}|-1$ that agrees with P on $\mathcal{H}$,

To state formally the univariate degree reduction function, we define the function $\pi_{\mathcal{H}}$ that takes a tuple of values indexed by a subset of $\mathcal{F}$ and projects onto those values indexed by elements of $\mathcal{H}$, $\pi_{\mathcal{H}}:(a_x)_{x\in\mathcal{F}}\to(a_x)_{x\in\mathcal{H}}$. Now, we can write the univariate degree reduction function $R^k_{d,\mathcal{H},\mathcal{F}}$ as a composition

$$R^k_{d,\mathcal{H},\mathcal{F}}((a_x)_{x\in\mathcal{F}})=E_{\mathcal{H},\mathcal{F}}(\pi_{\mathcal{H}}(D^k_{\mathcal{G},\mathcal{F}}((a_x)_{x\in\mathcal{F}}))),$$

where $\mathcal{G}$ is any subset of $\mathcal{F}$ of size $d-1$ such that $\mathcal{H}\subset\mathcal{G}$. (The reader might want to verify that this function is independent of the choice of $\mathcal{G}$.)

Bivariate degree reduction is analogous to univariate degree reduction in the same way that bivariate encoding and error-correction are analogous to univariate encoding and error-correction.

The idea of degree reduction appeared in Sudan's thesis [Sud92]. It's applicability to constructing small probabilistically checkable proofs was pointed out in [BF93]. Our application follows from the simple observation that their proofs can be constructed deterministically, rather than being checked non-deterministically.

Lemma 11 (one computation step). *Let $(a_{x,y})_{(x,y)\in\mathcal{H}^2}$ and $(b_{x,y})_{(x,y)\in\mathcal{H}^2}$ β-column represent maximum degree $(|\mathcal{H}|-1)$ polynomials A and B, respectively. Let ϕ be a degree c bivariate polynomial. Define $c_{x,y}$, $c^1_{x,y}$, and $c^2_{x,y}$ by the program:*

1 *For each $(x,y)\in\mathcal{F}^2$, compute $c_{x,y}=\phi(a_{x,y},b_{x,y})$*

2 *For each $y_0\in\mathcal{F}$, compute*

$$(c^1_{x,y_0})_{x\in\mathcal{F}}=R^k_{d,\mathcal{H},\mathcal{F}}((c_{x,y_0})_{x\in\mathcal{F}})$$

3 *For each $x_0\in\mathcal{F}$, compute*

$$(c^2_{x_0,y})_{y\in\mathcal{F}}=R^k_{d,\mathcal{H},\mathcal{F}}((c^1_{x_0,y})_{y\in\mathcal{F}})$$

4 *Output $(c^2_{x,y})_{(x,y)\in\mathcal{F}^2}$.*

Even if an ϵ fraction of the degree-reduction operations fail during each stage, $(c^2_{x,y})_{(x,y)\in\mathcal{F}^2}$ will β-column represent the maximum degree $|\mathcal{H}|-1$ polynomial that interpolates the values of $\phi(A,B)$ through $\mathcal{H}^2$, provided that

$$k>\max\{2\beta,\epsilon\}\,|\mathcal{F}| \qquad \textit{and} \qquad c\,|\mathcal{H}|<(1-\epsilon)\,|\mathcal{F}|$$

Proof: First note that $(c_{x,y})_{x,y\in\mathcal{F}^2}$ (2β)-column represents the polynomial $\phi(A,B)$. Because $k>2\beta\,|\mathcal{F}|$, $(c^1_{x,y_0})_{x\in\mathcal{F}}$ will differ from $R^d_{k,\mathcal{H},\mathcal{F}}(\phi(A(x,y_0),B(x,y_0))_{x\in\mathcal{F}})$ only if y_0 is one of the at most $\epsilon\,|\mathcal{F}|$ whose degree-reduction may fail. Because $k>\epsilon\,|\mathcal{F}|$, $(c^2_{x,y})_{(x,y)\in\mathcal{F}^2}$ will differ from the maximum degree $|\mathcal{H}|-1$ polynomial interpolating the values of $\phi(A,B)$ at $(x,y)\in\mathcal{F}^2$ only in those columns whose degree-reduction operation fails. □

We will write $R^d_{k,\mathcal{H}^2,\mathcal{F}}$ to denote the operation performed in stages 2 and 3 of the above computation.

