On Dynamic Algorithms for Algebraic Problems

By: John H. Reif and Stephen R. Tate

J. H. Reif and S. R. Tate. "On Dynamic Algorithms for Algebraic Problems", Journal of Algorithms, Vol. 22, No. 2, 1997, pp. 347–371.

Abstract:

In this paper, we examine the problem of incrementally evaluating algebraic functions. In particular, if $f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$ is an algebraic problem, we consider answering on-line requests of the form "change input x_i to value v'' or "what is the value of output y_j?'' We first present lower bounds for some simply stated algebraic problems such as multipoint polynomial evaluation, polynomial reciprocal, and extended polynomial GCD, proving an $\Omega(n)$. lower bound for the incremental evaluation of these functions. In addition, we prove two time-space trade-off theorems that apply to incremental algorithms for almost all algebraic functions. We then derive several general-purpose algorithm design techniques and apply them to several fundamental algebraic problems. For example, we give an $O(\sqrt{n})$ time per request algorithm for incremental DFT. We also present a design technique for serving incremental requests using a parallel machine, giving a choice of either optimal work with respect to the sequential incremental algorithm or superfast algorithms with $O(\log \log n)$ time per request with a sublinear number of processors.

Article:

1. INTRODUCTION

In this paper we examine the problem of designing incremental algorithms for algebraic problems. In particular, let $\mathcal{R} = (S, +, \cdot, 0, 1)$ be a ring with elements from a set S and appropriately defined addition and multiplication operations, and let $f: S^n \rightarrow S^m$ be an algebraic function over this ring. Given an initial input vector $(x_1, x_2, \ldots, x_n)$ for this function, an incremental algorithm for the function is allowed some preprocessing time using the initial values of the function and then must quickly handle on-line requests. The requests are in one of two forms: either ''change input k to new value x'_k'' or ''what is the value of output k?'' Requests of both types are mixed in the request stream, and we would like a fast guaranteed worst-case response time. The machine servicing the requests may be either a sequential or parallel machine. While incremental versions of many graph problems (for example [9, 11, 20]) and geometry problems (for example [1, 6, 21]) have been studied, very little has been done in incremental versions of algebraic problems. Two notable exceptions are the incremental maintenance of prefix sums studied by Fredman w10] and the incremental maintenance and evaluation of size n algebraic expressions studied by Frederickson [12].

There are many real-world situations in which such incremental alge-braic problems may arise, where the inputs change relatively slowly and various outputs need to be occasionally polled. Applications include real-time control (e.g., kinodynamic control of machines and robots), signal processing (e.g., low-level image processing, signal tracking, and dynamic error distortion control), data compression (the 2D transform methods), econometrics (analysis of time series), and business processing (incremental updates of numerical spread sheets). In all these applications, the input data often changes dynamically. The data comes as a continuous stream of updates, interspersed with requests for specified algebraic computations determined from the most recent version of the data. Moreover, there are often real-time constraints that make it essential to have a fast, efficient response to these incremental changes. As an interesting specific example, we note that in image or voice compression by transform methods, many of the outputs of an algebraic transform maybe ignored. For

example, JPEG, now a standard for image compression, requires as its key computation a two-dimensional discrete Fourier transform (DFT), which is defined as a mapping from an $n \times n$ matrix of pixel intensity values $A = (a_{ij})$ to an $n \times n$ matrix of transform coefficients $C = (c_{ij})$ by

$$c_{ij} = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} a_{ij} \omega_n^{ix+jy},$$

where ω_n is the nth complex root of unity. To browse this image (at low resolution), we need only a small portion of this two-dimensional DFT. The incremental version of this problem arises naturally when video images (consisting of sequences of slowly changing images) compressed by the JPEG transform are browsed at low resolution. An additional problem from the domain of signal processing that we examine in this paper is the ''chirp z-transform,'' which transforms a vector $A = (a_0, a_1, \ldots, a_{n-1})$ of values to a vector of coefficients $C = (c_0, c_1, \ldots, c_{n-1})$ by

$$c_i = \sum_{j=0}^{n-1} a_j z^{ij},$$

where z is an arbitrary complex number. Clearly, the standard one-dimensional discrete Fourier transform is a special case of the chirp z-transform.

The most basic approach to this problem is simply to re-evaluate the entire function each time an output value is requested. For example, if the function in question is a discrete Fourier transform (DFT), we can easily handle each request on a sequential machine in $O(n \log n)$ time by performing a fast Fourier transform each time an output value is requested. Since a change in a single input value affects all n output values, it is not immediately obvious how to do better. In this paper, we first present lower bounds that show that many basic algebraic problems cannot be solved substantially faster than using this naive ''reevaluate every time'' approach. In particular, we prove an $\Omega(n)$ lower bound for the problems of multipoint polynomial evaluation, polynomial reciprocal, triangular Toeplitz system solve, and extended polynomial GCD. In addition, we also prove a space—time trade-off that applies to most interesting algebraic problems: if $S(n)$ storage is available in addition to the input storage, then any problem in which at least one output is dependent on all inputs will require $\Omega(n/S(n))$ time per request. To prove this trade-off for some of the problems, we require the use of a slightly nonstandard algebra (which we call the ∞-augmented algebra) which, in fact, is closely imitated by the way floating point numbers are implemented on many real machines. See Table 1 for a summary of lower bounds proved in this paper.

TABLE 1
Incremental Time for the Problems Considered in This Paper

Problem	Incremental time	Lower bound
Prefix sum	$O(\log n)$	$\Omega(n/S(n))^a$
Solution of order ω linear recurrence	$O(M(\omega)\log n)$	$\Omega(n/S(n))^a$
Discrete Fourier transform	$O(\sqrt{n})$	$\Omega(n/S(n))$
Chirp z-transform	$O(\sqrt{n \log n})$	$\Omega(n/S(n))$
Two-dimensional DFT	$O(\sqrt{n \log n})$	$\Omega(n/S(n))$
Polynomial multiplication/convolution	$O(\sqrt{n \log n})$	$\Omega(n/S(n))$
Multipoint polynomial evaluation (changing coeff.)	$O(\sqrt{n} \log n)$	$\Omega(n/S(n))$
Multipoint polynomial evaluation (changing points)	$O(n)$	$\Omega(n)$
Matrix–vector product	$O(n)$	$\Omega(n^2/S(n))$
Matrix–matrix product	$O(\sqrt{M(n)})$	$\Omega(n^2/S(n))$
Restricted change linear system solve	$O(n)$	$\Omega(n/S(n))^a$
Restricted change Toeplitz system solve	$O(\sqrt{n \log n})$	$\Omega(n/S(n))^a$
Polynomial reciprocal	$O(n \log n)$	$\Omega(n)$
Extended polynomial GCD	$O(n \log^2 n)$	$\Omega(n)$

[a] Trade-off for problems over an ∞-augmented algebra

Second, we present two general methods for designing incremental algorithms on sequential machines. Both methods make use of existing algebraic circuits for the nonincremental versions of the problems. Using the first

technique, we derive an incremental algorithm for the DFT where each request is handled in $O(\sqrt{n})$ time. The second technique gives slightly worse time bounds for some problems $O(\sqrt{n \log n})$ time for DFT, for instance), but is applicable to a much wider range of problems. In particular, we derive $O(\sqrt{n}\ \log^{O(1)} n)$ time incremental algorithms for multipoint polynomial evaluation with changing coefficients, polynomial multiplication, sequence convolution, and various matrix problems. The best results for the problems examined in this paper are shown in Table 1.

