PARALLELISM AND COST IN PARALLEL TREE OPERATIONS

BY

BRYAN BARNEY REAGAN
B.S., University of Illinois at Chicago, May 1991

THESIS

Submitted as partial fulfillment of the requirements for the
degree of Master of Science in Electrical Engineering and Computer Science
in the Graduate College of the
University of Illinois at Chicago, 1995

Chicago, Illinois

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AVL tree	A Balanced Binary Tree
GI	Graph Interference
LSF	Least Square Fit
NP	Non-deterministic Polynomial Time
TI	Tree Interference
WSPTI	Weighted Sum of Previous Two Iterations

SUMMARY

This thesis examines the issues related to parallelism and costs in parallel execution of operations on tree based data structures in a parallel processing environment with shared memory. Two algorithms to schedule operations so to avoid conflicts, named graph interference and tree interference, are given, and their performance characteristics are measured and compared. In addition, a technique called dynamic window sizing is developed which limits overhead by adjusting how much parallelism is attempted on each iteration of a scheduling algorithm. A simple, but effective dynamic window sizing technique called the weighted sum of previous two iterations (WSPTI) is defined and tested, and found to limit the cost per operation to a constant value.

1. Introduction

As parallel processing continues to evolve, it is desirable to find efficient parallel analogues of uni-processor data structures and algorithms. This serves two purposes. First, these analogues facilitate re-implementation of existing uni-processor software on the newer machines. Second, and more significantly, these new implementations extend the set of parallel processing techniques to a wider range of problems which achieve speedup.

This thesis is concerned with implementation techniques for tree based algorithms on parallel processors with shared memory. The methods developed should apply to most types of trees, including binary search trees, red-black trees and B-trees, and AVL trees. In particular, tree operations which consists of a phase of reading down the tree, followed by a phase of updating the tree, can be parallelized with a high degree of efficiency using these techniques. AVL tree insertions have been used in example, since they have complications due to re-balancing with pointer aliasing, and non-uniform write accessing.

An algorithm is parallelized by decomposing the work performed into independent computations, which may be executed simultaneously, with the possible addition of work to coordinate these segments. These computations are then executed simultaneously on separate processors. Tree-based algorithms are well suited for parallelization, since the majority of the write accesses occur near the leaves of the tree, the parallelism would seem to increase linearly as the size of the tree increases (Solworth and Reagan, 1994). However, the nature of re-balancing requires that some operations must write access more nodes than others, often with several operations conflicting over a common ancestor node. Avoiding write conflicts requires careful testing, exclusion management, and operation scheduling requirements. Due to these complications, very little work has been done on parallel tree based structures, with (Tow, 1991), (Carlilse et al., 1994), (Solworth and Reagan, 1994), and (Solworth and Reagan, 1995) being exceptions.

1.1 Problem Description

The goal of this thesis is to determine the amount of achievable parallelism in tree operations in a shared memory environment, to quantify the relationship between testing and scheduling costs and achieved parallelism, and to develop a viable set of testing and scheduling algorithms that will yield substantial parallelism with a high degree of efficiency.

1.2 Background

Before any formal theory regarding parallel tree algorithms can be discussed, certain fundamental concepts must be enumerated. These include: measurements of parallel algorithms, the relationship between memory accesses and the relative ordering of operations, and trapdoors.

1.2.1 Measurements for the Evaluation of Parallel Algorithms

The primary motivation for parallel computation is to reduce execution time. Hence, for a given problem, the fundamental performance measurement of a parallelized algorithm is the relationship between the parallel and sequential execution times. Informally, ***speedup*** is the ratio of the execution time of the original, sequential algorithm to that of a new, parallel algorithm. More formally, consider an instance of a problem, I_n with input length n, which is solved by an optimal sequential algorithm on a uni-processor machine in time $T_1[I_n]$. Let $T_p[I_n]$ (for $p > 1$) be the execution time required for some parallel version of this algorithm, on the same input, running on p processors. Speedup, denoted $\sigma(I_n, p)$ is the ratio of $T_1[I_n]$ to $T_p[I_n]$ which measures how much faster the parallel version runs relative to the uni-processor version (Quinn and Deo, 1984). Notice that since the amount of required work is not decreased by conversion to a parallel algorithm, $\sigma(I_n, p)$ cannot exceed p (Baase, 1984). Similarly, when a sequential algorithm is converted to a parallel algorithm for $p > 1$ processors, the work performed must be partitioned into sets of operations, some of which may be executed in parallel, but some of which may not. Many algorithms contain some operations which are inherently sequential, and hence cannot be parallelized (Amdahl, 1967). The presence of such sequential components sets a limit on the ability to reduce the

execution time by simply adding more processors, and this limitation has come to be known of as ***Amdahl's Law***. These properties of speedup are summarized by formula (1) below.

$$1 \leq \sigma\ (I_n , p) = \frac{T_1[I_n]}{T_p[I_n]} \leq\ p \qquad (1)$$

Example: Consider an instance of a problem, of length n, for which a uni-processor algorithm finds a solution in 30 minutes. A parallel algorithm, which runs on 4 processors, finds the same solution for the same instance, in 10 minutes. The speedup is computed as follows.

$$\sigma\ (I_n , 4) = \frac{T_1[I_n]}{T_p[I_n]} = \frac{30}{10} = 3.0$$

Clearly the degree to which parallelism may be achieved dictates speedup. From Amdahl's law, a relationship can be derived, based on the amount of achievable parallelism, which will estimate the limits of speedup and parallelism (Amdahl, 1967), (Christianson, 1991). Let α denote the fraction of the operations in a given program that can be executed in parallel. (Criteria for parallel execution of operations will be covered below.) If we are given an instance of a problem I_n , its sequential execution time, $T_1[I_n]$, and α, its execution time on $T_p[I_n]$ can be estimated as a function of p, as per formula (2) below. By taking the limit as p diverges to infinity, the limit on execution time is computed, as given in (3) below. This limit corresponds to the time to execute the sequential portion of the program alone.

$$T_p[I_n] = (1-\alpha)\ T_1[I_n] + (\alpha\ T_1[I_n]) / p \qquad (2)$$

$$\lim_{p \to \infty}\ T_p[I_n] = (1-\alpha)\ T_1[I_n] \qquad (3)$$

Similarly, by substitution of (2) into (1), speedup can be estimated as a function of the number of processors, and the limit of achievable speedup predicted. These are given in equations (4) and (5) below respectively.

$$\sigma(I_n, p) = \frac{T_1[I_n]}{T_p[I_n]} = \frac{1}{(\alpha / p) + (1 - \alpha)} = \frac{p}{\alpha + (1 - \alpha)p} \quad (4)$$

$$\lim_{p \to \infty} \sigma(I_n, p) = \frac{1}{(1 - \alpha)} \quad (5)$$

Hence it is clear that α is critical to the effectiveness of parallel algorithm, insofar as it limits how much speedup is possible. It is important to note that speedup is simply a measure of the relationship between execution times, and does not reflect how well resources are used. ***Efficiency*** of a parallel algorithm is a measure of how it uses time on all available processors. Formally, efficiency for a given instance I_n on p processors is denoted $\varepsilon(I_n, p)$, and is defined as the speedup divided by the number of processors (Quinn and Deo, 1984). Since speedup is never greater that the number of processors, efficiency can never be greater than 1 (and only equal to one when α is also equal to 1). These concepts are summarized in formula (6) below.

$$\varepsilon(I_n, p) = \frac{\sigma(I_n, p)}{p} \leq 1 \quad (6)$$

Example: Consider the case of speedup above, where $\sigma(I_n, 4) = 3.0$. The efficiency for this instance is computed as follows.

$$\varepsilon(I_n, 4) = \frac{\sigma(I_n, p)}{p} = \frac{3}{4} = 0.75$$

Overhead is the collective name for the fraction of execution time on all processors that is either spent waiting, or that is used performing functions that are only required in the parallel execution (Leighton, 1984). Some sources of overhead are machine dependent, such as the communication latency and context switch time. Some portions of a sequential algorithm may be inherently sequential, and not be able to be executed in parallel, forcing some (or most) of the processors to sit idle (Amdahl, 1967). In addition, some overhead is incurred due to parallel execution, such as coordinating the computing of the sum of the subtotals computed by separate processors. The fraction of execution time on all processors that

is overhead is denoted by the symbol $\phi(I_n, p)$ and is the difference between one and the efficiency. This is summarized by formula (7) below.

$$\phi(I_n, p) = 1 - \varepsilon(I_n, p) \quad (7)$$

Example: Consider the case of efficiency above , where $\varepsilon(I_n, 4) = 0.75$. The overhead for this instance is computed as follows.

$$\phi(I_n, 4) = 1 - \varepsilon(I_n, 4) = 1 - (0.75) = 0.25$$

Unfortunately, speedup, efficiency, and overhead can only be measured for specific implementations executing on specific machines. Although they remain the key concepts, they will not be used in the experiments for this thesis, which will assume no limits on parallelism due to finite numbers of processors. Parallelism will be measured by the mean number of operations completed in one phase of parallel execution, which provides an machine independent upper bound for speedup. The cost per operation will be measured in the mean number of required read phases, which will be explained in chapter 2.

1.2.2 Memory Accesses and Relative Ordering of Operations

In order for a parallelized algorithm to be consistent with the sequential version, operations which are performed in parallel must produce resulting states of the data structures that are identical to some sequential execution of these operations. If this is not the case, the parallel version may violate some constraints on the states of the data structures, hence producing unexpected results. Not all sets of operations produce the same results as a uni-processor execution when executed in parallel, and hence may require execution in a fixed order relative to each other. Fortunately, it can be determined which operations may be performed in parallel.

