A Complexity Theoretic Approach to Incremental Computation

S. Sairam Jeffrey S. Vitter Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910

Technical Report No. CS-91-66
November 1991

A Complexity Theoretic Approach to Incremental Computation*

S. Sairam, Jeffrey Scott Vitter, and *Roberto Tamassia*
(ss@cs.brown.edu, jsv@cs.brown.edu, rt@cs.brown.edu)

Department of Computer Science
Brown University
Providence, R. I. 02912–1910

Abstract

We present a new complexity theoretic approach to incremental computation. We define complexity classes that capture the intuitive notion of incremental efficiency and study their relation to existing complexity classes. We show that all common LOGSPACE-complete problems for P are *incr*-POLYLOGTIME-complete for P. This suggests that non-redundant problems that seem inherently unparallelizable also seem hard to dynamize. We show that a form of transitive closure is complete under incremental reduction for NLOGSPACE and give similar problems which are incrementally complete for the classes LOGSPACE and non-uniform NC^1. We show that under certain restrictions problems which have efficient dynamic solutions also have efficient parallel solutions.

We also look at the time complexity of circuit value and network stability problems restricted to comparator gates. We show that the dynamic version of the comparator-circuit value problem and "Lex-First Maximal Matching" problem is in LOGSPACE and that of the comparator-network stability problem and "Man-Optimal Stable Marriage Problem" is in LOGSPACE given an oracle which operates in NLOGSPACE. This shows that the dynamic versions of these problems are solvable quickly in parallel even though there are no known NC algorithms to solve them from scratch.

---

*Support for the first two authors was provided in part by an National Science Foundation Presidential Young Investigator Award CCR–9047466 with matching funds from IBM, by NSF research grant CCR–9007851, by Army Research Office grant DAAL03–91–G–0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225. Support for the third author was provided in part by by NSF research grant CCR–9007851, by Army Research Office grant DAAL03–91–G–0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225.

1 Introduction

What is the "best" algorithm for a given problem? There is obviously more than one answer, depending on what we mean by "best." If we consider worst-case time behavior, traditionally an algorithm is considered the best possible if it meets the information theoretic lower bound. There is another way of looking at the situation, wherein we are not worried about the time taken to evaluate every instance from scratch. We ask that once the algorithm has preprocessed an instance of the problem, it should handle any changes to the instance (up to a certain limit) very fast.

In the next section we formalize these notions in terms of the classes *incr*-POLYLOGTIME, the class of problems whose dynamic versions are solvable in poly-logarithmic time, and *incr*-POLYLOGSPACE, the class whose dynamic versions can be solved with poly-logarithmic work space. We also give a restricted notion of nondeterministic incremental computation. We then introduce the concept of an *incremental reduction* between problems, and we present problems that are incrementally complete for other existing complexity classes under such a reduction.

In Section 3 we show that commonly known P-complete problems (under LOGSPACE reductions) are *incr*-POLYLOGTIME-complete for the class P. We further prove that all *non-redundant* P-complete problems are *incr*-POLYLOGTIME-complete for P. This suggests that problems that are hard to parallelize are hard to dynamize. We also show that a variant of transitive closure is *incr*-CONSTANTTIME-complete for NLOGSPACE and give similar problems which are *incr*-CONSTANTTIME-complete for LOGSPACE and NC^1. In Section 4 we demonstrate that under certain restrictions problems which have efficient dynamic solutions also have efficient parallel solutions.

In Section 5 we look at *incr*-LOGSPACE; the class of problems that can be "updated" using only logarithmic work space. This is an interesting class because problems in this class are amenable to fast parallel updates (since LOGSPACE $\subseteq$ NC^2). We show that the circuit value problem for comparator gates, discussed in Mayr and Subramanian [9] is in *incr*-LOGSPACE while the network stability problem is in relativized *incr*-LOGSPACE with respect to an NLOGSPACE oracle. Since these problems have no known efficient parallel solutions, we get a new class of problems which seem to be somewhat more sequential in nature than the problems in NC but nonetheless exhibit a degree of parallelism absent in P-complete problems. In Section 6 we present the conclusions and some open problems.

2 Preliminaries

Informally, we want a decision problem π to belong to the class *incr*-TIME$[f(n)]$ if we can "react" to the changes in a given instance of the problem in time proportional to $f(n)$. First we take some time to build a data structure for an initial instance of the problem. After that point we are required to react to any one-bit change in the current instance in time $O(f(n))$; that is, we should calculate the solution for the new instance and update the current data structure all in time $O(f(n))$.

Definition 1 A decision problem π belongs to the class *incr*-TIME$[f(n)]$ if there are RAM programs P_1 and P_2 such that for all $n \in \mathbf{N}$ we have

1. Given any initial instance I_0 of π ($|I_0| \leq n$), P_1 efficiently precomputes some auxiliary data structure D_{I_0} associated with instance I_0.

2. Given the current input instance I, the incremental change $\triangle$ to I (where $\triangle$ changes I to another instance I' of π, for which $|I'| \leq n$) and the current data structure D_I in the random access memory, P_2 determines $\pi(I')$, and modifies D_I into the new data structure $D_{I'}$ in $O(|\triangle| f(n))$ time.

Our notion of "efficient" in condition 1 is somewhat flexible; we could require that the precomputation be doable in polynomial time or logarithmic space, for example. The RAM programs are allowed to operate on $O(\log n)$ length words in constant time, where n is the size of the problem instance. They are therefore allowed to read, write, add and subtract $O(\log n)$ length words. The program counter is also allowed to transfer control to an arbitrary address (e.g., to execute a jump).

We denote by *incr*-POLYLOGTIME the class

$$\bigcup_{k \geq 0} incr\text{-TIME}[\log^k n].$$

It is easy to see that such a definition captures the intuitive notion of incremental computation; for example the problem of maintaining an ordered set through a set of updates and queries is in *incr*-TIME[$\log n$].

We can define the class *incr*-SPACE[$f(n)$] as above, except that P_2 is required instead to use $O(|\triangle| f(n))$ space. We denote by *incr*-LOGSPACE the class *incr*-SPACE[$\log n$], and we define *incr*-POLYLOGSPACE to be the class

$$\bigcup_{k \geq 0} incr\text{-SPACE}[\log^k n].$$

To extend these notions to nondeterminism seems to be hard since we can then have multiple copies of the data structure. We side step that issue by allowing nondeterminism only in the form of decision oracles and give the following restrictive definition to *rincr*-LOGSPACE(NLOGSPACE)

Definition 2 A decision problem π belongs to the class *rincr*-LOGSPACE(NLOGSPACE) (read as π is in *incr*-LOGSPACE relative to the class NLOGSPACE) if there are RAM programs P_1 and P_2 and a nondeterministic decision procedure S such that for all $n \in \mathbf{N}$

1. Given any initial instance I_0 of π ($|I_0| \leq n$), P_1 efficiently precomputes some auxiliary data structure D_{I_0} associated with instance I_0.