Finally, we extend this last sequential technique to a model where requests are handled by a parallel machine. We give a general theorem and show how it can be applied to various algebraic problems. In particular, we show that the discrete Fourier transform can be handled with an optimal work parallel algorithm when the number of processors is at most $O(\sqrt{n \log n})$. In addition, by using more processors we can obtain super-fast parallel algorithms that use $O(\log \log n)$ time per request and $O(n/\log^c n)$ processors for any constant c. Similar bounds hold for other problems, including multipoint polynomial evaluation and the chirp z-transform.

It should be noted that all of our algorithms give worst-case service times. Some previous incremental algorithms for problems in other areas use the technique of amortized analysis to give bounds for an entire request sequence. While amortized analysis may be acceptable in some cases, our worst-case analysis is far more useful in application areas involving real-time processing.

1. 1. *Algebraic Problems and Circuits*

The problems we examine will be problems over an arbitrary algebraic ring $\mathcal{R} = (S, +, \cdot, 0, 1)$. In the construction of our incremental algorithms we make use of efficient algebraic circuits for the nonincremental versions of the problems. For example, in constructing an incremental algorithm for FFT, we use the well-known algebraic circuit for FFT that has size $O(n \log n)$ [2, 7]. To make clear the form we would like the circuit to be in, we use the following definition of a circuit.

DEFINITION 1. 1. An *algebraic circuit over a ring* $\mathcal{R}$ is a circuit in which each noninput node performs some basic algebraic operation (either addition or multiplication) from the ring $\mathcal{R}$. We call such a circuit simply an *algebraic circuit* if the ring $\mathcal{R}$ is understood or unimportant. The nodes of the circuit $v_1, v_2, \ldots, v_{\sigma(n)}$ are ordered so that $v_1, \ldots, v_n$ are the input nodes, and if the node v_k has inputs v_i and v_j, then $i < k$ and $j < k$. The number of nodes $\sigma(n)$ is called the *size* of the circuit.

The restriction on the numbering of the nodes is simply a matter of circuit representation--any circuit can reorder its nodes so that the constraint on the node order holds. Since a circuit is a directed acyclic graph, to determine such an ordering it suffices to perform a topological sort (a linear time operation) on the circuit. The important property of this ordering is that we can evaluate the circuit in time $\sigma(n)$ on an algebraic RAM: simply calculate the value at each node in the order that the nodes are given. The order ensures that when node v_k is evaluated, all required values have already been computed.[1]

2. LOWER BOUNDS

In this section, we derive lower bounds on the complexity of incremental algorithms for some algebraic problems, including polynomial reciprocal and multipoint polynomial evaluation when the evaluation points change. In both cases we prove a linear time lower bound. We also prove a space–time trade-off that applies to most algebraic problems. The basis of our first lower bound results is the following theorem which follows directly from results of Motzkin [17] and Belaga [3] (for the result as stated here, see [2, Theorem 12.4] or [5, Theorem 3.2.5]).

THEOREM 2.1 [3, 17]. *A computation for polynomial evaluation requires at least* $n/2$ *multiplications to evaluate an arbitrary nth degree polynomial at a single point, even if multiplications in computations involving only coefficients are not counted.*

This theorem can be immediately applied to prove a linear time lower bound for incremental polynomial evaluation when the evaluation points are allowed to change. In particular, consider an arbitrarily complex preprocessing phase consisting of computations involving the coefficients of the polynomial. Now consider changing an input evaluation point to an arbitrary new value, and then polling the output corresponding to the new evaluation point. Theorem 2.1 now implies that $n/2$ multiplications are required to perform these two operations—the following corollary is thus a reinterpretation of Theorem 2.1 from the point of view of incremental algorithms (note that one output from a chirp z-transform is simply a polynomial evaluation at a point z^k for some integer k).

COROLLARY 2.1 [3,17]. *Any incremental algorithm for multipoint polynomial evaluation where evaluation points are allowed to change requires $\Omega(n)$ time per request. Likewise, any incremental algorithm for the chirp z-transform where z is allowed to change requires $\Omega(n)$ time.*

It is worth noting that there is a simple incremental algorithm for this problem that has request service time $O(n)$—we simply re-evaluate the polynomial at the appropriate point when an output is requested, which can be done in an asymptotically optimal $n/2 + o(n)$ multiplications if the polynomial coefficients are constant [19].

Proving a linear lower bound for the polynomial reciprocal problem uses the same idea, but requires a substantially more complex proof. More specifically, if $p(x) = \sum_{i=0}^{n-1} p_i x^i$ is a degree n - 1 polynomial and

$$\sum_{i=0}^{n-1} q_i x^i = q(x) = \left\lfloor \frac{x^{2n-2}}{p(x)} \right\rfloor$$

is the reciprocal of $p(x)$, then q_0 (the constant term of $q(x)$) can be viewed as a degree n polynomial $h_0(x)$ that is evaluated at $1/p_{n-1}$. The coefficients of $h_0(x)$ depend only on $p_{n-2}, p_{n-3},\ldots,p_1,p_0$. More importantly, by selecting $p_{n-2}, p_{n-3},\ldots,p_1,p_0$ properly, we can make $h_0(x)$be *any* arbitrary degree n polynomial in which the degree 1 and degree 0 terms are zero. Changing p_{n-1} and polling output q_0 is exactly the same as evaluating $h_0(x)$ at $1/p_{n-1}$ thus requires $\Omega(n)$ operations. The details of this proof follow.

LEMMA 2. 1. *Let $y = 1/p_{n-1}$. Then $q(x)$ can be written as*

$$q(x) = \sum_{k=0}^{n-1} h_k(y) x^k,$$

where $h_k(y)$ is a degree n - k polynomial in y, and the polynomial coefficients of $h_k(y)$ do not depend on p_{n-1}. If the coefficients are denoted by

$$h_k(y) = \sum_{i=0}^{n-k} h_{i,k} y^i,$$

then

$$h_{i,k} = \begin{cases} -\displaystyle\sum_{j=2}^{n-k-i+2} p_{n-j} h_{i-1,k+j-1} & \text{if } 0 \le k \le n-2 \text{ and } 2 \le i \le n-k, \\ 1 & \text{if } i = 1 \text{ and } k = n-1, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Proof. By considering the long division method, we can see that the coefficients of $q(x)$ can be described by

$$q_k = \begin{cases} \dfrac{1}{p_{n-1}} & \text{if } k = n-1, \\ -\dfrac{1}{p_{n-1}} \displaystyle\sum_{j=2}^{n-k} p_{n-j} q_{k+j-1} & \text{if } 0 \le k \le n-2. \end{cases}$$