7. Arithmetization of computation

To facilitate our arithmetization of computation, we will choose one type of computation to arithmetize and then observe that it can efficiently simulate any other computation of similar width. We will use algorithms that can be implemented on a hypercube in which each processor has a memory of a constant number of bits and, in each time step, each communication occurs in the same dimension.

In the n-dimensional hypercube, each processor is labeled by a string in $\{0,1\}^n$ and each processor is connected to those whose labels differ in just one dimension. Each processor is an identical finite automaton, independent of the size of the hypercube. The input to such a machine is a setting of the initial state of each processor. A program for such a machine is a list containing a communication direction in each time step, as well as an instruction for each processor at each time step. During a computation step, each processor updates its state according to its current state, its incoming instruction, and the state of its neighbor in the communication direction for that time step. Note that we insist that each processor communicate in the same direction in each time step, so that if processor $(0,0,0)$ communicates with processor $(1,0,0)$, then processor $(1,0,1)$ can only communicate with processor $(0,0,1)$. However, we allow a program to contain different instructions for distinct processors during the same time step. For more information on such models and proof of the following proposition, see [Lei92].

Proposition 12. *Any parallel machine with w processors can be simulated with polylogarithmic slowdown by a hypercube with $O(w)$ processors.*

To arithmetize hypercube computations, we choose $\mathcal{F}$ to be a field of the form $GF(2^\nu)$. Each element of $GF(2^\nu)$ can be naturally represented as a vector in $GF(2)^\nu$. We let $v_1,\ldots,v_\nu$ be the standard basis elements and let $\mathcal{H}$ be the space spanned by $\{v_1,\ldots,v_m\}$, where $m = n/2$ (assume $\nu > m$). We can now naturally identify each processor in the $\{0,1\}^n$ hypercube with an element of $\mathcal{H}^2$.

Since each processor is a finite automaton, we can identify the set of states of a processor with a finite set of the form $\mathcal{S} = GF(2^s)$, for some constant s. Hereafter, we insist that $\mathcal{S} \subset \mathcal{F}$, so that the state of the hypercube can be viewed as a word over $\mathcal{S} \subset \mathcal{F}$ indexed by elements of $\mathcal{H}^2$, say $\sigma_{(x,y)}$. As the reader may have by now guessed, this state will be stored in a fault-tolerant form as $E_{\mathcal{H}^2,\mathcal{F}}((\sigma_{x,y})_{(x,y)\in\mathcal{H}^2})$. Similarly, the list of instructions for each processor at a given time step can be viewed as a word over $\mathcal{S}$ indexed by elements of $\mathcal{H}^2$, and can be encoded similarly.

To arithmetize the communication of processors with their neighbors, we observe that for each communication direction there is a vector in $\mathcal{H}^2$ so that the label of the neighbor of a processor in that direction can be obtained by adding the vector to the label of that processor. For example, to swap states of processors in the first dimension, one need merely add $(v_1,0)$ to the name of each processor. Moreover, if $\vec{v} \in \mathcal{H}^2$, and

$$(\alpha_{\vec{x}})_{\vec{x}\in\mathcal{F}^2} = E_{\mathcal{H}^2,\mathcal{F}}((\sigma_{\vec{x}})_{\vec{x}\in\mathcal{F}^2}), \text{ then}$$

$$(\alpha_{\vec{x}+\vec{v}})_{\vec{x}\in\mathcal{F}^2} = E_{\mathcal{H}^2,\mathcal{F}}((\sigma_{\vec{x}+\vec{v}})_{\vec{x}\in\mathcal{F}^2}).$$

We can now arithmetize hypercube computations. This arithmetization is derived from the development in [PS94, Spi95] of ideas from [BFLS91]. While it may appear to be a minor consideration, the fact that the encoding of the permuted states can be obtained by permuting values in $\mathcal{F} \setminus \mathcal{H}$ is special to very few arithmetizations. One can use techniques from these papers to similarly arithmetize computations on shuffle-exchange and de Bruijn graphs.