2. Given the current input instance I, the incremental change $\triangle$ to I (where $\triangle$ changes I to another instance I' of π, for which $|I'| \leq n$) and the current data structure D_I in the random access memory, P_2 determines $\pi(I')$, and modifies D_I into the new data structure $D_{I'}$ in $O(|\triangle| \log(n))$ space. P_2 is allowed to make a polynomial number of calls to the procedure S which works in space $O(|\triangle| \log(n))$.

To compare the "hardness" of solving two problems in this incremental sense, we need the notion of incremental reduction. It is clear that the program doing the reduction from problem π_1 to problem π_2 has to be incremental in the same sense as the RAM program P_2 in the previous definition. In the same way as before we allow some computation to construct a data structure to an initial instance of π_1 and to find the corresponding instance of π_2. We then require that any change to the current instance of π_1 be reflected as a change to the corresponding instance of π_2, within a stated time limit. There's however a small difference, because now we are dealing with functions (in the previous case the solution was always a "yes" or a "no") which map instances of π_1 to instances of π_2. We take care of this by quantifying the change to the solution given a change to the input.

Definition 3 A decision problem π_1 is *incrementally reducible* to another decision problem π_2 in time and size bounds $[f(n), g(n)]$, denoted $\pi_1 \leq_{incr[f(n),g(n)]} \pi_2$, if

1. There is a transformation $T : \pi_1 \rightarrow \pi_2$ that maps instances of π_1 to instances of π_2 such that I is a positive instance of π_1 if and only if $T(I)$ is a positive instance of π_2.

2. There are RAM programs P and Q such that for all $n \geq 0$ we have

   (a) Given any initial instance I^0 of π_1 ($|I^0| \leq n$), P efficiently computes $T(I^0)$ and some auxiliary data structure S_{I^0} associated with I^0.

   (b) Given the current instance I, the incremental change $\triangle_1$ to I (where $\triangle_1$ changes I to another instance I' of π_1, and $|I'| \leq n$) and the current data-structure S_I in the random access memory, Q constructs the incremental change $\triangle_2$ to $T(I)$ (where $\triangle_2$ changes $T(I)$ to $T(I')$), such that $|\triangle_2| \leq g(n)|\triangle_1|$, and modifies S_I into the new data-structure $S_{I'}$ in $O(|\triangle_1| f(n))$ time.

Note that $g(n)$ is always $O(f(n))$. We say that $\pi_1 \leq_{incr[f(n)]} \pi_2$ if $\pi_1 \leq_{incr[f(n),f(n)]} \pi_2$.

Theorem 1 *If π_1 is incrementally reducible to π_2 in time and size bounds $[f(n), g(n)]$, and if π_2 is in incr-*TIME$[h(n)]$*, then π_1 is in incr-*TIME$[f(n) + h(n)g(n)]$. □

3 Incrementally-Complete Problems

We now turn to the notion of incremental completeness. The idea is to find a problem which is as hard to solve incrementally as any other problem in a given class. In this section we use incremental reductions to get natural problems incrementally complete for P, NLOGSPACE LOGSPACE, and NC^1.

Definition 4 A problem π is said to be *$incr[f(n), g(n)]$-complete* for a class C if

1. π is in the class C.

2. For all π_1 in C, π_1 is incrementally reducible to π in time and size bounds $[f(n), g(n)]$.

We say that π is *$incr[f(n)]$-complete* for a class C if it is *$incr[f(n), f(n)]$-complete* for a class C. We call a problem π *incr*-POLYLOGTIME-complete for a class C if π is *$incr[f(n)]$-complete* for C, where $f(n)$ is $O(\log^k n)$ for some k. We define *incr*-CONSTANTTIME-completeness in an analogous fashion.

The obvious question to explore is how the complexity classes *incr*-POLYLOGTIME, *incr*-POLYLOGSPACE, and *incr*-LOGSPACE are related to the well known sequential complexity classes? Our first intriguing result is that the commonly known P-complete problems listed in [11] and [5] are *incr*-POLYLOGTIME-complete for the class P. These problems include the Circuit-Value problem (CV), the Solvable Path System problem (SPS), Propositional Horn Satisfiability, and a host of other well known P-complete problems. These problems are therefore as hard to solve incrementally as any other problem in P.

Theorem 2 *All* P*-complete problems in [11] and [5] are* incr-POLYLOGTIME*-complete for the class* P*.*

Proof Sketch: We present the proof for the case of circuit value problem. Given any problem π in P and an initial instance I^0 with $|I^0| \leq n$, we use the standard LOGSPACE reduction to create a circuit of size $t(n)$ by $t(n)$ (where $t(n)$ is a time bound for some polynomial time turing machine M to solve instances of π of size at most n) that simulates the turing machine M used to solve the problem π. The inputs to this circuit are the bits of the initial instance I^0. This gives us the initial transformation of I^0. A one bit change to I^0 results in a one bit change to the corresponding instance of the CV (the input variable in the circuit corresponding to the changed bit is changed to reflect the new input). All this can be done in constant time, and thus the Circuit-Value problem is *incr*-POLYLOGTIME-complete for P.

We can provide similar proofs for the other problems listed in [11] and [5]. The reductions needed to show this are sometimes different from those given in the literature. We require for example that when a general problem π in P is changed by one bit, input to the incrementally complete problem under consideration changes only by a polylog number of bits. The use of preprocessing plays a significant role, since the size of the instance of the incrementally complete problem may be a polynomial factor larger than the problem π in P. □

Corollary 1 *If the* P*-complete problems in [11] and [5], like circuit value problem, are in* incr-POLYLOGTIME *then all of* P *is in* incr-POLYLOGTIME*.*

Corollary 2 *If there are* NC *algorithms to dynamically update the* P*-complete problems in [11] and [5], then we automatically get* NC *algorithms to dynamically update all of* P*.*

These corollaries suggest that it is highly unlikely that problems like the circuit value problem can be dynamized.