Rewriting this in terms of the $h_k(y)$ polynomials, we get

$$h_k(y) = \begin{cases} y & \text{if } k = n-1, \\ -y \sum_{j=2}^{n-k} p_{n-j} h_{k+j-1}(y) & \text{if } 0 \le k \le n-2. \end{cases}$$

Examination of these equations shows that $h_k(y)$ has degree $n - k$ and that all $h_k(y)$ polynomials have no degree 1 term except for $h_{n-1}(y)$. All the statements in the theorem follow immediately except the first case in (1). Some fairly simple algebra proves this final statement. In the first equality below, we make use of the fact that $h_{0,k} = 0$ for all k:

$$\begin{aligned} h_k(y) &= -y \sum_{j=2}^{n-k} p_{n-j} \sum_{i=1}^{n-k-j+1} h_{i,k+j-1} y^i \\ &= -\sum_{j=2}^{n-k} \sum_{i=1}^{n-k-j+1} p_{n-j} h_{i,k+j-1} y^{i+1} \\ &= -\sum_{i=1}^{n-k-1} \sum_{j=2}^{n-k-i+1} p_{n-j} h_{i,k+j-1} y^{i+1} \\ &= -\sum_{i=2}^{n-k} \sum_{j=2}^{n-k-i+2} p_{n-j} h_{i-1,k+j-1} y^i. \end{aligned}$$

From this last equation, we can immediately read off the $h_{i,k}$ term, proving the $i \ge 2$ case of **(1)**.

Next we prove two lemmas that give a closed form expression for the $h_{i,k}$ coefficients.

LEMMA 2.2. *For all i satisfying* $1 \le i \le n$,

$$h_{i,n-i} = (-p_{n-2})^{i-1}.$$

Proof. The proof is by induction in i, using the recurrence equations from Lemma 2.1. For $i = 1$, the lemma is trivially true. For $i > 1$, we see that

$$h_{i,n-1} = -\sum_{j=2}^{2} p_{n-j} h_{i-1,n-i+j-1} = -p_{n-2} h_{i-1,n-(i-1)}.$$

Using our inductive hypothesis, we get

$$-p_{n-2} h_{i-1,n-(i-1)} = -p_{n-2}(-p_{n-2})^{i-2} = (-p_{n-2})^{i-1},$$

which completes the proof of the lemma.

LEMMA 2.3. *If* $i + k < n$, *then*

$$h_{i,k} = -(i-1)(-p_{n-2})^{i-2} p_{i+k-2} + f_{i,k}(p_{i+k-1}, \ldots, p_{n-2}),$$

where $f_{i,k}$ *is some function that depends only on* i *and* k.

Proof. Again, we use a proof by induction on i. For $i = 1$, the lemma is easily seen to be true. For $i > 1$,

$$\begin{aligned} h_{i,k} &= -\sum_{j=2}^{n-k-i+2} p_{n-j} h_{i-1,k+j-1} \\ &= -p_{n-2} h_{i-1,k+1} - \sum_{j=3}^{n-k-i+1} p_{n-j} h_{i-1,k+j-1} - p_{k+i-2} h_{i-1,n-i+1}. \end{aligned}$$

Of the three parts of this equation, the inductive hypothesis can be applied to the first two terms, and Lemma 2.2 can be used for the last term, giving the rather large expression

$$\begin{aligned} &p_{n-2}(i-2)(-p_{n-2})^{i-3} p_{i+k-2} - p_{n-2} f_{i-1,k+1}(p_{i+k-1}, \ldots, p_{n-2}) \\ &\quad - \sum_{j=3}^{n-k-i+1} p_{n-j} \big[-(i-2)(-p_{n-2})^{i-3} \\ &\qquad \times p_{i+k+j-4} + f_{i-1,k+j-1}(p_{i+k+j-3}, \ldots, p_{n-2}) \big] \\ &\quad - p_{i+k-2}(-p_{n-2})^{i-2}. \end{aligned}$$

Most of these terms depend only on coefficients $p_{i+k-1},\ldots,p_{n-2}$, so combining these terms under the function $\tilde{f}(p_{i+k-1}, \ldots, p_{n-2})$, we can simplify notation considerably to give

$$\begin{aligned} & p_{n-2}(i-2)(-p_{n-2})^{i-3}p_{i+k-2} - p_{i+k-2}(-p_{n-2})^{i-2} \\ & \quad + \tilde{f}(p_{i+k-1},\ldots,p_{n-2}) \\ & = -(i-1)(-p_{n-2})^{i-2}p_{i+k-2} + \tilde{f}(p_{i+k-1},\ldots,p_{n-2}). \end{aligned}$$

By identifying $f_{i,k}$ with $\tilde{f}$, we have proved the lemma.

Finally, we use the last two lemmas to prove that we can make the constant term of $q(x)$. be (almost) any arbitrary nth degree polynomial in $1/p_{n-1}$, so we can apply Theorem 2. 1.

LEMMA 2.4. *Given any degree n - 2 polynomial A*(*x*), *there exists a degree n - 1 polynomial p*(*x*) *such that* $h0(y) = y^2(y)$.

Proof. If $A(y) = \sum_{i=0}^{n-2} a_i y^i$, then we want to find p_i's such that $h_{i+2,0} = a_i$ for all $0 \le i \le n - 2$. By Lemma 2.2, we can select

$$p_{n-2} = -(h_{n,0})^{1/(n-1)} = -(a_{n-2})^{1/(n-1)},$$

and we will next find p_{n-3}, p_{n-4}, ... in that order. By Lemma 2.3, each $h_{i,0}$ is a linear function in p_{i-2}, which can be inverted easily. In particular,

$$p_i = -\frac{1}{(i+1)(-p_{n-2})^i}\left[a_i - f_{i+2,0}(p_{i+1},\ldots,p_{n-2})\right].$$

Since $p_{n-2} \neq 0$, this clearly gives a procedure for finding the p_i's that satisfy the lemma.

Our main theorem is now an immediate consequence of Theorem 2.1 and Lemma 2.4.

THEOREM2.2. *Any incremental algorithm for polynomial reciprocal must perform* (1/2)(*n*/2-1) *multiplications per request in the worst case, thus must take* $\Omega(n)$ *time per request*.

Many algebraic problems can be reduced easily to the polynomial reciprocal problem, so lower bounds easily follow. As a simple example, consider solving a triangular Toeplitz system of equations. It is well known that this problem is equivalent to finding polynomial reciprocals, giving the following corollary.

COROLLARY 2.2. *Any incremental algorithm for solving a triangular Toeplitz system of equations must take* $\Omega(n)$ *time per request*.

For a less simple example, consider the extended GCD problem [4]. That is, given two polynomials $A(x)$ and $B(x)$, find the two unique polynomials $U(x)$ and $V(x)$ such that $U(x)A(x) + V(x)B(x) = G(x)$, where $G(x)$ is the GCD of $A(x)$ and $B(x)$ and $\deg[U(x)] < \deg[B(x)] - \deg[G(x)]$.

COROLLARY 2.3. *Any incremental algorithm for the extended GCD problem on polynomials must take* $\Omega(n)$. *time per request*.