Two or more operations ***conflict*** if a location is modified (written) by one operation and is accessed (read or written) by another (Larus and Hilfinger, 1988). Clearly the final state of the data

structures being accessed depends upon the relative order of execution of conflicting operations, and if these operations are performed in parallel, then sequential inconsistency can arise (Larus and Hilfinger, 1988). In contrast, if a set of operations does not conflict, then these operations may be executed in any relative order with no effect on the final state of the data structures being accessed, and with no danger of sequential consistency if performed in parallel (Solworth and Reagan, 1994).

Hence to find a set of operations which may be executed in parallel, a simple set of non-conflicting operations can be used (Tow, 1991), (Solworth and Reagan, 1994). Since no operation modifies a location used by another, there is no danger of inconsistency from out of date information, or from one operation overwriting the write accesses of another. In such sets of operations in which the order of the operations is irrelevant to the final state of the data structures, the operations are said to be ***arbitrary order operations*** (Solworth and Reagan, 1994). In contrast, operations which must be executed in a specific order are said to be ***fixed order operations*** (Solworth and Reagan, 1994).

The use of arbitrary order operations is the basis of the Graph Interference method, which shall be described below (Solworth and Reagan, 1994). However, due to some specific properties of tree structures, there exists circumstances where sets of operations with conflicts with exactly one write accesse to a shared location may be allowed, and this forms the basis of a new technique called Tree Interference, which will be introduced below (Solworth and Reagan, 1994).

1.2.3 Trees and Trapdoors

Before describing the parallelization of tree algorithms, the concept of a ***trapdoor*** must be introduced (Tow, 1991). In a data structure which will allow simultaneous accesses, certain portions of the data structure must be shared. Similarly, it is possible for some portions of the structure to require exclusive access. A trapdoor is a location, or a pointer to a location within a structure, at which shared portions of the data structure are linked to non-shared portions, and beyond which accesses must be performed sequentially on the non-shared portions (Tow, 1991). In the case of tree structures, each free (null) pointer represents a trapdoor (Tow, 1991). Associated with each trapdoor is a range of key values, which may be

validly inserted at that pointer (Tow, 1991). For an arbitrary binary search tree, where mutual exclusion must only be maintained only at the specific pointer to be modified , parallel insertions may be scheduled by selecting sets of values that are in the ranges of separate trapdoors (Tow, 1991). In more complex tree structures, such as AVL or B trees, an operation may require write accesses to portions of the tree other than just a single free pointer. An update of this type, such as AVL tree insertion, is said to write ***outside the trapdoor*** (Tow, 1991). In the case of writing outside the trapdoor, more sophisticated exclusion techniques are required to prevent read-write or write-write conflicts, which can result in sequential inconsistency (Tow, 1991). Figure 1 below shows a tree with three nodes and its four trapdoors, with integer key values in the range K = [K-,K+].

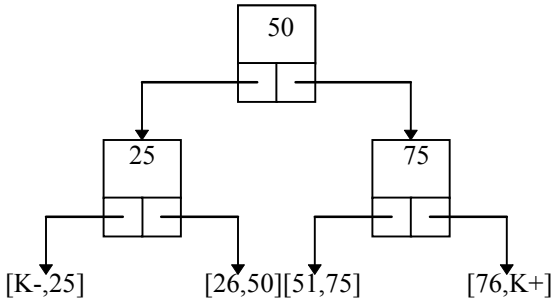

Figure 1. A Three Node Tree and its Trapdoors

2. Tree Algorithms

2.1 Access Paths, Read Sets and Write Sets

In order to qualify the limits of parallelism in tree algorithms, the nature of tree operations must be understood. For an operation on an arbitrary data structure, an ***access path*** is the in order sequence of elements or fields of elements accessed, which can be analyzed to prevent conflicts (Larus and Hilfinger, 1988). In tree operations, the access path is operated on in two phases: a ***read phase*** of read accesses scanning from the root down to some level, followed by a ***write phase*** of a possibly empty series of write accesses proceeding back up the tree along the read path, possibly including some nodes adjacent to the path (e.g., transformations in AVL re-balancing) (Solworth and Reagan, 1995). This pattern will be called the ***read down - write up paradigm*** , and is common to most tree structures (Solworth and Reagan, 1995).

In order to exploit locality in a tree structure, the access path is decomposed into two components. Specifically, the nodes that are write accessed, and on those that are not. Formally, given an operation q_i, let the access path be denoted $P(q_i)$, and contain elements which are nodes of the tree. Let the ***write set*** of q_i, denoted $W(q_i)$, be the subset of $P(q_i)$ containing all nodes which are write accessed, and the ***read set***, denoted $R(q_i)$ be the set of nodes in $P(q_i)$ that are not write accessed. The relationships between these sets are summarized in formula (8) below.

$$R(q_i) = P(q_i) - W(q_i) \qquad (8)$$

For tree operations which write outside of the trapdoor, the distance from the root at which the highest write access occurs directly determines the amount of parallelism. For example, an update to either the root pointer, or an operation that changes the value or identity of the root node conflicts with any other operation on the tree. In order to measure write access height, the function $Lev(q_i)$ is defined for an operation q_i as follows. If q_i writes the root pointer or its referent, $Lev(q_i) = 0$, and otherwise $Lev(q_i)$ = the minimum of the distances from the root from the nodes written (Solworth and Reagan, 1994).

2.2. Interference and Resolution

The primary source of difficulty in parallelizing of tree operations is that variable numbers of write accesses outside the trapdoor are required for each individual operation (Tow, 1991). In the case of an AVL tree insertion, which corrects the balances of all the nodes it traverses, only one pointer (i.e.. the one associated with the trapdoor) must be modified (Reingold and Hansen, 1986). Otherwise, re-balancing is required, which involves modifying three or four pointers, possibly including the root pointer (Reingold and Hansen, 1986). This has two immediate complications. Foremost, simultaneous accesses to the same pointer can result in write-write or read-write conflicts, introducing not only inconsistent structures, but also lost pointer referents. Secondly, the use of transformations in re-balancing can invalidate a recursively stored path stored in the stack (Carlisle et al., 1994). Hence, mutual exclusion is required, and may be maintained ether by explicit scheduling as in (Tow, 1991), or by requiring that memory be updated only by a local processor, having the processors themselves maintain mutual exclusions as in (Carlisle et al., 1994). Both (Carlisle et al., 1994) and (Tow, 1991) propose re-splicing the run time stack to repair erroneous stack information.

Both of these problems may be avoided, by the use of non-recursive access techniques, together with preventative scheduling, which shall be presented below. Clearly the pointers of nodes within the write set become sources of conflict or interference. Since updates to the same trapdoor have the same write set, they may be assumed to be associated with the trapdoors. Hence if a set of operations has disjoint write sets, then they can be executed in parallel. Two techniques for race free scheduling can be used: Graph Interference, and Tree Interference (Solworth and Reagan, 1994). These will be described below.

2.2.1 Graph Interference

Graph Interference is a general technique, which given a set of operations to perform, iteratively finds subsets of operations, which can be executed without conflicts (Solworth and Reagan, 1994). Graph interference (GI) is sufficient for most pointer based structures, and does not exploit any special characteristics of trees (Solworth and Reagan, 1994). This is accomplished by the following algorithm, which uses a graph based technique to find a set of operations in which no operation accesses (read or

write) a node that is updated by another operation in the set (Solworth and Reagan, 1994). This results in unrestricted concurrent reads, but maintains mutual exclusion on write accesses, hence eliminating read-write and write-write conflicts (Solworth and Reagan, 1994). Hence it can be used in an iterative process to reduce a set of operations into collections of non-conflicting operations. This is accomplished by creating a ***conflict graph***, which contains one vertex per operation, with undirected edges between pairs of vertices which correspond to conflicting operations (Solworth and Reagan, 1994). The information required to build this graph can be obtained by executing the read phases of all operations, which have no restrictions on parallel execution (Solworth and Reagan, 1994).

By selecting a set of vertices, such that no two are adjacent, the corresponding operations are mutually conflict free (Solworth and Reagan, 1994). Any independent vertex set of the conflict graph corresponds to a set of conflict-free operations. Similarly, by coloring the graph, all vertices of one color would also represent a set of conflict-free operations. Since the graph must be recomputed after each iteration, no advantage is gained by using graph coloring. In addition, maximum independent sets or optimal graph coloring could be used for small numbers of operations, but since they are ***NP***-Complete, this would be impractical to use for realistic sized sets of operations (Garey and Johnson, 1979). Hence greedy (non-optimal) versions are used in practice.

Algorithm 1. Basic Graph Interference

Given a tree and a set of operations to update it, $Q=\{q_0, q_1, q_2, ...\}$

While $(Q \neq \varnothing)$ do

1. $\forall\ q_i \in Q$, compute $R(q_i)$ and $W(q_i)$
2. Construct the conflict graph, $G=(V, E)$ for Q as follows:

$$V = Q$$

$$E = \{(q_i, q_j) \mid q_i, q_j \in Q,\ q_i \neq q_j\ \wedge\ (W(q_i) \cap P(q_j)) \cup (W(q_j) \cap P(q_i)) \neq \varnothing\}$$

3. Select $A \subseteq Q$ such that $\forall\ a_i, a_j \in A, (a_i, a_j) \notin E$
4. $Q = Q - A$
5. Perform the operations in A

EndWhile;

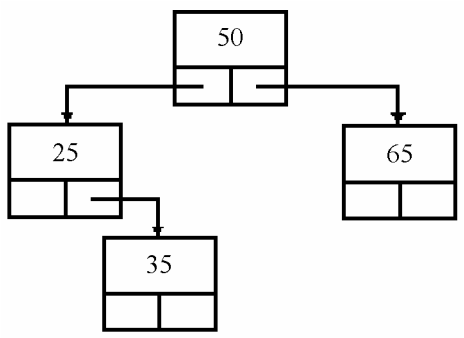

Figure 2. A Tree of Four Nodes

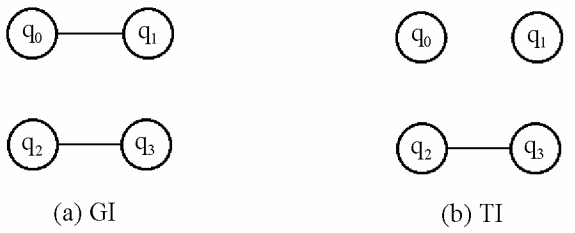

Figure 3. Conflict Graphs for Examples on Figure 2.