Kasif and Delcher [3] propose other notions of completeness. They define the incremental version of a function f as follows: Let f be a function that maps input X of length $|X|$ to output $f(X)$. Then Inc-f is the function that given inputs of form $(X, f(X), X')$, computes $f(X')$, where X and X' differ in at most $\log |X|$ bits. A function f is Inc-P-complete iff for every function g in P, Inc-g is LOGSPACE-reducible to Inc-f. With this definition they argue that the "All-Outputs Monotone Circuit Value Problem" is Inc-P-complete. The drawback in this approach is that it disallows the use of dynamic data structures. In other words their incremental reductions are not strong enough to conclude that it is unlikely that one will get good algorithms (meaning NC algorithms) for the incremental versions of Inc-P-complete problems irrespective of the size of the auxiliary storage. They try to overcome this drawback by proving that the incremental versions of many P-complete problems are also P-complete. Here they completely ignore the issue of preprocessing. Though their results are interesting, they are somewhat weak in that the issues of data structures and preprocessing are overlooked. They conjecture that the incremental versions of all P-complete problems are P-complete.

Interesting notions of completeness are sketched by Reif [14]. He shows that some problems are unlikely to have efficient incremental solutions, but he does not develop a comprehensive theory or consider the necessary details of preprocessing. Some interesting techniques are explored in [2], [4] and [13] to derive lower bounds for incremental algorithms. Their main idea is to construct an algorithm for the batch version of the problem with input I by computing a dynamic data structure for some easily-computed instance I', whose Hamming distance from I is small, and then applying the incremental algorithm repeatedly while I' is successively modified to I. This gives lower bounds for the incremental problem in terms of that for the batch problem. The drawbacks of this approach are that it limits the amount of preprocessing available and is problem specific.

Theorem 2 would seem to suggest that all P-complete problems are *incr*-POLYLOGTIME-complete for P. Unfortunately that is not the case, since one can take any *incr*-POLYLOGTIME-complete problem π and create another problem π' by duplicating the input, so that π' is no longer *incr*-POLYLOGTIME-complete even though it is still P-complete. In fact we prove that some P-complete problems can be solved effectively incrementally.

Theorem 3 *There are* P*-complete problems that are in incr*-POLYLOGTIME*.*

Proof Sketch: Let L be some P-complete language over the alphabet $\sum = \{0, 1\}$ such that it takes linear sequential time to determine whether a given word w is in L or not. Let us consider the language $L' = \{w^{|w|} | w \in L\}$. Obviously L' is also P-complete. We claim that membership queries for L' can be performed dynamically. We will abuse notation slightly by using the symbol L' both for the language and the associated membership problem. Consider an initial n^n-bit instance I^0 of the problem L' (instances which are not of the form n^n for some $n \in \mathbf{N}$ can be handled without too much trouble). We spend polynomial time to see if it is of the form $w^{|w|}$ for some $w \in L$. This, we can do by running a subroutine S_L to check membership in L and some associated linear time work to see if I^0 is of the form $w^{|w|}$. At any point in time let the current instance I be a concatenation of n bit strings $a_1, a_2, \ldots, a_n$. The dynamic algorithm works as follows:

1. As long as less than $n/2$ of the a_i's are equal (that is, less than $n/2$ of the strings are the copy of the same string w) answer "no" for every instance.

2. If at some point more than $n/2$ of the a_i's are all equal to the same w, start a background process which executes the subroutine S_L on the string w; and for every update from then on, spend constant amount of time (the constant depends on the sequential time complexity of L) on the background computation.

3. We now claim that whenever the current instance I of L' is $= w^{|w|}$ for some $w \in \sum^*$ the background process already has the answer for whether w is in L or not! To prove that let us consider the sequence of instances starting at I^0, going through $I^1, I^2, \ldots$, all the way up to the current instance $I = I^k$ for some k, where every instance I^l is derived from the previous instance I^{l-1} by updating the bit that was changed at

time $l-1$. Let I^j be the instance closest to I such that it is a concatenation of the strings $b_1, b_2, \ldots b_n$ and $n/2$ of the b_i's are equal. It is easy to see that $k - j \geq n/2$, therefore starting at time j the background computation S_l ran for at least $O(n)$ steps before the time k. This implies that by the time we reached instance $I = w^{|w|}$ the background computation S_L had already executed the number of steps necessary to determine whether w is an element of L or not. The membership query for I is the same as the one for w. If w is in L we answer "yes" for the instance I otherwise we answer "no."

A more careful look shows us that we had the time to execute the procedure S_L in the background due to the sparseness of L' which was a result of concatenating strings of L to themselves "many" times. □

The above construction works by taking an *incr*-POLYLOGTIME-complete problem and introducing redundancy by copying, and in fact all the ways to generate incrementally tractable problems from P-complete problems seem to involve redundancy in some form or the other. We restrict ourselves therefore to P-complete problems that are in some sense non-redundant. To do that we look at a much stricter definition of P-completeness; one in terms of projections, introduced by Skyum and Valiant [15].

Definition 5 A decision problem π_1 is *projection reducible* to another decision problem π_2 ($\pi_1 \leq_{proj} \pi_2$) if there is a function $p(n)$ bounded above by a polynomial in n, and a family of polynomial time computable mappings $\sigma = \{\sigma_n\}_{n \geq 1}$ where

$$\sigma_n : \{y_1, \ldots, y_{p(n)}\} \rightarrow \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}, 0, 1\},$$

and for any n-bit instance I of π_1 we can derive a $p(n)$-bit instance I' of π_2 using the mapping σ_n such that I is a positive instance of π_1 if and only if I' is a positive instance of π_2. To derive I' we give x_i the same value as the ith bit of I and use $\sigma_n(y_i)$ as the ith bit of I'. We say that σ is bounded above by p and π_1 is *σ-reducible* to π_2.

Definition 6 A problem π is said to be $<_{\text{proj}}$-*complete* for a class C if π is in C and there is a function $p(n)$ bounded above by a polynomial in n, such that every problem $\pi_1 \in C$ is σ-reducible to π by a projection $\sigma = \{\sigma_n\}_{n \geq 1}$ bounded above by p.

Problems like the circuit value problem are $\leq_{\text{proj}}$-complete for P. Even under this restricted setting it is possible to create problems which are $\leq_{\text{proj}}$-complete for P but are in *incr*-POLYLOGTIME. To remedy that we define the following notion of *non-redundancy:*

Definition 7 Let π_1 be a decision problem and π be another problem such that $\pi_1 \leq_{proj} \pi$. We say that π is *non-redundant with respect to* π_1 if there exists a polynomial time computable family $\sigma = \{\sigma_n\}_{n \geq 1}$ of mappings and a number $k \in \mathbf{N}$ such that π_1 is σ-reducible to π (where σ_n is a mapping from the set $\{y_1, \ldots, y_{p(n)}\}$ to $\{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}, 0, 1\}$), and for all symbols x_i, $|\sigma_n^{-1}\{x_i, \overline{x_i}\}| = O(\log^k n)$.