Proof. If $p(x) = \sum_{i=0}^{n-1} p_i x^i$ is a degree n - 1 polynomial, let $p_{\text{rev}}(x) = x^{n-1}p(1/x)$. denote the polynomial with coefficients in the reverse order. In other words, $p_{\text{rev}}(x) = \sum_{i=0}^{n-1} p_{n-1-i}x^i$. It is easy to see that if $p(x)$ has a nonzero degree n - 1 term, then $p_{\text{rev}}(x)$ and x^n are relatively prime. Solving the extended GCD problem for these two polynomials gives polynomials $U(x)$ and $V(x)$ such that

$$U(x)x^n + V(x)p_{\text{rev}}(x) = 1.$$

We can derive, for formal variable $y = 1/x$,

$$U\left(\frac{1}{y}\right)\left(\frac{1}{y}\right)^n + V\left(\frac{1}{y}\right)p_{\text{rev}}\left(\frac{1}{y}\right) = 1$$

$$\Leftrightarrow y^{2n-2}V\left(\frac{1}{y}\right)p_{\text{rev}}\left(\frac{1}{y}\right) = y^{2n-2} - y^{n-2}U\left(\frac{1}{y}\right)$$

$$\Leftrightarrow V_{\text{rev}}(y)p(y) = y^{2n-2} - y^{n-2}U\left(\frac{1}{y}\right).$$

We know that $\deg[U(x)] < \deg[p_{\text{rev}}(x)] \leq n - 1$, so it follows that $y^{n-2}U(1/y)$ is a polynomial with $\deg[y^{n-2}U(1/y)] < n - 1$, which shows that $V_{\text{rev}}(x)$ is the polynomial reciprocal of $p(x)$. Therefore, any incremental algorithm for extended GCD can trivially be converted into an incremental algorithm for polynomial reciprocal, so any incremental algorithm for extended GCD must take $\Omega(n)$ time per request.

2.1. *Space—Time Trade-Offs*

In this section, we consider space—time trade-offs for incremental algebraic algorithms. First we consider circuits working over a modified algebra. In particular, if we are considering a ring $\mathcal{R} = (S, +, \cdot, 0, 1.$ then we add the special ''infinity'' element ∞ that has the property $x + \infty = \infty$ and $x \cdot \infty = \infty$ for any element $x \in S$. We note that such an algebra cannot be a ring, but when considering actual floating point computer implementations, such infinite elements often do indeed exist. We will call such an algebra an ∞-*augmented algebra*. Our main space—time trade-off result can then be stated as follows.

THEOREM2.3. *Let $f(x_1, x_2, \ldots, x_n)$ be an algebraic function in which at least one output value is dependent on all input values. Then any algorithm that uses $S(n)$ storage in addition to the inputs and correctly computes $f(x_1, x_2, \ldots, x_n)$ over an ∞-augmented algebra must take $\Omega(n/S(n))$ time per request.*

Proof. The outputs and temporary values of any algebraic algorithm can be viewed as multivariate polynomials in the inputs. In particular, if $b_1, b_2, \ldots, b_{S(n)}$ are the $S(n)$. additional stored values, then each b_i can be viewed as a polynomial in the inputs. Since we have augmented our algebra with ∞, any nonconstant polynomial can be set to ∞ by setting a single appropriate variable to ∞. In other words, we can make $b_i = \infty$ values for *all* $1 \leq i \leq S(n)$ by setting at most $S(n)$ of the inputs to ∞.

In effect, we have now rendered the extra storage useless. We can set the remaining n — $S(n)$ inputs to any values that we want, and then substitute values for the $S(n)$ inputs that were set to ∞. These changes effectively define a new size n problem, in which useful partial computations with inputs can be performed only during the last $S(n)$ changes. By next requesting an output value that depends on all n input values, we must have examined all n inputs in the last $S(n) + 1$ requests in order to compute the value, so some query must have taken at least $n/(S(n) + 1) = \Omega(n/S(n))$ time.

We note here that for certain problems, a space—time trade-off can be obtained without having to resort to the ∞-augmented algebra. This result follows from Tompa's work on space—time trade-offs for standard (nonincremental) algorithms, in which he shows that straight-line programs for polynomial product, convolution, or DFT that use $S(n)$ space require $\Omega(n^2 / S(n))$ time [22]. In addition, Tompa proves a more general result about space—time trade-offs for general linear functions. Any set of n linear functions in n indeterminates can be represented as a product Ax, where x is the vector of n indeterminates. Tompa shows that if all of the minors of A are nonsingular, then computing Ax with a straight-line program that uses $S(n)$ space requires $\Omega(n^2 / S(n))$ time [22, Corollary 1]. While we cannot use this for our most basic problem of prefix sum, we use Tompa's results to obtain the following theorem. We call an incremental algorithm *oblivious* if the actions taken at a request depend only on the position of the input or output, and not on any data values.

THEOREM 2.4. *Any oblivious incremental algorithm that uses $S(n)$ space in addition to the input storage for polynomial multiplication, convolution, DFT, chirp z-transform, or two-dimensional DFT requires $\Omega(n / S(n))$ time per request.*

Proof. Consider a sequence of $2n$ requests in which we set all n input values and then request all n output values. By Tompa's results referred to above [22], this requires $\Omega(n^2 / S(n))$ total time; therefore, at least one of the $2n$ requests must take time $\Omega(n / S(n))$.

3. CUT VALUE INCREMENTAL ALGORITHMS

In this section, we present a method for constructing incremental algorithms for some algebraic problems—the quality of this method depends on the structure of a circuit for the nonincremental version of the problem. We will call incremental algorithms like the ones of this section '' cut value'' incremental algorithms, because they evaluate all the nodes at a cut of the problem's circuit.

In particular, we show that this method can create an $O(\log n)$. query time algorithm for the prefix sum problem and an $O(\sqrt{n})$ query time algorithm for the discrete Fourier transform. In the next section we will give a different and more widely applicable technique for constructing incremental algorithms, but the algorithms produced by that method require $\Theta(\sqrt{n})$. time per request for the prefix sum problem, and $\Theta(\sqrt{n \log n})$ time per request for the discrete Fourier transform. Thus the techniques of this section can give substantially better results, but for a smaller set of problems than the technique of Section 4. Note that it was known how to do an incremental prefix sum in $O(\log n)$ time per request by using the "prefix tree'' data structure w10x. We do not improve on these previous bounds for doing incremental prefix sum (although we do match the best previously known bound), and our presentation of this result is simply as an example of our algorithm construction methods and to show the general utility of our methods.

To create a cut value incremental algorithm, we use a cut in the supplied circuit. The cut will induce two subgraphs, with all of the input nodes in one subgraph (the *input side*) and all of the outputs in the other subgraph (the *output side*). The nodes that are in the cut itself are considered to be in *both* the input side and the output side. The idea is simple enough; when an input is changed, propagate that change as far as the cut. When an output is requested, take the set of all nodes in the cut that affect the output, and propagate the values through to the output. This second part can be simply performed by starting at the desired output node, and following edges to the cut in depth-first-search fashion.