Lemma 13. *There exist bivariate polynomials ϕ_1 and ϕ_2 of constant degree c such that for any parallel hypercube program that runs in time t on a $2m$-dimensional hypercube, there exists a sequence of* communication direction vectors $\vec{w}_1,\ldots,\vec{w}_t$ *in $\mathcal{H}^2$ and a sequence of* instruction words $W^1,\ldots,W^t$ *in $C_{\mathcal{H}^2,\mathcal{F}}$ such that for every input $(\sigma_1,\ldots,\sigma_{2^{2m}}) \in \mathcal{S}^{\mathcal{H}^2}$ to the hypercube program, the output of the hypercube program is the same as the output of:*

- *Let $(a^0_{\vec{x}})_{\vec{x}\in\mathcal{F}^2} = E_{\mathcal{H}^2,\mathcal{F}}(\sigma_1,\ldots,\sigma_{2^{2m}})$.*
- *For $i = 1$ to t,*

$$(b^i_{\vec{x}})_{\vec{x}\in\mathcal{F}^2} = R^{\sqrt{|\mathcal{F}|}}_{c(|\mathcal{H}|-1),\mathcal{H}^2,\mathcal{F}}\left(\phi_1(a^{i-1}_{\vec{x}}, a^{i-1}_{\vec{x}+\vec{v}_i})_{\vec{x}\in\mathcal{F}^2}\right),$$

$$(a^i_{\vec{x}})_{\vec{x}\in\mathcal{F}^2} = R^{\sqrt{|\mathcal{F}|}}_{c(|\mathcal{H}|-1),\mathcal{H}^2,\mathcal{F}}\left(\phi_2(b^i_{\vec{x}}, W^i_{\vec{x}})_{\vec{x}\in\mathcal{F}^2}\right).$$

- *Output $D_{\mathcal{H}^2,\mathcal{F}}\left(a^i_{\vec{x}}\right)_{\vec{x}\in\mathcal{F}^2}\big)$.*

Proof: Since the states that a processor can have as well as its set of instructions are identified with elements of a finite field $\mathcal{S}$, there are polynomials ϕ_1 and ϕ_2 over $\mathcal{S}$ such that, if σ is the state of a processor, σ' is the state of its neighbor in the communication direction, and η is the instruction for that processor, then $\phi_2(\phi_1(\sigma,\sigma'),\eta)$ will be the state of the processor after receiving instruction η and communicating with its neighbor.

Thus, the function of the above program is to produce a codeword that encodes the state of each processor in the hypercube algorithm at each time step. If one ignores the values of the codewords at points other than those corresponding to processors (*i.e.*, other than those in $\mathcal{H}^2$), then this becomes obvious. The superscript on R is irrelevant because we assume there are no errors in the above computation. □

By combining Lemmas 11 and 13 with Theorem 5, we obtain our main theorem

Theorem 14. *There exists a constant $\epsilon_P > 0$ and a deterministic construction that provides, for every parallel program M with w processors that runs for time t, a randomized parallel program M' that $(\epsilon_P, h \cdot 2^{-w^{1/4}}, E, D)$-simulates M and runs for time $t \log^{O(1)} w$ on $w \log^{O(1)} w$ processors, where E encodes the $O(\log^2 w)$-fold repetition of a generalized Reed-Solomon code of length $w \log^{O(1)} w$ and D can correct any $w^{-3/4}$ fraction of error in this code.*

Proof: By Proposition 12, M can be simulated by a n-dimensional hypercube algorithm with polylogarithmic slowdown, provided that $2^n > w$. We choose $\mathcal{F}$ to be the smallest field $GF(2^\nu)$ such that $\mathcal{S} \subset GF(2^\nu)$ and $2^\nu > c2^{n/2+2}$. By Lemma 13 and Lemma 9, there is an arithmetic program that computes the same function as M that can be computed by a parallel machine with $w \log^{O(1)} w$ processors that runs for time $t \log^{O(1)} w$. By Lemma 11, if the input to the program $1/\sqrt{|\mathcal{F}|}$-column represents $E_{\mathcal{H}^2,\mathcal{F}}(\sigma_1, \ldots, \sigma_{2^{2m}})$, and if at most a $1/\sqrt{|\mathcal{F}|}$ fraction of the univariate degree reduction operations fail during each of the t stages, then the output of each stage will $1/\sqrt{|\mathcal{F}|}$-column represent the output that would have appeared in the absence of faults.

As each univariate degree reduction operation is performed by $O(\sqrt{|\mathcal{F}|} \log^{O(1)} |\mathcal{F}|)$ processors that run for $\log^{O(1)} |\mathcal{F}|$ time, and the other operations all require many fewer processors, the machine can tolerate the failure of up to $w^{1/4}/(\log^{O(1)} w)$ processors at each time step.