We now define the notion of non-redundancy with respect to a class C.

Definition 8 Let π be a problem which is $\leq_{\text{proj}}$-complete for C. We say that π *is non-redundant with respect to* C if it is non-redundant with respect to every problem in C. We then call π a *non-redundant projection complete problem* (or a NRP-complete problem) for the class C.

Lemma 1 *Let* C *be a class of decision problems and let* π_1 *be a* NRP*-complete problem for* C*. If* π *is another decision problem in* C *such that* $\pi_1 \leq_{proj} \pi$ *and* π *is non-redundant with respect to* π_1 *then* π *is* NRP*-complete for* C

Proof: The proof follows from the definitions in a straightforward manner. □

The P-complete problems listed in [11] and [5] are all NRP-complete for P. The following theorem shows that this large class of P-complete problems are difficult to make efficient incrementally.

Theorem 4 *Let* C *be a class of decision problems and* π *be a* NRP*-complete problem for* C*, then* π *is incr-*POLYLOGTIME*-complete for* C.

Proof Sketch: The proof follows from noting the fact that given any problem $\pi_1 \in$ C, there is projection mapping from π to π_1 such that a one bit change in an input instance of π_1 causes at most polylog number of bits to change in the corresponding instance of π. □

We now look at the classes NLOGSPACE, LOGSPACE and NC^1.

Theorem 5 *The following variant of transitive closure is incr-*CONSTANTTIME*-complete for* NLOGSPACE*:*

Input: A directed graph $G = (V, E)$ *such that each* $e \in E$ *is colored with* 0 *or* 1*; a pair of vertices* v_1 *and* v_2*; a partitioning of* V *into sets* $V_1, V_2, \ldots, V_k$*; and a color vector* C *of length* k*.*

Output: Is there is a path p *from* v_1 *to* v_2 *such that all edges in* p *emanating from vertices in* V_i *have color* $C[i]$*?*

Proof: The reduction from any problem in NLOGSPACE to this variant is straightforward. The trick that makes the reduction go in constant time and constant storage is the use of the sets $V_1, V_2, \ldots, V_k$ and the color vector. We group all intermediate states that read the input bit i into the class V_i; $C[i]$ is assigned the same value as input bit i. Now a change in some input bit i changes $C[i]$. □

A restriction of the above problem is *incr*-CONSTANTTIME-complete for LOGSPACE while a similar variant of bounded width polynomial size branching programs [1] gives us an *incr*-CONSTANTTIME-complete problem for non-uniform NC^1.

Surprisingly however there are problems which are NC^1-complete for LOGSPACE, but are nevertheless in *incr*-POLYLOGTIME, as demonstrated by the following theorem:

Theorem 6 *The problem of Undirected Forest accessibility* (UFA) *is in incr-*TIME$[\log n]$*, under additions, deletions, linking and cutting of trees.*

Proof Sketch: The dynamic maintenance is done by using balanced trees such as red-black trees to maintain ordered sets with insert, delete, split, and join [6].

- Preprocessing: An Euler tour of each tree in the forest is maintained in a red-black tree. Each edge of the tree occurs twice in its Euler tour.

- Query(n_1, n_2): Let e be an edge in the adjacency list of n_1 and f an edge in the list of n_2. We follow pointers in the red-black tree to determine if they both have the same root.
- Linking and cutting are implemented using the split and join operations of red-black trees, with $O(\log n)$ time per operation.

$\square$

The interesting thing to note here is that even though UFA is not known to be in NC^1 it is in *incr*-TIME$[\log n]$. At the same time, all of NC^1 is not known to be in *incr*-TIME$[\log n]$.

4 Incremental Computation in Restricted Realms

The class of *Z-stratified trees* was introduced by Overmars [12] in an effort to form a general theory of balancing in search trees. It subsumes a large portion of the search structures used to design dynamic algorithms. In this section we show that under reasonable assumptions problems which have incremental solutions based on these structures also have parallel solutions.

Theorem 7 *Let π be a problem in incr-*TIME$[\log^k n]$*, for some $k \geq 1$. Then π is in* NC^{k+2}*, if the following constraints hold:*

1. *The data structure used is an augmented Z-stratified search tree. Each internal node may contain information relevant not only to enable search but may contain the result of some function f applied to its children. The only restriction is that f be computable in time $O(\log^k n)$.*
2. *There is a special instance I_0 such that the tree corresponding to I_0 can be built in* LOGSPACE.
3. *Any ℓ-bit change to the current input I, where $1 \leq \ell \leq n$, involves modifying the tree by inserting or removing some number of elements. The elements to be inserted or deleted are functions of the current input, the current data structure, and the change, and they can be computed in* LOGSPACE.
4. *After all the insertions and deletions some sort of search is conducted in the tree in time $O(\log^k n)$ to get the answer to the updated instance.*

Proof: Given instance I we proceed as follows: First we create the tree corresponding to I_0 in LOGSPACE and calculate all the deletions and additions needed to get the tree for I, also in LOGSPACE. Now we determine the order in which they are to be present in the final tree in NC^{k+2}. This can be done because the function f which is used in determining the order between two elements is computable in $O(\log^k n)$. Therefore the set of elements in the tree for I can be sorted in EREW^{k+1}, and hence in NC^{k+2} [8]. We then construct the tree in time NC^{k+2} in a bottom-to-top sweep. This can be done because Z-Stratified trees have logarithmic depth, and the value of the function f at a node depends only on the contents of the nodes of its children. All we have to do now is to let one processor walk down the tree and get the answer to I in time $O(\log^k n)$. $\square$

This deceptively simple theorem demonstrates why most known problems with efficient dynamic solutions have optimal parallel solutions. For example we can use this theorem to parallelize dynamic algorithms using degree-balanced trees, height-balanced trees, path-balanced trees etc. We can derive similar results for other classes of trees as well.

One important technique in getting dynamic solutions for problems is the divide and conquer approach. To exploit this technique Mehlhorn and Overmars [10] consider an important class of problems called order-decomposable problems.

Definition 9 A set problem π is called $C(n)$-*order-decomposable* if and only if there exists an ordering ORD and a function $\square$ such that for each set of $n \geq 1$ points $V = \{p_1,\ p_2,\ \ldots,\ p_n\}$, ordered according to ORD, and for each $1 \leq i \leq n$, we have

$$\pi(\{p_1, p_2, \ldots, p_n\}) = \square(\pi(\{p_1, p_2, \ldots, p_i\}), \pi(\{p_{i+1}, p_{i+2}, \ldots, p_n\})),$$

where $\square$ takes at most $C(n)$ time to compute when V contains n points. In other words a problem is order decomposable if after arranging it according to a specific ordering, the problem can be split at any point to give two smaller subproblems whose solutions can be glued together "efficiently" to get the solution for the original problem.