For this to be an efficient procedure, we need to find a cut such that a particular input affects a limited number of nodes in the input side of the circuit. To make this more concrete, we introduce the following notation. For any input node v, let $\mathsf{InAff}(v)$ denote the set of nodes in the input side of the circuit whose values are affected by input node v, and let $N_{\text{in}}(v) = |\mathsf{InAff}(v)|$ denote the number of nodes in the input side of the circuit that input v affects. In the previously described cut value incremental algorithm, $N_{\text{in}}(v)$ is exactly the number of nodes that have to be reevaluated in order to propagate a change from input node v to the nodes in the cut. We let $\mathsf{MaxInAff}$ denote the size of the largest input effect set, or

$$\mathsf{MaxInAff} = \max_v N_{\text{in}}(v).$$

For any output node v, we define the set $\mathsf{OutAff}(v)$ to be the set of nodes in the output side of the circuit that affect the value of output v (note this is perfectly symmetric with the definition of $\mathsf{InAff}(w)$). Finally, define $N_{\text{out}}(w)$ and $\mathsf{MaxOutAff}$ in the obvious way.

Given the above discussion, the following theorem is simple to prove.

THEOREM 3.1. *Assume we are given a circuit for a function* $f(x_1, x_2, \ldots, x_n)$ *with* $\mathsf{MaxInAff}$ *and* $\mathsf{MaxOutAff}$ *as defined above. Then we can construct a* ''*cut value*'' *incremental algorithm that services all requests in* $O(\max(\mathsf{MaxInAff}, \mathsf{MaxOutAff}))$ *time.*

We now apply this theorem to produce incremental algorithms for the prefix sum problem and the discrete Fourier transform. The presentation of prefix sum is as a simple example of the above theorem—we note that it was already known how to obtain $O(\log n)$ bounds [10].

COROLLARY 3. 1. *There is a ‘‘cut value’’ incremental algorithm for prefix sum over a semigroup S that services requests in O(A* log *n) time, where A is the time required to apply one semigroup operation to a pair of elements in S.*

Proof. The data flow of the most common prefix sum algorithm [151 can be viewed as a tree in which we make two passes: First, we sweep from the leaves to the root, computing for each node the sum of its children (thus each node has computed the sum of all children in the subtree rooted at that node). Second, we sweep from the root to the leaves, computing the sum of all nodes ‘‘to the left’’ of the current node at that level. When we reach the leaves, we have the prefix sum.

This intuitive data flow idea can be converted into a circuit by placing two trees back to back and connecting each node with its ‘‘mirror image’’ on the other side (see Fig. 1). Each node of this tree actually does an operation over semigroup *S*, so takes time *A.* Call the tree that is closest to the inputs the ‘‘input tree,’’ and the tree closest to the outputs the ‘‘output tree.’’ The cut we use is the entire input tree. This cut consists of 2*n* — 1 nodes, but each input only affects *O*(log *n*) of these nodes (the ones on the path from the input to the root of the input tree). In other words, MaxInAff = *O*(log *n*).

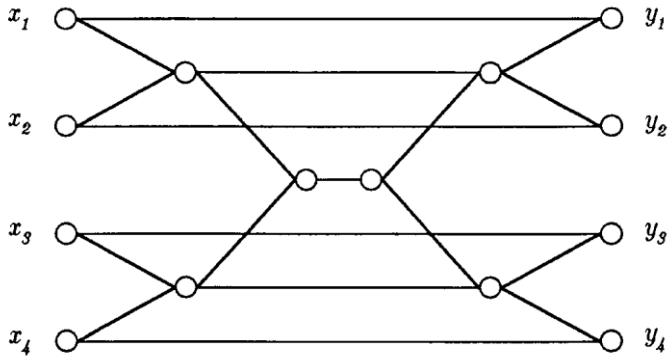

FIG. 1. Data-flow for the prefix sum operation.

The only nodes in the output tree that have any effect on a particular output are the nodes in the path from that output node to the root of the output tree. Each one of these nodes in the output tree is also affected by its “mirror image’’ node in the input tree, but this only doubles the number of nodes that affect the output. Thus MaxOutAff = *O*(log *n*).

Finally, by Theorem 3.1 we can construct an incremental algorithm for prefix sum that has worst case request service time of *O*(log *n*) node evaluations. Since each node evaluation takes time *A*, the total time of the algorithm is *O*(*A* log *n*).

Note. It should be noted that since the cut we used was actually the entire input tree, what we have derived is an incremental data structure that has a tree structure. It turns out that in this case this was a known data structure: the prefix tree [10]. In the usual cases such as computation over the integers, *A* = 1, so the total update time is *O*(log *n*). An example over a more complex semigroup is shown below.

COROLLARY 3.2. *Consider an order w linear recurrence defined by*

$$x_i = a_{i,0}x_{i-1} + a_{i,1}x_{i-2} + \cdots + a_{i,w-1}x_{i-w} + b_i$$

with a vector of initial values $(x_{w-1}, \ldots, x_1, x_0)$*, where we are interested in the values* $x_0, x_1, \ldots, x_{n-1}$*. There is an incremental algorithm for computing this function with worst-case request response time of O*(*M*(*w*)log *n*)*, where M*(*w*) *is the time required to multiply two w* × *w matrices. Equivalently, there is an incremental algorithm for solving a linear system Ax = b, where A is an n* × *n triangular banded matrix with bandwidth w* + 1 *that has worst-case request response time O*(*M*(*w*)log *n*).

Proof. The equivalence between solving a triangular banded linear system and solving a linear recurrence is well known [14], so we concentrate on solving the linear recurrence. Define *w* × *w* matrices A_i for $i = w, w + 1, \ldots, n - 1$ by

$$A_i = \begin{pmatrix} a_{i,0} & a_{i,1} & a_{i,2} & \cdots & a_{i,w-2} & a_{i,w-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{pmatrix},$$

and vectors $B_i = (b_i, 0, 0, \ldots, 0)^T$. We use X_k to denote the vector $X_k = (x_k, x_{k-1}, \ldots, x_{k-w+1})^T$, then we can rewrite the recurrence as $X_k = A_k X_{k-1} + B_k$.

We define the elements of a semigroup S to be all pairs (A, B), where A is a $w \times w$ matrix and B is a $w \times 1$ vector. The semigroup operation $\circ$ can then be defined by

$$(A, B) \circ (C, D) = (AC, AD + B).$$

If should now be obvious that if

$$(M, V) = (A_k, B_k) \circ (A_{k-1}, B_{k-1}) \circ \cdots \circ (A_w, B_w)$$

then $X_k = MX_{w-1} + V$, where X_{w-1} is just the vector of initial conditions. Computing any such (M, V) pair can be viewed as a prefix computation over semigroup S, and each semigroup operation can be done in time $M(w)$. Thus by Corollary 3.1 we can construct an incremental algorithm for this problem with worst-case request response time $O(M(w)\log n)$.

COROLLARY 3.3. *There is a* "*cut value*'' *incremental algorithm for the discrete Fourier transform that services requests in* $O(\sqrt{n})$ *time.*

Proof. The FFT graph (showing the data flow of the Fast Fourier Transform algorithm) is a layered graph with $\log n + 1$ levels of gates (see [7] or [16, pp. 713ff]). The cut we use consists of all gates at level $(\log n)/2$ (here we are assuming that $\log n$ is even—if this is not the case, then simply use the gates at level $(\log n + 1)/2$, and the rest of this proof is very similar).