We now apply Theorem 5 to this circuit, replacing each processor by a collection of r processors and inserting amplifiers. Applying a Chernoff bound, we see that we can choose $r = O(\log^2 w)$ so that the probability that more than $w^{1/4}/2^{O(\log^2 w)}$ collections of processors fail during any stage is at most $2^{-w^{1/4}}$. Thus, our encoding-decoding pair works with the r-fold repetition of $C_{\mathcal{H}^2,\mathcal{F}}$ and the probability that the simulation fails is at most $h \cdot 2^{-w^{1/4}}$. □

When we construct a parallel machine with w processors that runs for time t, we are really thinking of a circuit of width w and depth t. If we just concern ourselves with the size of the simulating circuit and ignore its depth, then we can get much lower error probabilities. Instead of Lemma 9, we can use Theorem 7 to perform univariate error-correction. As the later decoder is much better, each univariate operation can tolerate a constant fraction of error in its inputs. This enables us to arithmetize the computation in $\log \log w$ dimensions and prove the following theorem:

Theorem 15. *There exists a constant $\epsilon_P > 0$ and a deterministic construction that provides, for every circuit C of width w and height t, a circuit C' of size $tw \log^{O(1)} w$ that $(\epsilon_P, t \cdot 2^{w^{1-1/\log\log w}}, E, D)$-simulates C, where E and D encode and decode the $O(\log^2 w)$-fold repetition of generalized Reed-Solomon codes of length $w \log^{O(1)} w$.*

8. Application to Self-Correcting Programs

So far, the results we have presented have dealt with faults in individual wires. It is interesting to investigate whether similar techniques can be applied to protect against faults of larger modules within a system. We now present a simple example of how some of the ideas used in this paper can be combined with techniques for self-correcting linear functions to obtain a result of this form.

Blum, Luby, and Rubinfeld [BLR90] introduced self-testing/correcting programs to enable the reliable computation of a function with a device that usually computes the function, but is occasionally wrong. One drawback of the Blum-Luby-Rubinfeld self-correcting procedures is that they require multiple calls to the device for each reliable computation of the function. We observe that if one wants to compute many instances of a linear function with an unreliable device, then one can reliably compute all these instances of the function while making very few extra calls to the device (see [Rub92] for an application of related ideas to program checking).

We begin with a modification of an idea of Elias [Eli58]. Let (E, D) be an encoding-decoding pair for a linear error-correcting code of rate r such that D can correct a δ fraction of error. That is, there is an alphabet F such that $E : F^{rn} \to F^n$ is a linear function and if $w \in F^n$ is a word that differs from a word $E(x)$ in at most a δ fraction of its entries, then $D(w) = x$.

Now, let M be a linear function, let $x_1, \ldots, x_{rn}$ be instances on which we would like to compute M, and let P be an occasionally faulty device that usually computes M. To use P to compute $M(x_1), \ldots, M(x_{rn})$, we begin by computing $(y_1, \ldots, y_n) = E(x_1, \ldots, x_{rn})$ and applying P to each of $y_1, \ldots, y_n$. Because M and E are linear functions, $E(M(x_1), \ldots, M(x_{rn})) = (M(y_1), \ldots, M(y_n))$. Thus, if P produces the correct output on at least $n - \delta n$ of the y_i's, then $D(P(y_1), \ldots, y_n) = (M(x_1), \ldots, M(x_{rn}))$.

For self-correction in the sense of [BLR90], it is necessary to make the instances $y_1, \ldots, y_n$ randomly dis-

tributed. This can be done by computing M on a few less instances and adding a few randomly chosen codewords to the vector $y_1, \ldots, y_n$.

For example, if we have a program P that computes a linear function M on all but an ϵ fraction of its inputs, we can use polynomial codes to compute m instances of M while making only $m + 3k + 1$ queries to P with a probability of error at most $\epsilon^k \binom{m+3k+1}{k}$. Let $\mathcal{F}$ have size $m+3k+1$. The m instances, $(a_1, \ldots, a_m)$, will be identified with, $\mathcal{A}$, a subset of $\mathcal{F}$ of size m. We will then choose an additional k problem instances $b_1, \ldots, b_k$ uniformly at random. These will be identified with a set $\mathcal{B} \subset \mathcal{F}$ of size k. Note that the elements of $\vec{r} = E_{\mathcal{B},\mathcal{F}}(b_1, \ldots, b_k)$ are a set of $m+3k+1$ k-wise independent random variables.