Mehlhorn and Overmars essentially prove that if π is an $O(\log^k n)$-order-decomposable set problem and it takes less than $O(\log^k n)$ time to compare two elements to determine their ordering with respect to ORD, then $\pi \in$ *incr*-TIME$[\log^{k+1} n]$. We make the following rather simple extension.

Theorem 8 *If π is an $O(\log^k n)$-order-decomposable set problem and the ordering of two elements with respect to* ORD *takes time $O(\log^k n)$, then $\pi \in$ incr*-TIME$[\log^{k+1} n]$ *and* $\pi \in$ NC^{k+2}.

Proof: The parallel algorithm uses a divide and conquer approach to split the problem recursively into roughly equal parts. The solutions are later glued together using $\square$. We first sort the input according to ORD. This takes time $O(\log^{k+1} n)$ in a EREW PRAM and is therefore in NC^{k+2} then we solve the problem for n singleton sets. We now use a bottom-up approach to glue the solutions back together. This takes $\log n$ phases and hence can be done in $O(\log^{k+1} n)$ by an EREW PRAM, which in turn implies that it can be done in NC^{k+2}. $\square$

This theorem gives automatically generated parallel algorithms for problems such as

- Calculating the Convex hull of a set of points.
- Finding the maximal element in two dimension; and
- Calculating the Voronoi diagram of a set of points in two dimension.

5 Problems on Comparator Gates

In this section we consider the circuit value and network stability problems over comparator gates, introduced in [9]. We show that these two problems are in *incr*-LOGSPACE and in *rincr*-LOGSPACE(NLOGSPACE) respectively, and thus their dynamic versions can be solved quickly in parallel, even though the batch versions have no known NC solutions.

The definitions of circuits and gates are taken from [9] and are given here for the sake of completeness.

Definition 10 A *circuit* is a directed acyclic graph, where each node represents a gate. The incoming edges are inputs while the outgoing ones are the outputs of the gate. A *network* is a circuit with feedback, in other words the underlying graph need not be acyclic. A gate preserves adjacency (is *adjacency preserving*) if it maps adjacent input words (binary words of the same length that differ in at most one bit) into adjacent output words. A gate is *monotone* if it cannot simulate the NOT gate. A *comparator* gate, takes as input a and b and gives as output $a \wedge b$ and $a \vee b$. The circuit value problem over comparator gates (C-CV) is the circuit value problem where all the gates are comparator gates.

Theorem 9 *The circuit value problem over comparator gates* (C-CV) *is in incr*-LOGSPACE.

Proof: Comparator circuits are adjacency preserving; a one bit change at any place transmits only a one bit change to succeeding levels. The desired algorithm follows by using a variant of list ranking to do the incremental computation.

- Preprocessing: The circuit is evaluated in polynomial time for the initial instance and the values at each edge are maintained.

- Change in one input bit: This change is propagated in LOGSPACE, by walking through the circuit, since a one bit change in one level implies a one bit change in the succeeding level. The space required is obviously logarithmic.

- Introduction of a gate: Let i_1 and i_2 be inputs to gates g_1 and g_2, such that the new gate g to be introduced takes as input i_1 and i_2 and feeds its outputs o_1 and o_2 to g_1 and g_2 respectively. To effect this change, we just walk down the circuit twice, starting at gates g_1 and g_2, taking into account the new inputs to these gates caused by the introduction of the new gate.

- Deletion of a gate: The process here is similar to insertion and we again walk down the circuit twice from the point where the deletion took place, updating edge values in the process.

All these steps can be done in logspace therefore C-CV is in *incr*-LOGSPACE. □

A similar argument holds for any circuit that is adjacency preserving. This result is interesting because C-CV is not known to be in NC. Therefore C-CV has the distinction of being a problem that can be updated fast in parallel, though it is not known whether it can be solved fast from scratch in parallel. This also means that problems like the "Lex-First Maximal Matching," problem which can be parsimoniously reduced to C-CV are in *incr*-LOGSPACE.

We now turn our attention to the network stability problem on comparator gates. A network is stable for a given input assignment if we can assign values to the edges which are consistent with the gate equations and the given input assignment. Given a network N

consisting of only monotone gates and an input assignment S_{in}, it can be easily seen that there exists a stable configuration. Therefore the answer to the network stability problem on comparator gates is always a yes. The interesting question therefore is the value of an edge in particular types of stable configurations. Given a network N and an input assignment S_{in} we are interested in knowing the value of an edge in the "most-zero" stable configuration S_{min}. This configuration has the property that if an edge e is 0 in any stable configuration S of the network, it is 0 in S_{min}.

Theorem 10 *The problem of finding whether an edge e is 0 in the "most-zero" configuration of a comparator network N is in rincr-*LOGSPACE*(*NLOGSPACE*).*

We provide a description of the algorithm used in updating the network and then prove that it is correct.The approach is similar to the one taken in the case of the comparator circuit value problem, except that we now need the use of nondeterminism to recognize and traverse cycles in the network.

Proof:

- Preprocessing: The most zero configuration of the network is evaluated in polynomial time for the initial instance and the values at each edge are maintained. The initial edge values are evaluated by forcing the inputs one by one in some predetermined order until no more edges can be forced. For any input i the edges forced form a path (not necessarily simple) from i to some out put o. We will call this path the snake i and label every edge in the snake with i. Let π be the order in which the inputs are forced. We say that snake i is less than snake j, if input i was forced before input j. We call i a zero-snake (a one-snake) if the input i is zero (one). All the edges which are not forced by any inputs are given the value 0 and are given a special cycle label C. C is placed at the end of the ordering π. It can be shown that C is a collection of zero-cycles.