From the structure of the FFT graph, we know that each input affects 2^{k-1} nodes on level k (where the inputs are labeled as level 1). So input i affects a total of

$$\sum_{k=1}^{(\log n)/2} 2^{k-1} = 2^{(\log n)/2} - 1 = \sqrt{n} - 1$$

nodes in the input side of the circuit. Since this is independent of the particular input, MaxInAff = $\sqrt{n}$ - 1.

An entirely symmetric argument shows that MaxOutAff = $\sqrt{n}$ - 1, so by Theorem 3.1, we can construct a "cut value'' incremental algorithm that has request time $O(\sqrt{n})$.

One of the important properties of the FFT graph, which makes finding the optimal cut easy, is that the FFT graph is very regular: the circuit is a layered graph in which the fanout and fanin of every node is 2. Because of this, the optimal cut is always the middle layer of the graph. We point out here that a large number of other circuits exhibit similar regularity properties. One large class of algorithms that has this property is the class of "normal hypercube algorithms'' [16] (originally called ASCEND/DESCEND algorithms [18]. which includes efficient algorithms for matrix multiplication and several popular parallel sorting algorithms such as odd—even merge sort, bitonic sort, and flash sort (see [16] or [23] for further details.. Another large class of algorithms that has the desired regularity property is the class of algorithms constructed using tensor product factorizations [13], which includes many of the transform computations used in signal processing. For these classes of problems, and others with regular circuits, finding good (or even optimal) cuts is easy, and hence constructing incremental algorithms as described in this section is easy.

4. GENERAL INCREMENTAL ALGORITHM DESIGN FOR SEQUENTIAL MACHINES

In this section we present another general purpose technique for design-ing incremental algorithms. For some problems, the resulting algorithms may not be as good as those designed using techniques from the previous section, but the techniques of this section can be applied to a fairly wide class of problems. In designing an incremental algorithm for a function $f(x_1, x_2, \ldots, x_n)$, we require both an algebraic circuit for the function and a procedure called an *update function*. The update function uses a set of inputs $(x_1, x_2, \ldots, x_n)$, *one* particular output value from $f(x_1, x_2, \ldots, x_n)$, and an optional set of precomputed values. From this information, the update function computes the change in the output value due to one of the input values changing. We will use the notation

$$\text{Update}(k, y_k, x_1, x_2, \ldots, x_n, i, x_i'),$$

where $y_k = f(x_1, \ldots, x_i, \ldots, x_n)$, to denote the function that computes the new value $y_k' = f(x_1, \ldots, x_i', \ldots, x_n)$. We use $U(n)$. (called the *update time*) to denote the time complexity of the update function.

EXAMPLE. For the discrete Fourier transform, a single output can be viewed as a polynomial evaluation at a power of a principal nth root of unity, say ω^k. Thus the value of the output maybe written as

$$y_k = x_0 + x_1\omega^k + x_2\omega^{2k} + \cdots + x_{n-1}\omega^{k(n-1)}.$$

If the input value x_i is changed to some new value x_i', then the new value of output y_k can be simply computed by

$$y_k' = y_k + (x_i' - x_i)\omega^{ik}.$$

By precomputing all of the powers of ω in the initialization phase of the incremental algorithm, we can compute this update function in constant time, so $U(n) = O(1)$.

Our incremental algorithms divide the sequence of requests into blocks of b requests, where b is called the *block size*. We evaluate the circuit during the b requests of a block, fixing the inputs to their values at the beginning of the block. If $\sigma(n)$ is the size of the circuit, we evaluate the next $\lceil \sigma(n)/b \rceil$ gates of the circuit after each request, and the entire circuit is evaluated by the end of the block. Thus at any point in the input sequence, we know all of the correct output values at the beginning of the *previous* block. We can calculate any current output value by calling Update at most $2b - 1$ times using the input changes that have been requested since the beginning of the previous block.

THEOREM 4.1. *For a particular algebraic function $f(x_1, x_2, \ldots, x_n)$, let $\sigma(n)$ be the size of the smallest circuit for that problem, and let $U(n)$. be the update time, as described above. Then we can construct an incremental algorithm for function f where each request is handled in time $O(\sqrt{\sigma(n)U(n)})$*

Proof. For each request, we evaluate $\lceil \sigma(n)/b \rceil$ gates of the circuit and may need to perform $2b$ Update operations (if the request is an output query). Thus the worst case complexity is

$$\left\lceil \frac{\sigma(n)}{b} \right\rceil + 2bU(n).$$

The b that minimizes this complexity is $b = \lceil \sqrt{\sigma(n)2U(n)} \rceil$, giving a complexity of $O(\sqrt{\sigma(n)U(n)})$.

An important aspect of this theorem is that the update time is at most $\sigma(n)$, meaning that this theorem never creates an incremental algorithm with time worse than $O(\sigma(n))$. In most cases, however, $U(n) = o(\sigma(n))$, so each request is handled in $o(\sigma(n))$ time, beating the naive approach to this problem. In fact, in many common cases $U(n) = O(1)$, so the incremental algorithm time is $O(\sqrt{\sigma(n)})$ We now look at some immediate consequences of Theorem 4.1.

The following corollary deals with algebraic functions in which each output is a linear function in each input. For example, output y_k can be expressed as a function of input x_i by $y_k = d_{i,k} + c_{i,k}x_i$. We call $c_{i,k}$ the *constant of linearity*. Note that there are common functions in which each output is linearly dependent on each input, but it

is hard to compute the constant of linearity; in other words, simply having a linear update func-tions does not guarantee fast incremental algorithms. For example, the permanent of an $n \times n$ matrix is linearly dependent on each input, but the constant of linearity cannot be computed in polynomial time unless $P = P^{\#P}$ [24], which is very unlikely.

COROLLARY4.1. *Let $f(x_1, x_2, \ldots, x_n)$ be any algebraic function such that each output value is linearly dependent on each input value, and each constant of linearity can be computed from the function inputs in constant time. Then if we have access to an algebraic circuit for f that has size $\sigma(n)$, we can construct an incremental algorithm for f that has worst-case query time of $O(\sqrt{\sigma(n)})$.*

Proof. Due to Theorem 4.1, we need only show that for such an f, $U(n) = O(1)$. If input i changes from value x_i to value x_i' then output y_k changes in value to y_k' , which can be written as

$$y_k' = y_k + c_{i,k}(x_i' - x_i),$$

for a constant of linearity $c_{i,k}$ that can be computed in constant time. The new output value can clearly be computed in constant time. The DFT example above is a concrete example of this corollary.

COROLLARY 4.2. *There exist sequential incremental algorithms for each problem in the following table, with worst case request service times as shown. In Table* 2, *the incremental version of multipoint polynomial evaluation allows the polynomial coefficients to change* (*not the points of evaluation*), *the matrices are assumed to be $n \times n$, and $M(n)$ refers to the complexity of matrix multiplication* (*currently known to be $O(n^{2.376})$* [8]). "*Restricted change linear system solve*'' *means that in a system $Ax = b$, only the elements of b are allowed to change*.