Example: Consider the binary tree depicted in Figure 2. Let Q contain operations to insert (assuming no rebalancing) the following values: 24, 37, 52, and 67, denoted as q_0, q_1, q_2, and q_3 respectively. For purposes of notation, let N[x] denote the node in the tree containing the key value x. For the first iteration, the read sets and write sets for each operation are computed as follows:

$R(q_0) = \{N[50], N[25]\}$ $\quad W(q_0) = \{N[25]\}$

$R(q_1) = \{N[50], N[25], N[35]\}$ $\quad W(q_1) = \{N[35]\}$

$R(q_2) = \{N[50], N[65]\}$ $\quad W(q_2) = \{N[65]\}$

$R(q_3) = \{N[50], N[65]\}$ $\quad W(q_3) = \{N[65]\}$

The conflict graph may then be constructed for graph interference as follows, which is also depicted in Figure 3 (a) above.

$$V = \{ q_0, q_1, q_2, q_3 \}$$

$$E = \{(q_0, q_1), (q_1, q_0), (q_2, q_3), (q_3, q_2)\} = \{(q_0, q_1), (q_2, q_3)\}$$

Hence there exist four possible independent vertex sets of size two. Hence for the first iteration, the set A of Algorithm 1 may be any of the following $\{ q_0, q_2,\}$, $\{ q_0, q_3 \}$,$\{q_1, q_2, \}$, or $\{q_1, q_3 \}$.

2.2.2 Tree Interference

Tree Interference (TI) is a refinement of graph interference which allows increased parallelism, due to the properties of tree structures (Solworth and Reagan, 1994). It also uses a write conflict graph, but with several fundamental differences from graph interference. For each operation q_i , in addition to the $R(q_i)$ and $W(q_i)$, $Lev(q_i)$, is also recorded (Solworth and Reagan, 1994). When vertices are selected in the write conflict graph, preference is given to operations with smaller $Lev(q_i)$, corresponding to operations which have write accesses closer to the root (Solworth and Reagan, 1994). For this reason, the implementation of the graph must include some ordering of the vertices on $Lev(q_i)$. Due to the structure of trees, the

prohibition against read-write conflicts can be relaxed, so long as write-write conflicts are avoided (Solworth and Reagan, 1994).

Algorithm 2. Basic Tree Interference

Given a tree and a set of operations to update it, $Q=\{ q_0, q_1, q_2 , ...\}$.

While $(Q \neq \varnothing)$ do

1. $\forall\ q_i \in Q$, compute $R(q_i)$ and $W(q_i)$
2. Construct the conflict graph, $G= (V, E)$ for Q as follows:
   $V = Q$
   $E= \{ (q_i , q_j) \mid q_i , q_j \in Q , q_i \neq q_j \wedge (W(q_i) \cap W(q_j) \neq \varnothing)\}$
3. Select $A \subseteq Q$ such that $(\forall\ a_i , a_j \in A, (a_i , a_j) \notin E)$, with preference to minimal $Lev(a_i)$
4. $Q = Q - A$
5. Perform the operations in A

EndWhile;

Example: In order to compare result of one iteration, consider the binary tree depicted in Figure 2 above, with the same Q and notation as in the example for graph interference above. The read sets, write sets, and Lev ( ) for each operation are computed as follows:

$R(q_0) = \{N[50],N[25]\}$ $W(q_0) = \{N[25]\}$ $Lev(q_0) = 1$

$R(q_1) = \{N[50],N[25],N[35]\}$ $W(q_1) = \{N[35]\}$ $Lev(q_1) = 2$

$R(q_2) = \{N[50],N[65]\}$ $W(q_2) = \{N[65]\}$ $Lev(q_2) = 1$

$R(q_3) = \{N[50],N[65]\}$ $W(q_3) = \{N[65]\}$ $Lev(q_3) = 1$

The conflict graph may then be constructed for tree interference as follows, which is also depicted in Figure (3) b. above:

$$V = \{ q_0, q_1, q_2, q_3 \}$$

$$E = \{(q_2, q_3), (q_3, q_2)\} = \{(q_2, q_3)\}$$

Note that while operations q_0, and q_1 conflict under graph interference, they do not conflict under tree interference, since their write sets are disjoint. Hence any independent vertex set may include both of these operations. q_2, and q_3 do conflict however, and since neither has a higher Lev( ), either may be selected in the first iteration. Hence the first iteration may have A = { q_0, q_1, q_2 } or { q_0, q_1, q_3 }, with fifty percent more parallelism than in the case graph interference above.

3. Empirical Measurements of Achievable Parallelism and Cost

The amount of achievable parallelism, and the associated cost depend upon a number of factors, and therefore do not lend themselves to simple analytical models. Hence in order to determine the relationship between parallelism and overhead, empirical values from simulated runs had to be gathered.

3.1 Properties of the Write Sets of AVL Tree Insertions

The general nature of tree insertions was determined by running a series of test insertions on an AVL tree (without actually modifying the tree) and recording the write set for each operation. The write sets where then used to develop models for estimating the relationship between Lev( ) and the probability of writing the root node versus the tree size. Essentially, these test were performed to confirm the viability of parallel execution of AVL tree insertions.

3.1.1 Empirical Measurement of Mean Lev(q_i) Versus Tree Size

Since Lev (q_i) determines the level within the tree at which mutual exclusion must be maintained, it determines the amount of parallelism achieved. For example, for a given operation q_i , if Lev(q_i) is 0, then the root pointer or node is modified (write accessed), and hence under GI, q_i must be performed without other operations in parallel. Before analytical models of achievable parallelism can be developed, the behavior of Lev (q_i) is measured and modeled.

For the case of integer value AVL tree insertions, the mean value of Lev (q_i) was measured for 8,192 operations on sixteen unique trees of each size: 32; 64; 128; 256; 512; 1,024; 2,048; 4,096; and 8,192; with a total of 131,072 operations on each tree size. For each tree size, the mean and standard deviation of Lev (q_i) were computed, and the results were regressed by least square fit against the log of the tree size. The results of the least square fit of the mean Lev (q_i) versus the base two logarithm of the tree size gave an R^2 of approximately 0.9981, indicating a very close to linear relationship. Surprisingly, the standard deviation of Lev (q_i) remains relatively constant, increasing from 1.1755 to 1.6149 (increase of about 37.4 percent) with an increase of tree size from 32 to 8,192 (an increase of about 25,500.0 percent). The empirical data and least square fit model are summarized in table (I) below, and in graphed

in figure (4) below. In addition, the resulting model of the mean Lev (q_i) from the least square fit is given in equation (9) below, where S is the size of the tree.

$$\text{LSF Mean Lev } (q_i) = ((0.9997) \log_2 (S)) - 3.0236 \qquad (9)$$

Clearly this relationship is simple enough to be used in statistical models for AVL insertions. Since the most leaves are at a distance of $\log_2 (S) - 1$ from the root, this mean figure implies that approximately fifty percent of all operations write only within two levels of the leaves, and hence that the vast majority of the activity takes place near the leaves.

Table I: MEAN EXPERIMENTAL VALUES FOR LEV(q_i)

Tree Size S	$\log_2(S)$	Mean Lev(q_i)	StandardLSF Deviation	Lev(q_i)
32	5	2.1969	1.1755	1.9747
64	6	2.7385	1.3951	2.9744
128	7	3.9953	1.3711	3.9740
256	8	4.9603	1.4654	4.9737
512	9	5.9452	1.5596	5.9733
1,024	10	6.9184	1.6224	6.9730
2,048	11	8.0100	1.5356	7.9727
4,096	12	8.9870	1.5987	8.9723
8,192	13	10.0086	1.6149	9.9720

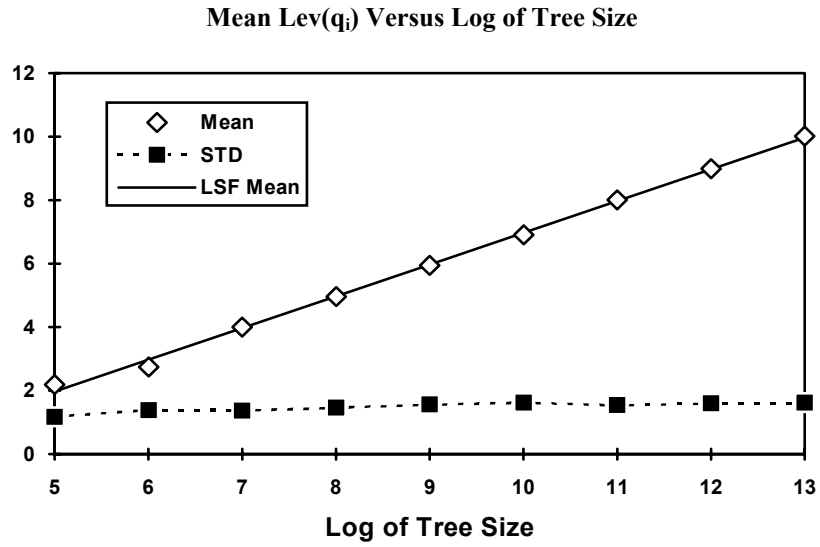

Figure 4. Graph of Mean and Standard Deviation of Lev (q_i) Versus Log of Tree Size