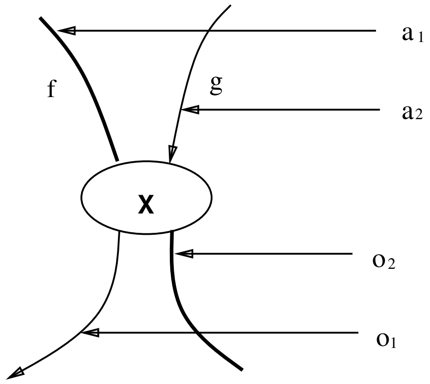

Figure 1: When two zero snakes meet

- Update: We now show how to maintain these values and the labels of snakes when an input is changed; the other operations of introducing or removing gates can be similarly handled. We consider just the change from 0 to 1; the other case is similar. At all times we maintain two values (Sn, Fl), where Sn is our current perception of the snake whose input has changed from zero to one and Fl is the snake label that

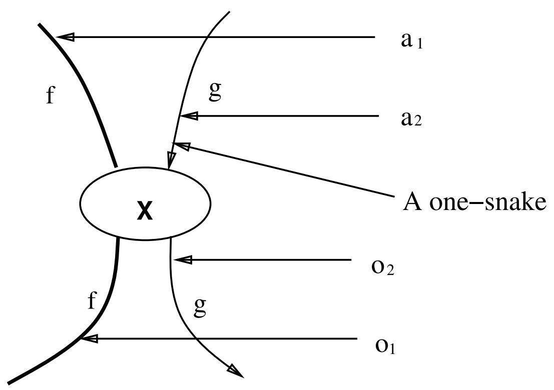

Figure 2: When a zero snake meets with a one-snake

we are following. Initially both Sn and Fl have the value i (the changed input). We walk down the circuit following label Fl changing edge values and replacing edge labels with Sn. All the changes are recorded in a write only copy of the network which in the beginning of the update corresponds to the current state of the network. Any changes made cannot be read again. This restriction is to constrain the algorithm to work in LOGSPACE. At each step we decide whether to continue along the same path, or to do certain intermediate computations. Let X be the current gate along the path, with input edges a_1 and a_2 and output edges o_1 and o_2. Let o_1 be the AND output and o_2 the OR output. Suppose that a_1 was changed from 0 to 1 in the last step then we make the decision on what to do next based on the following criteria.

1. - a_1 is on the snake f and a_2 is on snake g.
   - Both f and g are zero-snakes.
   - f is less than g in the ordering.

   We are now in a situation where o_1 was on f and o_2 was on g (see Figure 1). We therefore need to do some label changing since the OR output should be on snake f (this is so because f is less than g in the ordering and hence should force the OR output). Snake f should therefore get label g beyond X (or put another way the snakes are changing pieces). However we can't just change the label of f to g, all the zero-snakes in between f and g will be affected as well. We do this by the subroutine ZERO-LABEL(f, g) (to be described later), which relabels edges in all zero-snakes $f \leq h < g$ with labels greater than h. It is clear that these will be the only snakes that will be affected. We then assign the follow Fl to be g change o_2 to have value 1 and label Sn and continue.

2. - a_1 is on the snake f and a_2 is on snake g.
   - Both f and g are zero-snakes.
   - f is greater than g in the ordering.

   We are now in a situation where the AND output o_1 was on snake g and the OR output o_2 on snake f (see Figure 1), o_2 should now become 1, we therefore make no changes to either Sn or Fl. We change the edge o_2 to have label Sn and value 1 and continue forcing (changing labels and values as warranted) the input change in the network.

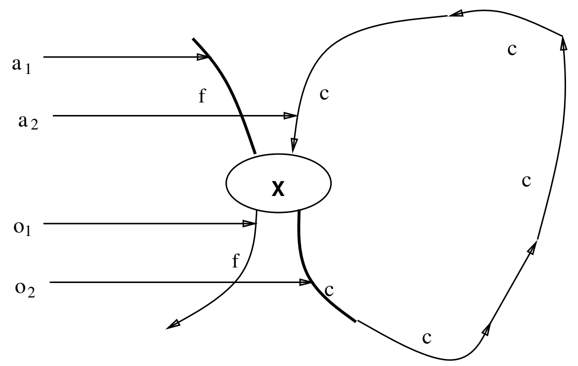

Figure 3: When the forcing path meets a zero-cycle

3. – a_1 is on the snake f (or on C) and a_2 is on snake g.
   – f is a zero-snake while g is a one-snake.
   – Sn is less than g in the ordering.

   We are in a situation similar to Case 1 (see Figure 2). The OR output is on snake g (since g is a one-snake), but now that f (C) has become part of the one-snake Sn the OR output should be on Sn . The edges on snake g therefore have to get the label Sn (and maybe others in between Sn and g). We now use a similar routine ONE-LABEL(Sn, g) to (this routine is symmetric to the ZERO-LABEL(f, g) subroutine) relabel the edges of all one-snakes $Sn < h \leq g$. Then we assign the value g to S_n; change the edge o_1 to have value 1 and label Sn and continue (note: assigning Sn the value g is equivalent to changing our perception of the input that changed to g).

4. – a_1 is on the snake f (or on C) and a_2 is on snake g.
   – f is a zero-snake while g is a one-snake.
   – Sn is greater than g in the ordering.

   We are now in a situation similar to Case 2 (see Figure 2); since Sn is greater than g the OR output o_2 should be on g. Now that a_1 changes to 1, so should o_1. Sn and Fl remain the same, and we continue forcing the input change in the network.

5. – a_1 is on the snake f and a_2 is on C.
   – f is a zero-snake.

   We are now in a situation wherein one or more zero-cycles of C will become part of the new one-snake Sn (see Figure 3). We perform the following steps

   – Find out if the edges of this cycle C_1 of C have already had their values changed. This is done by a call to an oracle FORCED(a_2, X) which looks to see whether the edge a_2 can be forced to a 1 before reaching a_1. The oracle FORCED works in NLOGSPACE and will be described later.
   – If the cycle has already been forced we don't enter it but force the AND output o_1 to 1 instead and change its label to Sn before continuing.