TABLE 2

Problem	Incremental time
Discrete Fourier transform	$O(\sqrt{n \log n})$
Chirp z-transform	$O(\sqrt{n \log n})$
Two-dimensional DFT	$O(\sqrt{n \log n})$
Polynomial multiplication/convolution	$O(\sqrt{n \log n})$
Multipoint polynomial evaluation	$O(\sqrt{n} \log n)$
Matrix–vector product	$O(n)$
Matrix–matrix product	$O(\sqrt{M(n)})$
Restricted change linear system solve	$O(n)$
Restricted change Toeplitz system solve	$O(\sqrt{n \log n})$

Proof. It is fairly simple to show that each of these problems satisfies the conditions of Corollary 4.1, and the worst-case time follows from Corollary 4.1 by using the size of the best known algebraic circuit for that problem (many of these circuits are easily derived from well-known sequential algorithms that can be found in standard references [2, 4, 5]). For the "restricted change linear system solve'' we note that in a system $Ax = b$, if we precompute A^{-1}, then the dynamic problem is simply an instance of matrix-vector product.

Note. The restriction on the multipoint polynomial evaluation problem that only the coefficients change is vital. Recall that in Section 2 we proved an $\Omega(n)$ lower bound for the problem when the evaluation points are allowed to change.

For a more complex example, we examine the $\Delta(r)$ *dynamic polynomial reciprocal problem*. By $\Delta(r)$, we mean that dynamic changes are allowed only in the coefficients of the r lowest order terms. For example, if the input polynomial is $p(x) = \Sigma_{i=0}^{n-1} p_i x^i$, then we only allow changes on $p_0, p_1, \ldots, p_{r-1}$.

THEOREM4.2. *There is an incremental algorithm for the $\Delta(n/2)$ dynamic polynomial reciprocal problem that has worst-case request service time of $O(\sqrt{n \log n})$.*

Proof. We denote the input polynomial by $p(x) = p_1(x)x^{n/2} + p_2(x)$, where the degree of $p_2(x)$ is at most $(n/2) - 1$. Using this notation, in the $\Delta(n/2)$ dynamic polynomial reciprocal problem, we allow changes only to the coefficients of $p_2(x)$.

If $q_1(x)$ is the polynomial reciprocal of $p_1(x)$, then the reciprocal of $p(x)$ is exactly

$$\left\lfloor \frac{2q_1(x)x^{(3/2)k-2} - (q_1(x))^2 p(x)}{x^{k-2}} \right\rfloor, \qquad (2)$$

where this formula is obtained by using the Newton iteration formula (see [2, Section 8.3]).

From this formula it is easy to see that each coefficient of the output polynomial is linearly dependent on each input coefficient, with the constant of linearity depending on the coefficients of $q_1(x)$. Since $p_1(x)$ is not allowed to change, $q_1(x)$ in the above formula is constant, and the constants of linearity may be computed in the preprocessing phase. After the preprocessing, each output update can be done in $O(1)$ time. Combining this update function with a size $O(n \log n)$ circuit for evaluating (2) gives the claimed time bound.

Note. The dramatic improvement possible by restricting the changeable inputs is fascinating here. Recall that in Section 2 we showed that if all coefficients are allowed to change, any incremental algorithm for polynomial reciprocal requires $\Theta(n)$ time; however, by restricting changes to half of the input coefficients, we manage to produce an algorithm with $O(\sqrt{n \log n})$ time per request!

5. GENERAL INCREMENTAL ALGORITHM DESIGN FOR PARALLEL MACHINES

In this section we investigate the following question: what if requests can be answered using a parallel machine, such as a PRAM? Our incremental algorithms for parallel machines use the same basic concepts as the sequential algorithm, with the added benefit of being able to do the circuit recalculation and the update function in parallel. We use $T_U(b, n)$ to represent the parallel time required to update a single output with b input changes using $P_U(b, n)$ processors. We let b be a parameter to this function, since in parallel we can often perform all b updates in $o(b)$ time.

The overall parallel algorithm is similar to the sequential algorithm of the previous section, except that we evaluate the circuit by *levels*. In other words, a time $T(n)$ circuit for a particular problem has $T(n)$ levels—if we divide the circuit re-evaluation into B phases, then we evaluate $T(n)/B$ levels per phase (note that $T(n)/B$ may be less than 1), and may have to handle as many as $2B$ — 1 individual update operations. We will sometimes refer to B as the *block size*.

THEOREM 5. 1. *For a particular algebraic function $f(x_1, x_2, \ldots, x_n)$, assume we have an algebraic circuit with width $P(n)$ and depth $T(n)$, and we have a parallel algorithm for the update function that uses $P_U(b, n)$ processors and $T_U(b, n)$ time. Then the parallel incremental algorithm with block size B can service each request with $P_{inc}(n)$ processors in time*

$$T_{inc}(n) = \frac{P(n)T(n)}{B \min(P_{inc}(n), P(n))} + \frac{P_U(2B, n)T_U(2B, n)}{\min(P_{inc}(n), P_U(2B, n))}. \quad (3)$$

We can immediately apply this theorem to give fast parallel incremental algorithms for multipoint polynomial evaluation and the discrete Fourier transform.

COROLLARY 5.1. *For the problems of discrete Fourier transform, two-dimensional discrete Fourier transform, and chirp z-transform with constant z, there are parallel incremental algorithms with request service time*

$$T_{\text{inc}}(n) = \begin{cases} \dfrac{\sqrt{n \log n}}{P_{\text{inc}}(n)} & \text{if } 1 \le P_{\text{inc}}(n) \le \sqrt{\dfrac{n}{\log n}} \\ \log \dfrac{n \log n}{P_{\text{inc}}(n)} & \text{if } \sqrt{\dfrac{n}{\log n}} < P_{\text{inc}}(n) \le n. \end{cases}$$

In addition, there are parallel incremental algorithms for multipoint polynomial evaluation with changing coefficients that have request service time

$$T_{\text{inc}}(n) = \begin{cases} \dfrac{\sqrt{n} \log n}{P_{\text{inc}}(n)} & \text{if } 1 \le P_{\text{inc}}(n) \le \sqrt{n}, \\ \log \dfrac{n \log^2 n}{P_{\text{inc}}(n)} & \text{if } \sqrt{n} < P_{\text{inc}}(n) \le n. \end{cases}$$

Proof. This proof is given for multipoint evaluation, which is almost identical to the proof of the DFT (in fact, the DFT can be viewed as multipoint evaluation at the n powers of a principal nth root of unity). Two-dimensional DFT is also similar, but with slightly different weights on the coefficients (when evaluating output $y_{i,j}$ the weight on input value $x_{k,m}$ is ω^{ik+jm}). Let $z_0, z_1, \ldots, z_{n-1}$ be the fixed points of evaluation for the input polynomial $p(x)$ with coefficients $p_0, p_1, \ldots, p_{n-1}$. Consider the $k < 2B$ input coefficient changes since the beginning of the previous block. Let s_i be the position of the coefficient for the ith change (i.e., the ith change is on coefficient p_{s_i}), and let $c_i = p'_{s_i} - p_{s_i}$ when coefficient p_{s_i} is changed to p'_{s_i}. Using this notation, we can write the change in the jth output since the beginning of the previous block as

$$y'_j = y_j + c_1 z_j^{s_1} + c_2 z_j^{s_2} + \cdots + c_k z_j^{s_k}.$$

Clearly, this is a simple weighted sum of $k + 1$ terms using the precomputed powers of an evaluation point as weights, so updates can be computed with $T_{\text{U}}(k, n) = O(\log k)$ time using $P_{\text{U}}(k, n) = O(k/\log k)$ processors.