Now, let $\vec{q} = E_{\mathcal{A}\cup\mathcal{B},\mathcal{F}}(a_1, \ldots, a_m, 0, \ldots, 0)$, by which we mean to interpolate the degree $|\mathcal{A}| + |\mathcal{B}| - 1$ polynomial that has values $(a_1, \ldots, a_m)$ at $\mathcal{A}$ and is zero at $\mathcal{B}$. Because $\vec{r}$ is a representation of a degree $|\mathcal{B}| - 1$ polynomial, $\vec{q} + \vec{r}$ is a codeword of $C_{\mathcal{A}\cup\mathcal{B},\mathcal{F}}$, and the result of applying M to each element of $\vec{q} + \vec{r}$ will be as well. We now apply P to each element of $\vec{q} + \vec{r}$. If P returns the wrong answer on fewer than k instances, then the decoding algorithm applied to P's answers will return the correct evaluation of M at each element of $\vec{q}+\vec{r}$. To obtain the values of M at $a_1, \ldots, a_m$, it only remains to subtract off the vector $E_{\mathcal{B},\mathcal{F}}(M(b_1), \ldots, M(b_k))$.

Since the elements of $\vec{q} + \vec{r}$ are k-wise independent, the probability that P returns the incorrect answer on more than k instances is at most $\epsilon^k \binom{m+3k+1}{k}$.

9. Directions for further work

Greater fault tolerance: If there were a processor-efficient polylog depth circuit for decoding Reed-Solomon codes, then it would be possible for parallel machines to achieve the fault tolerance achieved by circuits in Theorem 15. Even if one can achieve this, it would remain open whether one can construct parallel machines with w processors that fail with probability only $2^{-w/\log^{O(1)} w}$ or even $2^{-\Omega(w)}$. A machine with the latter error probability would necessarily be resistant to a constant fraction of faults chosen by an adversary.

Constant blow-up: Taylor [Tay68] used Gallager's low-density parity check codes to construct fault-tolerant memories. His construction was improved by Kuznetsov [Kuz73] to obtain stable memories that store n bits using $O(n)$ wires and tolerate a constant rate of error with probability $2^{-O(n)}$. Kuznetsov's construction was probabilistic, but can be made deterministic using techniques from [SS96]. Thus, a natural way to try to construct fault-tolerant circuits with constant blow up would be to find a way to modify these codes so that they allow computation.

Connection patterns of fault-tolerant circuits: The fault-tolerant circuits that we construct cannot be embedded well in the three dimensional space we occupy. This is in contrast with the results of Gács [Gác86] that enable fault-tolerant computation with cellular automata. We wonder whether connectivity patterns as complex as ours are necessary to obtain our high degree of fault-tolerance.

Using algebraic geometry codes: Using algebraic geometry, one can construct asymptotically-good error-correcting codes that have the same multiplicative property as Reed-Solomon codes: if one performs component-wise multiplication on codewords of low rate, then one obtains a codeword in a larger error-correcting code. One can use such codes in place of Reed-Solomon codes in our constructions. One should also be able to use them in constructions of probabilistically checkable proofs in place of the polynomial codes currently used. The advantage of this is that the alphabet size of algebraic geometry codes is constant, while it must grow for polynomial codes.

Fault-tolerant simulation of all circuits: We ask whether all circuits have an efficient fault-tolerant simulation. We conjecture that the answer to this question is "no". We suspect that there are circuits whose width times height is much greater than their size and which cannot be made fault-tolerant efficiently in our model. However, we would prefer to be surprised.

Quantum Computation: Can ideas from this paper used to compensate for decoherence in quantum computations?

Other applications: We have shown that parallel computation can assume a rather unusual form. Such computations may very well have other applications. We are optimistic because fault-tolerance can be used as a metaphor for many other computational concepts.

10. Acknowledgements

I would like to thank Amin Shokrollahi for helpful discussions of error-correcting codes and Manuel Blum, Torsten Suel, and Umesh Vazirani for other inspiring conversations.

References

[Ahl84] R. Ahlswede. Improvements of Winograd's results

on computation in the presense of noise. *IEEE Transactions on Information Theory*, 30:872–877, 1984.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley series in computer science and information processing. Addison-Wesley, Reading, Massachusetts, 1974.

[BCS96] P. Bürgisser, M. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory.* Springer Verlag, 1996. to appear.