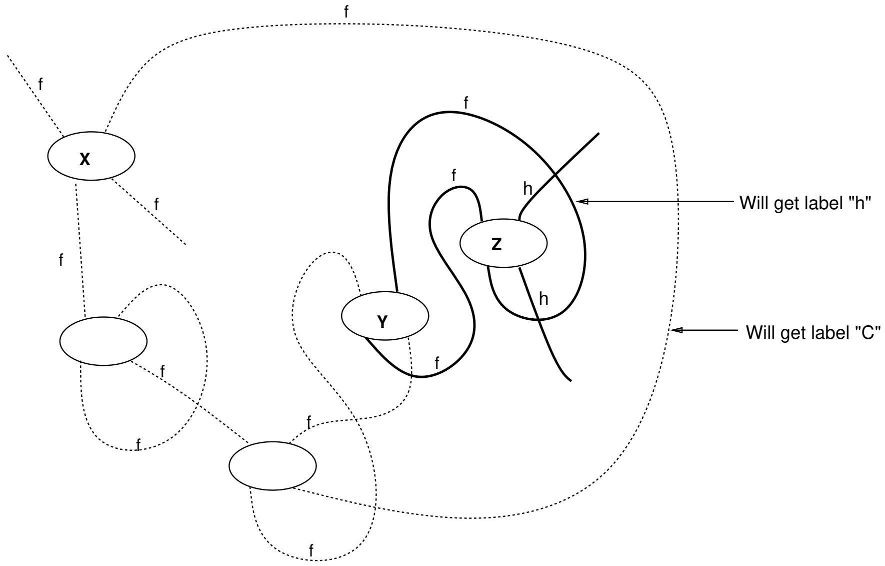

Figure 4: A self loop that breaks up

- If C_1 has not been entered before, we mark X, and enter the cycle, by making use of a temporary follow TFl which is set to C.
- We continue with the forcing using the criteria for Cases 3 and 4 until we get back to X. During this process we may enter into other zero-cycles, but we can argue inductively that that we will always come back since the follow TFl is always C (this is so because Cases 1 and 2 never apply since C is greater than all the snakes in the ordering).
- The only thing left to show is how to recognize that we have indeed come back. Suppose we are at a node Y with input nodes a_1 and a_2 and output nodes o_1 and o_2 all of which have the label C. To find out if we have come back we need to determine whether to take the AND output o_1 or to take the OR output o_2, we can determine that by calling the oracle FORCED(a_2, Y) and taking OR if it answers "no" and the AND output if it answers "yes."
- When we reach X we continue as before.

6. – a_1 a_2 o_1 and o_2 belong to the same zero-snake f.

We have now encountered a loop in the snake f (see Figure 4). Since a_1 has changed in value to 1, we should now take the OR output o_2. The edges in the loop formed with the AND output should therefore get other labels. As before we need to see if the loop has been encountered; this we do by calling FORCED(a_2, X). If the answer is "yes" we change the label and value of o_2 and continue, otherwise we perform the following steps to change the labels of the loop L consisting of all the edges starting from o_1 to a_2 on the snake f.

- Change the labels of the edges in L to C (in the output tape).
- For all zero-snakes $h \in \pi$ (in the reverse order) except f, find the earliest node Y in L such that we can reach Y with a label h. Due to the change in snake f the default labeling for the edges in the loop L is C, however if we

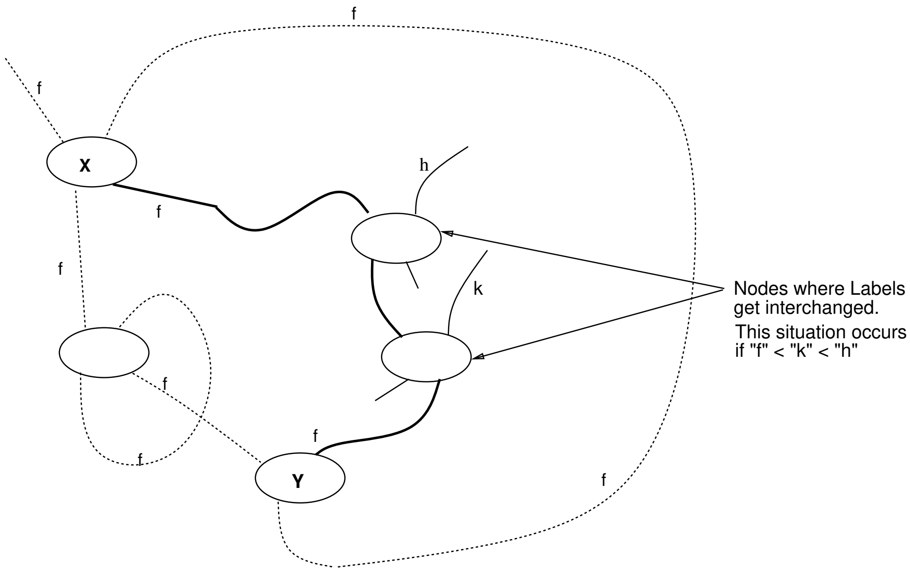

Figure 5: How label interchanges affect the new labeling

can reach some node Y in L with some label h (where h is a zero-snake), then some of the edges in L beyond Y would get the label h (since C is greater than all the snakes in the ordering π). We therefore find the earliest such Y by calling the routine REACHED(X, Y, h) (to be described later) for all the Y's one by one. Some of the ways a label h can reach the node Y are shown in Figures 4 and 5. In Figure 4 we see the case wherein a subcycle of L gets the label h, and in Figure 5 we see how to reach Y with a label h after a complicated interchange of labels. In this case all of L gets the label h.

– Label the edges in the smallest subcycle of L containing Y with label h.

The Loop L has now been divided amongst the other snakes and C, note since we did all the writes in the reverse order as in π we are guaranteed that the various orderings will be obeyed at all nodes in the loop.

We now give brief descriptions of the subroutines FORCED, REACHED, and ZERO-LABEL. The procedure ONE-LABEL is similar to the ZERO-LABEL procedure and is omitted. The procedure FORCED(a_2, Y) works as follows:

1. Start at the input which has been changed, and follow the path of forcing until a node with a zero-cycle is reached. At this point we know that if we take the OR output we will come back; therefore we have a non deterministic choice to make as to whether we should take the OR or the AND output to reach the edge a_2 of gate Y.

2. If we reach a_2 within n steps without reaching a_1 we answer "yes" otherwise "no".

The procedure REACHED(X, Y, h) works as follows:

1. We can reach Y with h if h is a zero snake which meets the loop at Y and one of the following holds

- The label on the edges of h is not changed by the change in the value of a_1 (see Figure 4) of X.
- We can reach Y with the label h on edge a_1 (for instance) not in the loop L, such the label interchanges happen either when a forcing path meets a zero-snake k such that $k < Fl$, or if a zero-snake x whose label has been changed to z meets another zero-snake y, such that $x < y < z$. See Figure 5 in which $f < k < h$. Therefore node Y is reachable with label h.
- If either of the cases outlined above, hold we answer "yes" otherwise we answer "no".

2. Fortunately we can do all this in NLOGSPACE with calls to FORCED.

The procedure ZERO-LABEL(f, g) works as follows:

1. For all zero-snakes h, such that $f \leq h < g$ consider all the zero-snakes $k \neq h$ between h and g in the reverse order, and do the following:

2. Find the earliest node Y on h with inputs a_1, a_2 and outputs o_1, o_2 such that, the AND output o_1 is on snake h, and Y can be reached with label k. This is done using the subroutine REACHED.

3. If such a Y exists relabel all the edges in h beyond Y with the label k.

An inductive proof can be used to show that ZERO-LABEL(f, g) labels the snakes $f \leq h < g$ correctly. Note all the changes are made in the output tape.