3.1.2 Probability of Writing the Root

Closely related to Lev (q_i) is the probability of writing the root pointer or root node. Specifically, for a given operation q_i ,this is the probability P[Lev (q_i) = 0]. In fact, when this probability is derived from histograms of the experimental data for the mean Lev (q_i) above, it can be subjected to a least square fit against 1/S and found to have an R^2 of 0.9754, strongly suggesting that it is inversely proportional to the size of the tree, S. The resulting relation is given in formula (10) below. The data and least square fit approximations are summarized in table (II) below, and graphed in figure (5).

$$\text{LSF P[Lev } (q_i) = 0](S) = 4.6315\ (1/S) + 0.003769 \qquad (10)$$

Table II: EXPERIMENTAL AND LSF PROBABILITIES FOR LEV(q_i) =0

Tree Size S	$\log_2(S)$	1/S	Measured P[Lev (q_i) = 0]	LSF -V- 1/S
32	5	0.031250	0.139565	0.148502
64	6	0.015625	0.091133	0.076136
128	7	0.007813	0.047867	0.039952
256	8	0.003906	0.016479	0.021861
512	9	0.001953	0.017365	0.012815
1,024	10	0.000977	0.007385	0.008292
2,048	11	0.000488	0.000954	0.006031
4,096	12	0.000244	0.000900	0.004900
8,192	13	0.000122	0.001175	0.004334

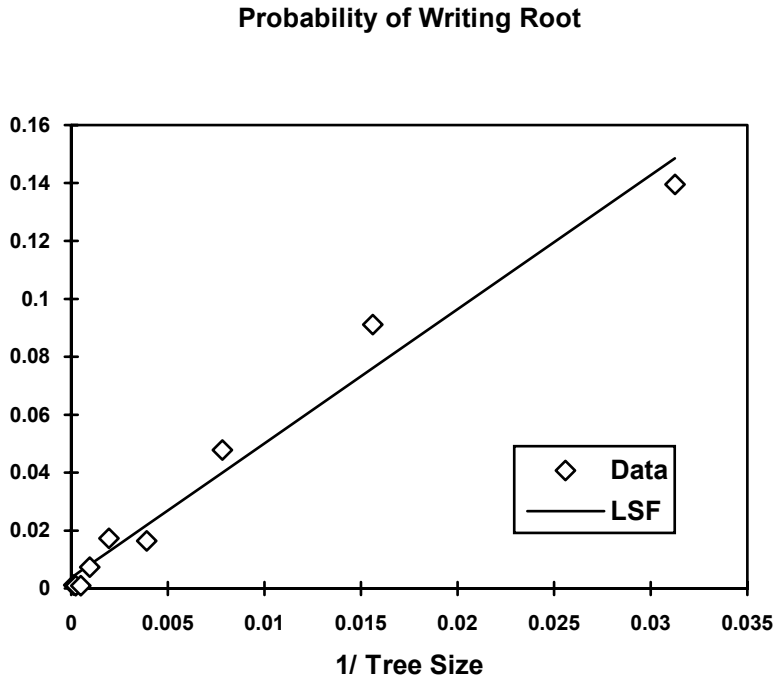

Figure 5. Graph of Probability to Write Root Pointer or Node

3.2 Parallelism and Costs of AVL Insertions under TI and GI

In general, the number of operations which may be performed in parallel is influenced by a number of factors. These include: the size of the tree, the total number of operations to be performed, and the type of interference resolution used. The number of trapdoors (locations where inserts may occur) is directly linear on the size of the tree. As the number of operations to test increases, the probability of finding operations for each trapdoor increases. These two factors are parameters of an instance however, and hence do not enter into the design of an unbounded, general case. However, the type of interference resolution techniques used can be selected.

In order to compare the performances of Tree Interference and Graph Interference, sample runs, using identical trees and data, were executed for each algorithm. For each tree size S in: 32; 64; 128; 256; 512; 1,024; 2,048; and 4,096; and sets of operations where selected of size S/8, S, and 8S. For each tree size and number of operations, ten runs where executed. Measurements where taken of the total number of phases, the total number of operations completed for phase, and the total cost of each operation in read phases. The results are summarized in the following sections.

3.2.1 Mean Number of Phases

Table (III) below summarizes the experimental mean number of phases (iterations) required to complete the entire set of insertions using each algorithm. Furthermore, these results where subjected to least square fit analysis and the corresponding data values and regression lines are plotted in figures (6) and (7) below. In general, for a given algorithm, the number of phases required was found to be linear on the logarithm of the tree size, when the number of operations to tree size ratio was preserved. In addition, the linear equations resulting from the least square fit analysis are given, along with their respective correlation coefficients in equations (11) through (16). Note that the R^2 coefficients are all above 0.98, strongly implying linear relationships.

Table III: MEAN NUMBER OF PHASES

Initial Tree Size (S)	$Log_2(S)$	Tree Interference S/8	Tree Interference S	Tree Interference 8S	Graph Interference S/8	Graph Interference S	Graph Interference 8S
32	5	1.7	6.9	24.2	1.9	8.6	35.4
64	6	2.1	7.5	25.1	2.5	12.2	45.2
128	7	2.7	8.8	27.0	3.2	15.2	53.5
256	8	2.7	10.0	27.9	4.0	18.7	63.5
512	9	3.2	10.8	30.0	5.1	22.6	72.5
1024	10	3.8	12.2	31.3	6.2	25.2	80.9
2048	11	4.0	12.7	33.0	6.8	28.7	92.8
4096	12	4.6	13.6	35.4	7.5	32.0	101.6

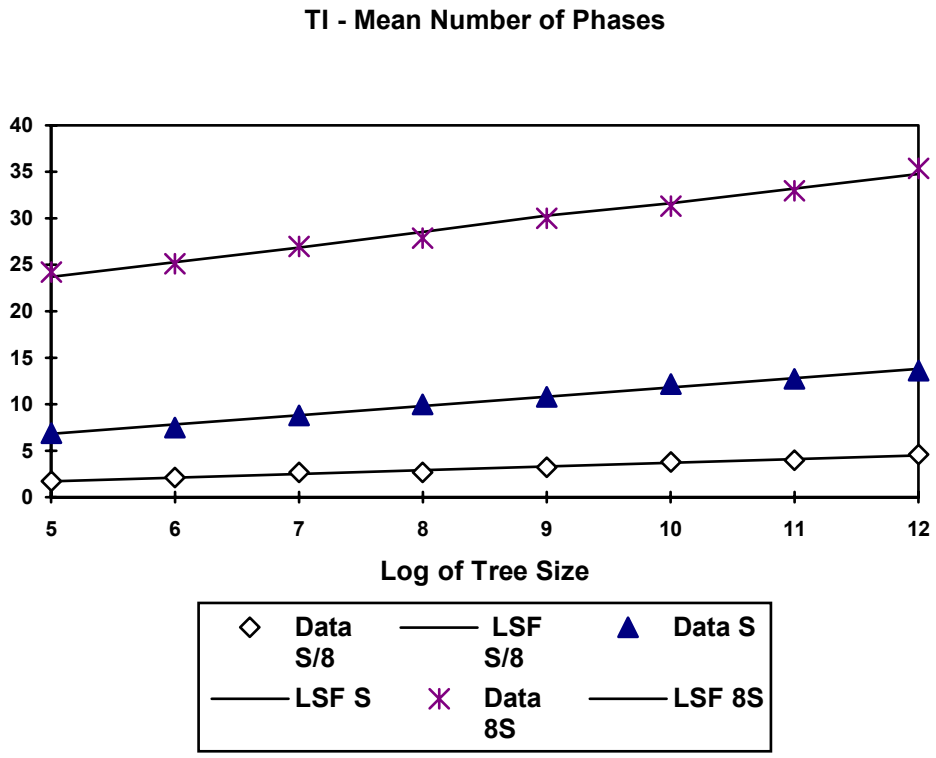

Figure 6. Graph of Mean Number of Phases, TI

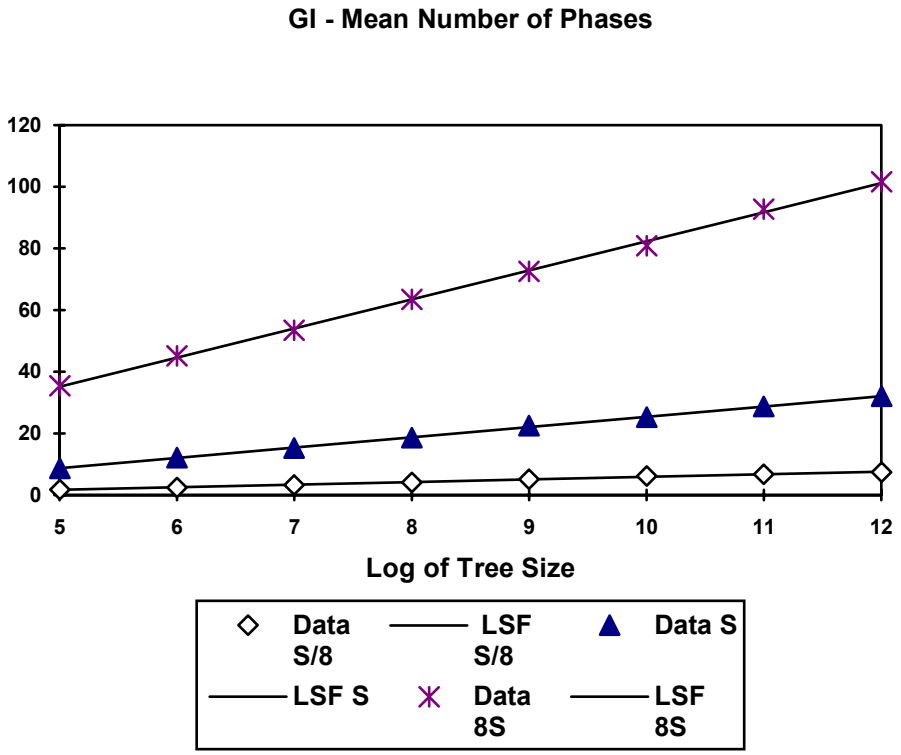

Figure 7. Graph of Mean Number of Phases, GI

LSF Number of Phases (TI, S/8) = (0.4000) $\log_2$(S) -0.3000 (R^2 = 0.9825) (11)