[BF93] L. Babai and K. Friedl. On slightly superlinear transparent proofs. CS 93-13, The University of Chicago, Department of Computer Science, Chicago, IL, 1993.

[BFL91] L. Babai, L. Fortnow, and C. Lund. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BFLS91] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proc. of the 23rd ACM STOC*, pages 21–31, 1991.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. of the 22nd ACM STOC*, pages 73–83, 1990.

[DO77] R. L. Dobrushin and S. I. Ortyukov. Upper bound for the redundancy of self-correcting arrangements of unreliable functional elements. *Problems Inform. Transmission*, 13:203–218, 1977.

[Eli58] P. Elias. Computation in the presense of noise. *IBM J. Res. Develop.*, 2:346–353, 1958.

[Gác86] P. Gács. Reliable computation with cellular automata. *J. Comput. Syst. Sci.*, 32(1):15–78, February 1986.

[Gál91] A. Gál. Lower bounds for the complexity of reliable Boolean circuits with noisy gates. In *Proc. of the 32nd IEEE FOCS*, pages 594–601, 1991.

[GS95] A. Gál and M. Szegedy. Fault tolerant circuits and probabilistically checkable proofs. In *Proceedings of the 10th IEEE Structure in Complexity Theory Conference*, pages 65–73, 1995.

[JáJ92] J. JáJá. *An introduction to parallel algorithms.* Addison Wesley, 1992.

[Jus76] J. Justesen. On the complexity of decoding Reed-Solomon codes. *IEEE Transactions on Information Theory*, 22(2):237–238, March 1976.

[KP94] E. Kaltofen and V. Pan. Parallel solution of toeplitz and toeplitz-like linear systems over fields of small positive characteristic. In *Proc. 1st Internat. Symp. Parallel Symbolic Comput.*, pages 225–233. World Scientific Publ. Co., 1994.

[Kuz73] A. V. Kuznetsov. Information storage in a memory assembled from unreliable components. *Problems of Information Transmission*, 9(3):254–264, 1973.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Archictectures.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1992.

[MS77] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes.* North Holland, Amsterdam, 1977.

[Pip85] N. Pippenger. On networks of noisy gates. In *Proceedings of the 26th Ann. IEEE Symposium on Foundations of Computer Science (Portland, OR)*, pages 30–38. IEEE, IEEE, 1985.

[Pip89] N. Pippenger. Invariance of complexity measures for networks with unreliable gates. *J. ACM*, 36(3):531–539, July 1989.

[Pip90] N. Pippenger. Developments in "the synthesis of reliable organisms from unreliable components". In *Proceedings of Symposia in Pure Mathematics*, volume 50, pages 311–324, 1990.

[PS94] A. Polishchuk and D. A. Spielman. Nearly linear-size holographic proofs. In *Proc. of the 26th ACM STOC*, pages 194–203, 1994.

[RS91] R. Reischuk and B. Schmeltz. Reliable computation with noisy circuits and decision trees—a general $n \log n$ lower bound. In *Proc. of the 32nd IEEE FOCS*, pages 602–611, 1991.

[Rub92] R. Rubinfeld. Batch checking with applications to linear functions. *Information Processing Letters*, 42:77–80, May 1992.

[Sar77] D. V. Sarwate. On the complexity of decoding Goppa codes. *IEEE Transactions on Information Theory*, 23(4):515–516, July 1977.

[Spi95] D. A. Spielman. *Computationally efficient error-correcting codes and holographic proofs.* PhD thesis, M.I.T., May 1995. Available at `http://theory.lcs.mit.edu/~spielman`.

[SS96] M. Sipser and D. A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 1996. to appear.

[Str83] V. Strassen. The computational complexity of continued fractions. *SIAM J. Comput.*, 12(1):1–27, February 1983.

[Sud92] M. Sudan. *Efficient checking of polynomials and proofs and the hardness of approximation problems.* PhD thesis, U.C. Berkeley, Oct. 1992.

[Tay68] M. G. Taylor. Reliable informatoin storage in memories designed from unreliable components. *Bell System Technical Journal*, 47:2299–2337, 1968.

[vL92] J. H. van Lint. *Introduction to Coding Theory.* Springer-Verlag, 1992.

[vN56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies.* Princeton University Press, 1956.

[Win62] S. Winograd. Coding for logical operations. *IBM J. Res. Develop.*, 6:430–436, 1962.