The proof of correctness depends upon one crucial fact. When we reach a node X such that a_1 is on the snake f and a_2 is on snake g, both f and g are zero-snakes, and f is greater than g in the ordering, then g has not been changed in any previous labeling. The proof follows from the fact that after every labeling the value of Fl becomes greater than all the snakes whose edges have changed. This implies that even though we are allowed to look only at the input tape, we have sufficient information to make correct decisions. A similar statement can be made about the situation when we meet a one-snake. In all our calls to NLOGSPACE oracles we have taken the liberty of assuming that the oracle gives both "yes" and "no" answers correctly, which is not entirely correct. Every call to any of the subroutines should be interpreted as a dual call to the routine and its complement (executed in an interleaved fashion). When one of them answers we stop the other routine, the whole computation is still in NLOGSPACE because NLOGSPACE and CO-NLOGSPACE are the same [7]. Since the algorithm above works in LOGSPACE and uses NLOGSPACE oracles, the network stability problem in in *rincr*-LOGSPACE(NLOGSPACE). □

Subramanian [16] has shown that a number of problems concerning stable marriage are equivalent to the problem of network stability in comparator gates. Decision problems based on the "man-optimal stable marriage problem " and other problems parsimoniously reducible to the comparator network stability problem are therefore in *rincr*-LOGSPACE(NLOGSPACE). Like the Comparator Circuit Value problem, these too have no known NC algorithms to solve them from scratch.

6 Conclusions and Open Problems

We have provided a firm theoretical base to conduct the study of incremental computation. We have demonstrated the existence of problems that are incrementally complete for various natural complexity classes and shown some important special cases wherein dynamic solutions imply parallel ones. We have also shown that the comparator circuit-value problem is in *incr*-LOGSPACE and the comparator network stability problem is in *rincr*-LOGSPACE(NLOGSPACE). We would like canonical problems for the classes we have defined, and would like to answer some of the following questions.

1. How is *incr*-POLYLOGTIME related to the class LOGSPACE?

   - Is there some restriction to LOGSPACE that will automatically give us incremental algorithms for languages in this restricted class? The problem here seems to be in defining a meaningful restriction of LOGSPACE for which Directed forest accessibility is *incr*-POLYLOGTIME-complete.
   - Is there some restriction of *incr*-POLYLOGTIME which is a subset of LOGSPACE?

2. What is the relation between *incr*-POLYLOGTIME and NC, the class of problems solvable by bounded fan-in circuits of polynomial size and polylog depth?

   - Are restrictions of either class comparable to the other?

3. What is the relation between *incr*-POLYLOGTIME and problems which have optimal parallel algorithms?

   - Are there general reductions from one to the other? Do problems like *st*-numbering that have optimal parallel algorithms also have incremental algorithms.
   - It seems hard to convert any parallel algorithm that uses reachability in some form into a corresponding dynamic algorithm for the same problem. We feel that restricting ourselves to parallel algorithms which avoid using transitive closure as a subroutine may help us in getting dynamic algorithms. This is the case with a number of recent parallel algorithms for planar graphs. It would be interesting to see if any generic reductions are possible in this restricted realm.

4. A particularly interesting class of problems in computational geometry deals with the manipulation of generalized trees. In this restricted class there seems to be an even stronger relation between *incr*-POLYLOGTIME and NC, as is demonstrated by Theorem 7 and Theorem 8. We would like to explore this relation further.

5. How are *incr*-POLYLOGTIME *incr*-LOGSPACE and *rincr*-LOGSPACE(NLOGSPACE) related to the class CC of problems reducible to C-CV? Does the fact that comparator gates preserve adjacency make these problems incrementally tractable? We have partially solved this problem by showing that the circuit value problem on comparator gates is in *incr*-LOGSPACE and the network stability problem is in *rincr*-LOGSPACE(NLOGSPACE). However we are unable to show that CC is in *rincr*-LOGSPACE(NLOGSPACE); this is because the many-one reduction used by Subramanian [16] may not be parsimonious. In particular we don't

know whether the network stability problem on non-bipartitionable X-gates is in *rincr*-LOGSPACE(NLOGSPACE).

We believe that the classes *incr*-TIME and *incr*-SPACE are an important means for getting a better understanding of the relationship between incremental and parallel computation.

References

[1] D. Barrington, "Bounded Width Polynomial Size Programs Recognize Exactly Those Languages in NC^1," *Proceedings of STOC'86* (1986), 1–5.

[2] M. Berman, M. C. Paul, and B. G. Ryder, "Proving Relative Lower Bounds for Incremental Algorithms," *Acta Informatica* (1989).

[3] A. Delcher and S. Kasif, "Complexity Issues in Parallel Logic Programming," Johns Hopkins University, Ph.D Thesis, 1989.

[4] S. Even and H. Gazit, "Updating Distances in Dynamic Graphs," *Methods of Operations Research* 49 (1985), 371–387.

[5] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, "A Compendium of Problems Complete for P," Department of Computer Science and Engineering, University of Washington, Technical Report TR-91-05-01, 1991.

[6] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978), 8–21.

[7] N. Immerman, "Nondeterministic Logspace is Closed Under Complement," Yale University, TR No YALEEU/DCS/TR 552, 1987.

[8] R.M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared Memory Machines," in *Handbook of Theoretical Computer Science*, North Holland, 1990.

[9] E. Mayr and A. Subramanian, "The Complexity of Circuit Value and Network Stability," *Proceedings of Structures '89* (1989).

[10] K. Mehlhorn and M. Overmars, "Optimal Dynamization of Decomposable Searching Problems," *Information Processing Letters* 12 (1981), 93–98.

[11] S. Miyano, S. Shiraishi, and T. Shoudai, "A List of P-Complete Problems," Research Institute of Fundamental Information Science, Kyushu , Research Report , 1989.

[12] M. Overmars, "The Design of Dynamic Data Structures," *Lecture Notes in Computer Science* 156 (1983).

[13] M. C. Paull, Charles Ching an Chang, and A. M. Berman, "Exploring the structure of incremental algorithms," Department of Computer Science, Rutgers University, Technical Report DCS-TR-139, May, 1984.

[14] J.H. Reif, "A Topological Approach to Dynamic Graph Connectivity," *Information Processing Letters* 25 (1987), 65–70.

[15] S. Skyum and L. G. Valiant, "A Complexity theory based on Boolean Algebra," *Proc. 22nd IEEE Symposium on Foundations of Computer Science* (1981), 244–253.

[16] A. Subramanian, "A New Approach to Stable Matching Problems," Department of Computer Science, Stanford University, Research Report, 1989.