LSF Number of Phases (TI, S) = (0.9988) $\log_2$(S) + 1.8226 (R^2 = 0.9988) (12)

LSF Number of Phases (TI, 8S) = (1.5821) $\log_2$(S) +15.7892 (R^2 = 0.9894) (13)

LSF Number of Phases (GI, S/8) = (0.8445) $\log_2$(S) -2.5309 (R^2 = 0.9929) (14)

LSF Number of Phases (GI, S) = (3.3357) $\log_2$(S) - 7.9535 (R^2 = 0.9991) (15)

LSF Number of Phases (GI, 8S) = (9.4357) $\log_2$(S) -12.0286 (R^2 = 0.9989) (16)

3.2.2 Operations Per Phase

A more intuitive measurement of parallelism is number of operations performed in each phase (iteration). This directly measures the number of operations which may be performed in parallel. For the same runs as above, the mean number of writes per phase was tabulated, and the results are given below in table (IV). Similarly, the data was plotted in figures (8) and (9), along with least square fit regressions against the tree size S. The resulting equations from these regressions are given in equations (17) through (22). In addition, the R^2 coefficients are all above 0.99, strongly implying linear relationships.

For small numbers of operations, Tree Interference has negligible gains. For example, for the case of four insertions on a tree of thirty two nodes, only 0.2 of an additional operation (about nine percent) increase in parallelism was achieved. However, in the case of large numbers of operations, the additional parallelism begins to dominate. In the case of 32,768 operations on a tree of 4,096 nodes, and increase of about 186 percent was achieved.

Table IV: NUMBER OF OPERATIONS PER PHASE

Initial Tree		Tree Interference			Graph Interference		
Size (S)	$Log_2(S)$	S/8	S	8S	S/8	S	8S
32	5	2.4	4.6	10.6	2.2	3.8	7.3
64	6	3.8	8.5	20.4	3.3	5.3	11.4
128	7	5.9	14.5	37.9	5.1	8.5	19.2
256	8	11.9	25.6	73.4	8.2	13.7	32.4
512	9	20.0	47.4	136.5	13.1	22.8	56.6
1024	10	33.7	83.9	261.7	21.0	40.8	101.4
2048	11	64.0	161.3	496.5	37.9	71.5	176.6
4096	12	111.3	301.2	925.6	69.4	128.2	322.9

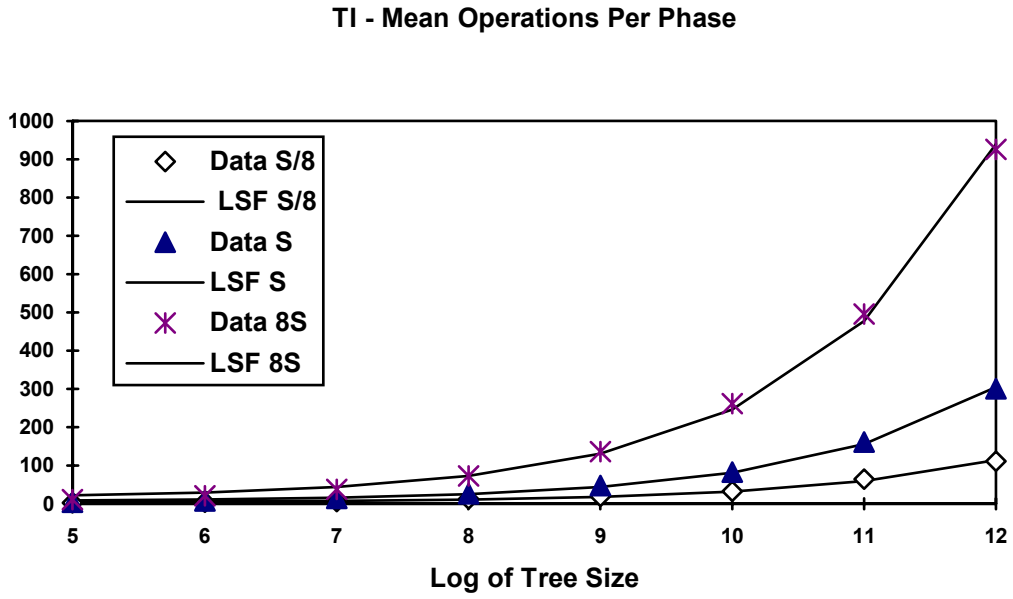

Figure 8. Graph of Mean Operations per Phase, TI

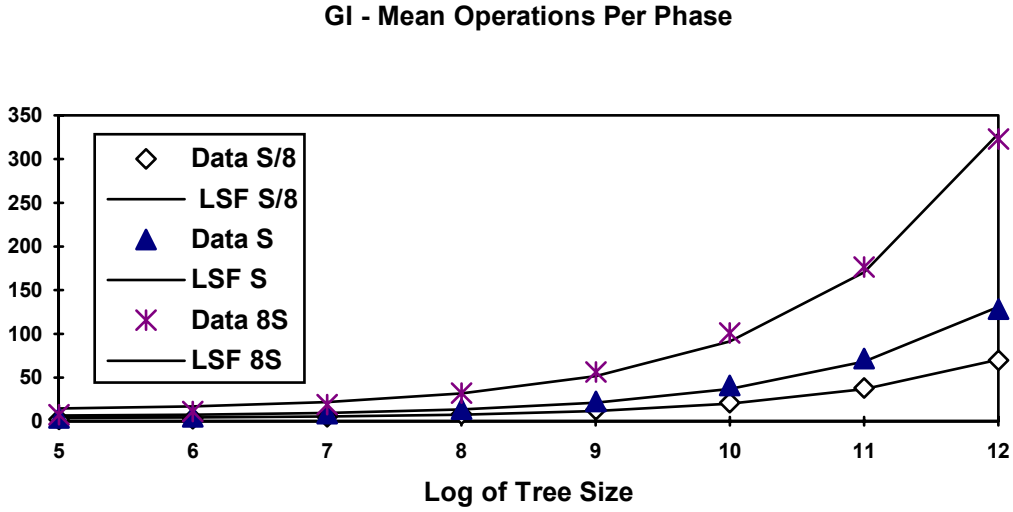

Figure 9. Graph of Mean Operations per Phase, GI

LSF Operations per Phase (TI, S/8) = (0.0269) S +4.1787 (R^2 = 0.9946) (17)

LSF Operations per Phase (TI, S) = (0.0728) S + 6.5859 (R^2 = 0.9988) (18)

LSF Operations per Phase (TI, 8S) = (0.2257) S + 15.1738 (R^2 = 0.9985) (19)

LSF Operations per Phase (GI, S/8) = (0.0164) S + 3.3505 (R^2 =0.9976) (20)

LSF Operations per Phase (GI, S) = (0.0306) S + 5.6098 (R^2 =0.9963) (21)

LSF Operations per Phase (GI, 8S) = (0.0774) S + 12.052 (R^2 = 0.9965) (23)

3.2.3 Mean Cost per Operation

In both graph interference and tree interference, the read phases of all uncompleted operations must be performed during each iteration. This is required since the read and write sets become out of date each time the tree is modified. Furthermore, the write phase of each operation is only performed once, and can be executed quickly using stored information acquired during the previous read phase. Hence the execution of the read phases requires the most time for most operations. For this reason, ***cost*** was measured in the mean number of times an operation's read phase was performed.

Table (V) summarizes the mean cost per operation in read phases for executions using the same standard tree sizes and operations set sizes as above. These are also plotted in figures (10) and (11) , and the least square fit regressions performed on them are summarized in equations (24) through (29). From the plots, they all appear to be linear with respect to the base two logarithm of the tree size, but the R^2 values are all under 0.98 except for the tree interference, 8S operations case. However, as the number of operations to perform increases with respect to the tree size, the R^2 values approach one. This suggests that the relationship becomes more linear as the number of operations (and hence the sample size) increases.