Our incremental algorithms use this update algorithm along with a circuit for multipoint polynomial evaluation with $P(n) = O(n)$ width and $T(n) = O(\log^2 n)$ depth. By selecting a block size of $B = \sqrt{n} \log n$ when $1 \le P_{\text{inc}}(n) \le \sqrt{n} \log n$, and a block size of $B = n \log^2 n / P_{\text{inc}}(n)$ when $\sqrt{n} \log n < P_{\text{inc}}(n) \le n$, simply plugging into (3) gives the time bounds stated in the corollary.

For the discrete Fourier transform and chirp z-transform, we use circuits with $P(n) = O(n)$ width and $T(n) = O(\log n)$ depth [2, 7]. For block size selection, we use

$$B = \begin{cases} \sqrt{n \log n} & \text{if } 1 \le P_{\text{inc}}(n) \le \sqrt{n \log n}, \\ \dfrac{n \log n}{P_{\text{inc}}(n)} & \text{if } \sqrt{n \log n} < P_{\text{inc}}(n) \le n. \end{cases}$$

Using these values in (3) gives the request service times claimed in the theorem.

What we notice from this corollary is that for the discrete Fourier transform, when $P_{\text{inc}}(n) \le \sqrt{n}$, the processor/time trade-off is optimal with respect to the work required for the sequential algorithm of the previous section. For larger numbers of processors, there is still an improvement in request service time, but the trade-off is no longer linear. For the fastest algorithm, notice that with $O(n/\log^c n)$ processors for any constant c, the request service time is $O(\log \log n)$. The same bound is possible for multipoint evaluation.

CONCLUSIONS

We have examined lower bounds and algorithms for incremental versions of algebraic problems. We proved $\Omega(n)$ lower bounds for many simply stated algebraic problems such as multipoint polynomial evaluation, polynomial reciprocal, and extended polynomial GCD, meaning that sub-linear time incremental algorithms for these problems are impossible. The simple interpretation of this fact is that for many algebraic problems, no algorithm can substantially beat the trivial "reevaluate every time'' solution.

On the other hand, we have shown that by examining the algebraic circuits for certain other problems, fast incremental algorithms can be obtained. We have given incremental algorithms for a wide class of problems that have $O(\sqrt{n}\log^{O(1)} n)$ worst-case request service time (see Table 1 for a complete list of results). In addition, we have shown that for several problems in this class, superfast parallel algorithms can be derived that handle requests in $O(\log\log n)$ time using $O(n/\log^c n)$ processors for any constant $c \geq 0$.

The incremental algorithm design techniques presented are general purpose, and can be easily applied to other algebraic problems. However, due to our lower bound results, it is clear that many algebraic problems will not have fast incremental algorithms.

Notes:

1 There is clearly an equivalence between circuits with nodes presented in this order and straight-line programs. Our results could have been presented in terms of straight-line programs, but the circuit model will be more useful in Section 3.

REFERENCES

1. P. K. Agarwal, D. Eppstein, and J. Matoušek, Dynamic half-space reporting, geometric optimization, and minimum spanning trees, *in* "Proceedings, 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, 1992,'' pp. 80]89.
2. A. Aho, J. Hopcroft, and J. Ullman, "The Design and Analysis of Computer Algorithms,'' Addison-Wesley, Reading, MA, 1974.
3. E. G. Belaga, Some problems in the computation of polynomials, *Dokl. Akad. Nauk. SSSR* 123 (1958), 775]777.
4. D. Bini and V. Pan, "Polynomial and Matrix Computations.'' Vol. 1, "Fundamental Algorithms,'' ¨Birkauser, Boston, 1994.
5. A. Borodin and I. Munro, "The Computational Complexity of Algebraic and Numeric Problems,'' American Elsevier, New York, 1975.
6. Y. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry,'' Technical Report CS-91-24, Brown University Department of Computer Science, 1991.
7. J. M. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex fourier series, *Math. Comput*. 19 (1965), 297]301.
8. D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic expression, *J. Symbolic Comput*. 9, No. 3 (1990), 251]280.
9. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, Sparsification—A technique for speeding up dynamic graph algorithms, *in* "Proceedings, 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, 1992,'' pp. 60]69.
10. M. L. Fredman, "The complexity of maintaining an array and computing its partial sums,'' *J. Assoc. Comput. Mech*. 29, No. 1 (1982), 250]260.
11. G. N. Frederickson, Ambivalent data structures for dynamic 2-edge-connectivity and *k* smallest spanning trees, *in* "Proceedings, 32nd Annual Symposium on Foundations of Computer Science, San Juan, 1991,'' pp. 632]641.
12. G. N. Frederickson, A data structure for dynamically maintaining rooted trees, *in* "Proceedings of the Fourth Annual ACM–SIAM Symposium on Discrete Algorithms, Austin, TX, 1993,'' pp. 175]184.
13. J. Granata, M. Conner, and R. Tolimieri, Recursive fast algorithms and the role of the tensor product, *IEEE Trans. Signol Process*. 40, No. 12 (1992), 2921]2930.
14. D. Heller, A survey of parallel algorithms in numerical linear algebra, *SIAM Rev*. 20, No. 4 (1978), 740]776.
15. R. E. Ladner and M. J. Fischer, Parallel prefix computation, *J. Assoc. Comput. Mech*. 27, No. 4 (1980), 831]838.
16. F. T. Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,'' Morgan Kaufmann, San Mateo, CA, 1992.
17. T. S. Motzkin, Evaluation of polynomials and evaluation of rational functions, *Bull. Amer. Math. Soc*. 61 (1955),163.

18. F. P. Preparata and J. Vuillemin, The cube-connected cycles: A versatile network for parallel computation, *Comm. ACM* 24, No. 5 (1981), 300]309.
19. M. O. Rabin and S. Winograd, "Fast Evaluation of Polynomials by Rational Preparation,'' IBM Technical Report RC 3645, Yorktown Heights, NY, 1971.
20. D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comput. System Sci.* 26 (1983), 362]391.
21. K. J. Supowit, New techniques for some dynamic closest-point and farthest-point problems, *in* "Proceedings of the First Annual ACM—SIAM Symposium on Discrete Algorithms, San Francisco, 1990,'' pp. 84]90.
22. M. Tompa, Time—space tradeoffs for computing functions, using connectivity properties of their circuits, *J. Comput. System Sci.* 20 (1980), 118]132.
23. J. Ullman, "Computational Aspects of VLSI,'' Comput. Sci. Press, Rockville, MD, 1984.
24. L. Valiant, The complexity of computing the permanent, *Theoret. Comput. Sci.* 8 (1979), 189]201.