Table V: MEAN COST IN READ PHASES PER OPERATION

Initial Tree Size (S)	$Log_2(S)$	Tree Interference S/8	Tree Interference S	Tree Interference 8S	Graph Interference S/8	Graph Interference S	Graph Interference 8S
32	5	1.3	3.1	10.0	1.2	4.3	19.1
64	6	1.3	3.5	10.3	1.5	5.5	24.1
128	7	1.5	3.5	10.3	1.6	6.6	27.6
256	8	1.4	3.7	10.4	1.7	7.2	31.8
512	9	1.4	3.6	10.6	1.9	8.0	35.8
1024	10	1.4	3.6	10.6	2.0	8.4	38.7
2048	11	1.4	3.6	10.8	2.1	9.2	43.1
4096	12	1.4	3.6	10.8	2.1	9.3	46.0

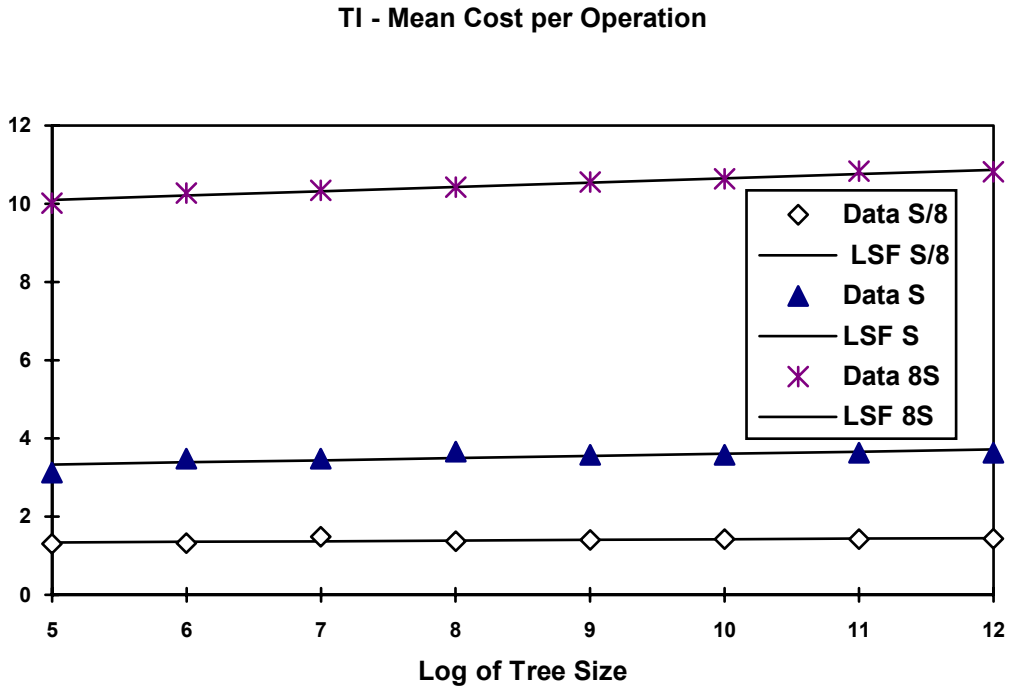

Figure 10. Graph of Mean Cost per Operation, TI

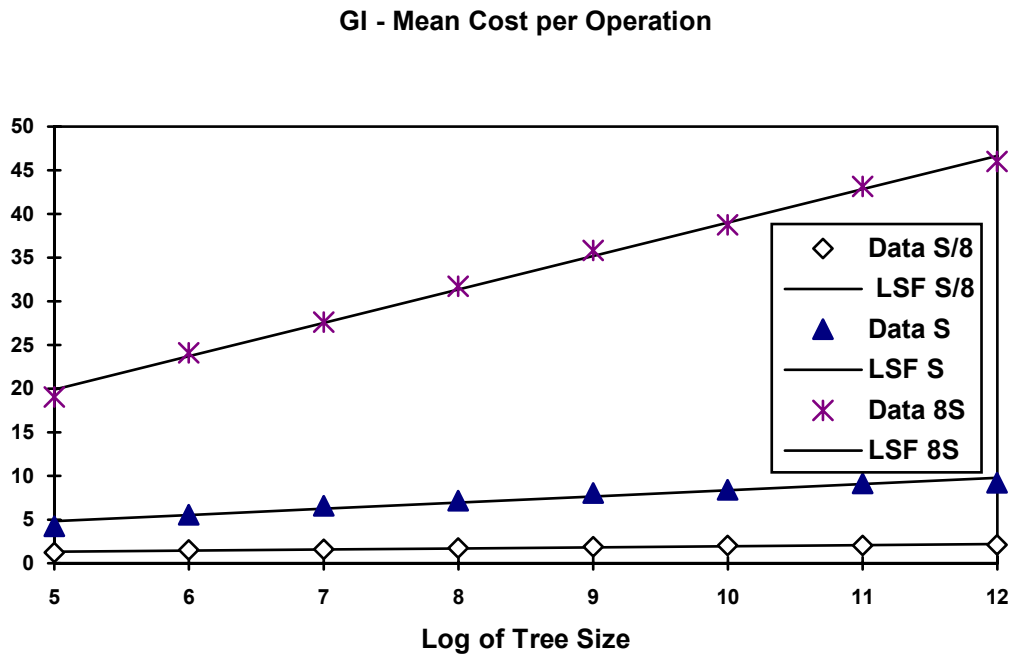

Figure 11. Graph of Mean Cost per Operation, GI

$$\text{LSF Cost per Operation (TI, S/8)} = (0.0155)\log_2(S) + 1.2660 \quad (R^2 = 0.3761) \qquad (24)$$

$$\text{LSF Cost per Operation (TI, S)} = (0.0551)\log_2(S) + 3.0577 \quad (R^2 = 0.5748) \qquad (25)$$

$$\text{LSF Cost per Operation (TI, 8S)} = (0.1108)\log_2(S) + 9.5442 \quad (R^2 = 0.9619) \qquad (26)$$

$$\text{LSF Cost per Operation (GI, S/8)} = (0.1237)\log_2(S) + 0.7146 \quad (R^2 = 0.9383) \qquad (27)$$

$$\text{LSF Cost per Operation (GI, S)} = (0.7079)\log_2(S) + 1.2909 \quad (R^2 = 0.9572) \qquad (28)$$

$$\text{LSF Cost per Operation (GI, 8S)} = (3.8252)\log_2(S) + 0.7636 \quad (R^2 = 0.9968) \qquad (29)$$

3.2.4 A Comparison of Execution Traces

The differences in the specific behavior of the Graph and Tree Interference algorithms on a phase by phase basis must be understood. For ten pairs of trees and queues, each of size 512 and 4,096 respectively, runs of both TI and GI scheduling where performed.

The mean operations performed in each phase by each algorithm were computed, and plotted in figure (12) below. In tree interference, the number of operations per phase increases as a crisp curve as the tree grows, followed by a step drop, and then a slow trailing off. In contrast, the number of operations completed per phase for graph interference is a very noisy, shallow curve. This is possibly do to the fact that disastrous levels of parallelism occur when writes near the root occur. A simple plot of the fraction of operations completed versus phase for the executions run is included as figure (13). In addition, the data points plotted in figure (13) are listed in table (VI). Note that only every fifth phase is plotted due to legibility problems. These plots indicates that for both algorithms, the work performed per phase accelerates to a constant rate, runs to a peak, and then tapers off slowly.

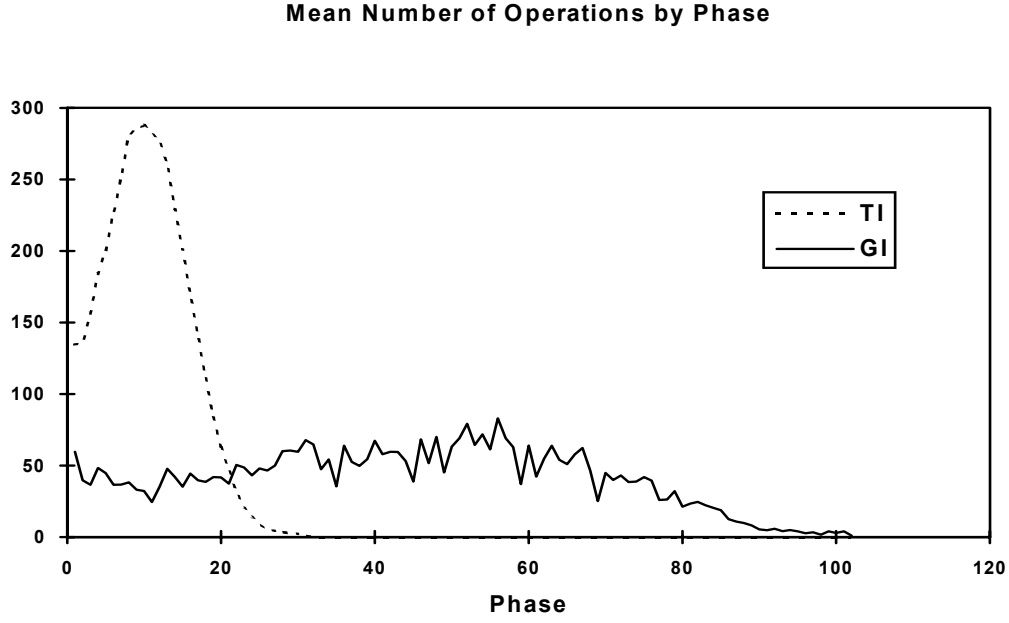

Figure 12. Graph of Operations by Phase

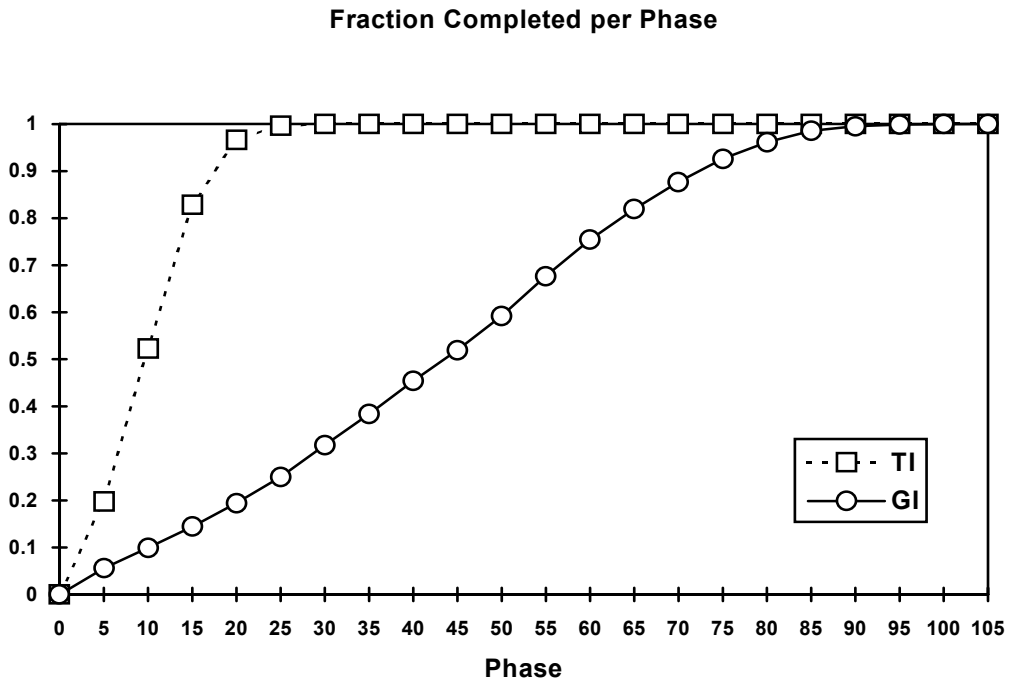

Figure 13. Graph of Fraction of Operations Completed

Table VI: SAMPLE FRACTIONS OF OPERATIONS COMPLETED

Phase	TI	GI	Phase	TI	GI
0	0.0000	0.0000	55	1.0000	0.6770
5	0.1979	0.0560	60	1.0000	0.7542
10	0.5229	0.0996	65	1.0000	0.8192
15	0.8281	0.1442	70	1.0000	0.8771
20	0.9668	0.1946	75	1.0000	0.9264
25	0.9967	0.2502	80	1.0000	0.9614
30	1.0000	0.3178	85	1.0000	0.9854
35	1.0000	0.3836	90	1.0000	0.9955
40	1.0000	0.4540	95	1.0000	0.9989
45	1.0000	0.5197	100	1.0000	0.9999
50	1.0000	0.5926	105	1.0000	1.0000

4. Dynamic Window Sizing Algorithms

4.1 Motivation

In both the tree interference and graph interference algorithms, the realized parallelism, as measured by the mean number of operations completed per phase, grows linearly on the size of the tree. This is consistent with the number of trapdoors (free pointers) which is also linear on the size of the tree. Similarly, the mean cost per operation (in read phases) grows as a logarithm of the size of the tree. This is consistent with the lengths of the access paths, which run from the root down to the leaves. Longer access paths introduce more potential sites for conflict, resulting in an increase in the number of conflicts, and the number of required read phases. Both algorithms give scaleable parallelism at an increasing cost per operation. In general, tree interference produces more parallelism than graph interference, at a lower cost. However, these differences are only in the leading coefficients of the growth rates. The real advantage of tree interference is the stability of its execution traces, due to its relative robustness with respect to write accesses near the root. Hence both graph and tree interference are viable parallelization techniques, with tree interference yielding superior results in the general case.

Under certain circumstances the increasing cost per operation may be prohibitively expensive. Ideally, the mean cost per operation should be constant, or limited by a constant upper bound. If this can be achieved, then it is possible to arbitrarily scale up the parallelism, without any limits due to cost.

Regardless of the amount of time required for each test read phase, the number of test read phases required is linear on the number of operations considered. Up to this point, cost has been measured in test read phases, and has not considered the scheduling costs. Both algorithms require some variation of either graph coloring or independent vertex set, the optimal cases of which are well known to be ***NP***-Complete (Garey and Johnson, 1979). In addition, non-optimal versions require quadratic time to perform pair-wise comparisons to determine which operations conflict. This will be shown in the lemma below. The significance of equation (30) can be made clear with the following example.

Lemma: Given a set of Q operations to select a conflict free set from, in which all elements must be considered, O ($|Q|^2$) access path comparisons may be required.

Proof. The number of unique pairs of distinct elements from Q is :

$$\binom{|Q|}{2} = \frac{|Q|!}{(|Q|-2)!(2)!} = \frac{|Q|(|Q|-1)}{2} = O\left(|Q|^2\right) \tag{30}$$

Example: Consider a set of ten operations to schedule, where none of the operations conflict. If these are scheduled in one set, the number of comparisons required is :

$$\binom{10}{2} = 45 \text{ comparisons.}$$

If the ten operations are taken as two sets of five operations instead, the number of comparisons is :

$$2\binom{5}{2} = 2\,(10) = 20 \text{ comparisons.}$$

By considering the operations in some subset of Q on each iteration, both the cost in read phases and scheduling costs may be reduced. Such a subset will be called a ***window*** into Q, and be denoted by V.

4.2 Dynamic Window Sizing

From figure (12), it is clear that the amount of parallelism varies from phase to phase. Hence the size of the window V must be able to expand quickly to accommodate increases in parallelism, decrease quickly enough to prevent excessive costs, and yet remain stable enough to facilitate parallelism and control costs. A ***dynamic window sizing algorithm*** is an algorithm that determines the size of the window, based on information that is available at run time.

Let v_i denote the size of a window V on iteration i of some scheduling algorithm. Let v_1 be the initial window size, which is used in the first iteration, and is considered a parameter of the execution. Let

A_i denote the number of operations performed (selected) in iteration i. Hence for a given scheduling algorithm, executing for x iterations, the mean cost per operation is defined by equation (31) below.

$$\text{cost per operation} = \frac{\sum_{i=1}^{x} v_i}{\sum_{i=1}^{x} A_i} \qquad (31)$$

4.3 The WSPTI Dynamic Window Sizing Algorithm

The ***weighted sum of the previous two iterations*** (WSPTI) dynamic window sizing algorithm calculates v_i as the sum of the number of operations performed during the previous two iterations, multiplied by a weight factor. This weight factor is denoted as γ , and is considered to be a parameter of the execution. Note that γ must be greater than 0.50, since otherwise the window size might shrink to one, and the algorithm would be unable to increase its size. This has two desirable properties. First, the use of the sum of previous two iterations' completed operations allows it to be responsive, but also adds stability. Second, the use of the weight factor allows the mean cost per operation to be tightly controlled by an upper bound. The value of v_i is computed using equation (32).

$$v_i = \begin{cases} v_1 & \text{if } i = 1 \\ \gamma\,(v_1 + A_1) & \text{if } i = 2 \\ \gamma\,(A_{i-2} + A_{i-1}) & \text{if } i > 2 \end{cases} \qquad (32)$$

In the case of the final few iterations of a scheduling algorithm, the computed v_i may exceed the number of remaining elements in Q. In this case, v_i represents a limit on the size of V, not its actual size. By substituting (32) into (31), and taking the limit as the number of iterations (x) goes to infinity, the upper bound on the mean cost per operation can be computed. Note that since at least one operation is performed on each iteration, $A_i \geq 1$. Consequently, the sum of A_i is strictly increasing , which allows it to dominate the other terms in the limit. The derivation of this limit is given here as equation (33) which summarizes

the relationship between the predicted cost per operation and the weight factor γ. Using equation (33), γ may be selected to set the mean cost per operation.

$$\lim_{x\to\infty}(\text{mean cost/operation}) = \lim_{x\to\infty} \frac{v_1 (1+\gamma) + 2\gamma \sum_{i=1}^{x} A_i}{\sum_{i=1}^{x} A_i} = 2\gamma \qquad (33)$$

4.4 Performance of WSPTI

In order to determine the performance of the WSPTI dynamic window sizing algorithm, two sets of tests were performed. The first was to determine how much parallelism was lost under graph and tree interference when dynamic window sizing was used. The second was performed to test how closely the mean cost per operation followed the theoretical limit of 2γ.

4.4.1 Loss of Parallelism

In order to determine how much parallelism was lost, simulations were run for the same range of tree sizes as earlier experiments, with $|Q| = 8S$, $v_1=2$, and $\gamma = 1.0$ Ten simulations were run for each tree size, and the average results for the mean number of operations per phase are tabulated in tables VII and VIII for graph interference and tree interference respectively. In addition, these data were regressed against the size of the tree, and the results are given in equations (34) and (35). The mean number of operations per phase remains linear on the size of the tree under the addition of dynamic window sizing.

Table VII: EXPERIMENTAL DATA FOR GI, WSPTI

Size of Tree (S)	|Q|	Data Operations Per Phase	LSF Operations Per Phase	Mean Cost Per Operation
32	256	3.74	6.71	1.89
64	512	6.25	8.28	1.92
128	1,024	10.17	11.42	1.91
256	2,048	18.42	17.69	1.92
512	4,096	32.08	30.24	1.93
1,024	8,192	58.94	55.33	1.93
2,048	16,384	107.86	105.52	1.93
4,096	32,768	203.65	205.90	1.94

Table VIII: EXPERIMENTAL DATA FOR TI, WSPTI

Size of Tree (S)	|Q|	Data Operations Per Phase	LSF Operations Per Phase	Mean Cost Per Operation
32	256	8.86	15.13	1.75
64	512	15.33	20.59	1.75
128	1,024	29.01	31.50	1.74
256	2,048	55.80	53.33	1.75
512	4,096	100.64	96.98	1.75
1,024	8,192	191.85	184.28	1.75
2,048	16,384	364.09	358.88	1.75
4,096	32,768	703.18	708.08	1.75

LSF (Mean Operation Per Phase, GI, WSPTI) = 0.0490 (S) + 5.1444 (R^2 = 0.9988) (34)

LSF (Mean Operation Per Phase, TI, WSPTI) = 0.1705 (S) + 9.6753 (R^2 = 0.9995) (34)

The lost parallelism can be estimated by dividing the (S) coefficients for the WSPTI equations (34) and (35), by those equations (23) and (19) respectively. By doing so, we find that graph interference retains about 63.30 percent of its mean number of operations per phase when WSPTI is used, and that tree interference retains about 75.54 percent.

4.4.2 The Mean Cost per Operation Versus Weight

In order to determine how closely the measured mean cost per operations approaches the theoretical limit, for both tree and graph interference, ten simulations for each weight (γ) of 0.75, 1.00, 1.25, 1.50, 1.75, and 2.00 where run for trees of size 4,096 with |Q| = 32,768, and v_1 =2. The mean values were computed for each weight and the results tabulated in table IX. In addition, these values were plotted in figure (14). The measured values where found to be less than the predicted values. This is reasonable, considering that the predicted values are for an execution using an infinite number of iterations.

Table: IX: MEAN COST PER OPERATION VERSUS WEIGHT

Weight (γ)	Theoretical	Measured GI	Measured TI
0.75	1.50	1.4463	1.3651
1.00	2.00	1.9421	1.7502
1.25	2.50	2.4030	2.1361
1.50	3.00	2.8847	2.5204
1.75	3.50	3.3482	2.8908
2.00	4.00	3.8060	3.2455

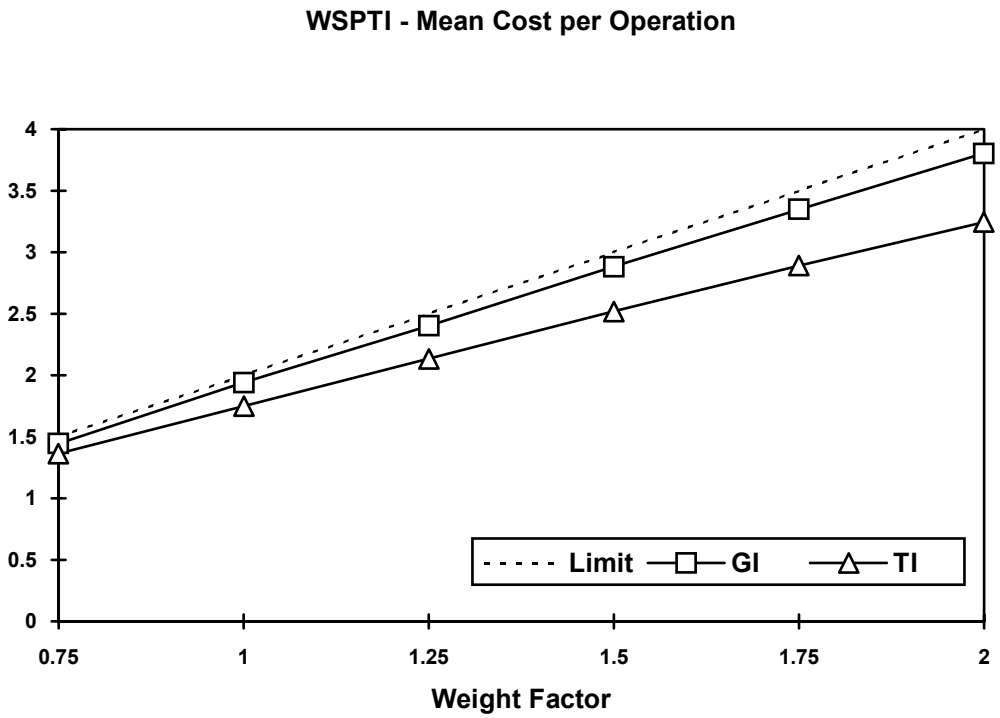

Figure 14. Graph of Mean Operations per Phase in WSPTI

5. Conclusions and Future Work

5.1 Conclusions

The following conclusions have been reached.

AVL tree insertions have been shown to be well suited for parallel execution, since the majority of write accesses occur near the leaves. Any tree structure which has this property should be similarly well suited.

Both the graph interference and tree interference methods are effective techniques for determining which operations may be performed simultaneously on tree structures. They both give parallelism, as measured by the mean number of operations performed per phase, which is linear in the size of the tree. This corresponds with the number of free pointers, which is also linear in the size of the tree. Similarly, they both incur a mean cost per operation which grows as a logarithm of the size of the tree. This corresponds to the length of the path from the root to a free pointer, which provides potential sites for conflicts. Tree interference exploits the structure of the tree, and therefore provides greater parallelism and a reduced cost in the general case. Furthermore, since tree interference is more robust with respect to write accesses, it produces more consistent amounts of parallelism each phase.

The use of a dynamically sized window algorithm provides an effective technique for controlling the mean cost per operation, at the cost of slightly reduced parallelism. Experimental confirmation of the WSPTI dynamic window sizing algorithm demonstrated that for a loss of about 24 percent of operations completed per phase in tree interference, and a loss of about 37 percent in graph interference, the mean cost per operation could be bounded by a constant.

5.2 Future Work

Several areas present themselves for future research and experimentation. Further work regarding self adjusting scheduling algorithms is required. For example, additional window sizing techniques should be investigated. In addition, parallel operations upon tree structures which do not have the majority of write accesses near the leaves, will not work well using the graph interference or tree interference algorithms. A typical tree which would require different techniques are those which are constructed form the leaves up, such as those used in Huffman encoding. Techniques for use with these types of trees should be developed.

Cited Literature

Amdahl, G. M.: Validity of the single processor approach to achieve large scale computing capabilities. Proceedings of the AFIPS 1967 Spring Joint Computer Conference 30:483-5, 1967.

Baase, Sara: Computer Algorithms: Introduction to Design and Analyses, Second Edition. Reading, Addison-Wesley Publishing Company, 1988.

Carlisle, M. C., Rogers, A., Reppy, J. H., and Hendren, L. J.: Early experiences with olden. Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing 1-20, 1994.

Christianson, B.: Amdahl's law and the end of system design. ACM Sigmetrics Performance Evaluation Review 19:2:30-2, 1991.

Garey, M. R. and Johnson, D. S. : Computers and Intractability: a Guide to the Theory of NP Completeness. New York, W. H. Freeman and Company, 1979.

Larus, J. R,. and Hilfinger, P. N.: Detecting conflicts between structure accesses. Proceedings of the Sigplan 88 Conference on Programming Language Design and Implementation. 21-34, 1988.

Leighton, F. H.: Introduction to Parallel Algorithms and Architectures: Arrays ° Trees ° Hypercubes. San Mateo, Morgan Kaufman Publishers, 1992.

Quinn, M. J. and Deo, N.: Parallel graph algorithms. ACM Computing Surveys 16:3:319-48, 1984.

Reingold, E. M. and Hansen, J. H.: Data Structures in Pascal . Boston, Little, Brown and Company, 1986.

Solworth , J. A. and Reagan, B. B.: Arbitrary order operations on trees. Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing 21-36, 1994.

Solworth , J. A. and Reagan, B. B.: Parallelizing tree algorithms: overhead vs. parallelism. Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing (To Appear), 1995.

Tow, A. S. : Parallelization Strategies for Tree Algorithms: the Independent Write Set Approach. Masters project, University of Illinois at Chicago, Chicago, 1991.

APPENDIX

The Table X, is included here to summarize the important symbols and notations used in this thesis

Table X: SUMMARY OF NOTATION USED

α	Fraction of work which may be executed in parallel
A	A set of operations selected in an iteration of a scheduling algorithm, a subset of Q
A_i	For iteration i, the size of A (i.e.. The number of operations performed in iteration i.)
a_i	An operation in A
I_n	A problem instance with input size n
ε (I_n , p)	Efficiency for problem instance I_n on p processors
ϕ (I_n , p)	Overhead for problem instance I_n on p processors
γ	Weight factor used in computing window size in the WSPTI algorithm
GI	Graph interference exclusion control
K	The set of valid key values for a tree, denoted as the range [K-, K+]
K+	The maximum legal key value
K-	The minimum legal key value
Lev (q_i)	Adjusted maximum height in W (q_i) for operation q_i
n	Size of an input for a problem instance
P (q_i)	The access path for operation q_i
p	Number of processors
Q	A queue of operations to be performed
q_i	An operation in Q
R (q_i)	The read set for operation q_i
σ (I_n , p)	Speedup for problem instance I_n on p processors
S	The size of a tree, equal to the number of nodes in it
T	Execution time (general), either T_p [I_n] or T_1 [I_n] below.
TI	Tree interference exclusion control
T_p [I_n]	Execution time of problem instance I_n on p processors
T_1 [I_n]	Execution time of problem instance I_n on uni-processor machine
V	A window, a subset of Q above, consisting of potential operations
v_i	For iteration i, the computed size limit on the window V
W (q_i)	The write set for operation q_i

VITA

NAME:	Bryan Barney Reagan
EDUCATION:	Mechanical Engineering, Valparasio University, Valparaiso, Indiana, 1986-89
	B.S., Mathematics and Computer Science, University of Illinois at Chicago, Chicago, Illinois, May 1991
	M.S., Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois, August 1995
HONORS:	Alpha Lambda Delta Honor Society, Valparaiso University, 1987.
	Gold Key Honor Society, University of Illinois at Chicago, 1991.
	Phi Kappa Phi Honor Society, University of Illinois at Chicago, 1991.
PROFESSIONAL MEMBERSHIP:	Phi Mu Alpha Professional Music Fraternity
EXPERIENCE:	Data Entry Clerk, Campos and Stratis Certified Public Accountants, Hoffman Estates, Illinois, Seasonal work 1987-90.
	Co-op Engineer, Technology Division, Inland Steel Company, East Chicago, Indiana, 1988-89.
	Lab Assistant, Material Science Lab, Valparaiso University, Valparaiso, Indiana, 1989.
	Research Assistant, Parallel Systems Lab, University of Illinois at Chicago, Chicago, Illinois, 1992-Present.
PUBLICATIONS:	Solworth , J. A. and Reagan, B. B.: Arbitrary Order Operations on Trees. Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing 21-36, 1994.
	Solworth , J. A. and Reagan, B. B.: Parallelizing tree algorithms: overhead vs. parallelism. Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing (To Appear), 1995.
	Zhang, W., Yu, C., Reagan, B., and Nakajima, H.: Context-dependent interpretations of linguistic terms in fuzzy relational databases. Proceedings of the IEEE Data Engineering Conference (To Appear), 1995.