Evaluating Arithmetic Expressions Using Tree Contraction: A Fast and Scalable Parallel Implementation for Symmetric Multiprocessors (SMPs) (*Extended Abstract*)

David A. Bader*, Sukanya Sreshta, and Nina R. Weisse-Bernstein**

Department of Electrical and Computer Engineering
University of New Mexico, Albuquerque, NM 87131 USA

Abstract. The ability to provide uniform shared-memory access to a significant number of processors in a single SMP node brings us much closer to the ideal PRAM parallel computer. In this paper, we develop new techniques for designing a uniform shared-memory algorithm from a PRAM algorithm and present the results of an extensive experimental study demonstrating that the resulting programs scale nearly linearly across a significant range of processors and across the entire range of instance sizes tested. This linear speedup with the number of processors is one of the first ever attained in practice for intricate combinatorial problems. The example we present in detail here is for evaluating arithmetic expression trees using the algorithmic techniques of list ranking and tree contraction; this problem is not only of interest in its own right, but is representative of a large class of irregular combinatorial problems that have simple and efficient sequential implementations and fast PRAM algorithms, but have no known efficient parallel implementations. Our results thus offer promise for bridging the gap between the theory and practice of shared-memory parallel algorithms.

Keywords: Expression Evaluation, Tree Contraction, Parallel Graph Algorithms, Shared Memory, High-Performance Algorithm Engineering.

1 Introduction

Symmetric multiprocessor (SMP) architectures, in which several processors operate in a true, hardware-based, shared-memory environment and are packaged as a single machine, are becoming commonplace. Indeed, most of the new high-performance computers are clusters of SMPs having from 2 to over 100 processors per node. The ability to provide uniform-memory-access (UMA) shared-memory

* Supported in part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, and Biocomplexity DEB-01-20709.
** Supported by an NSF Research Experience for Undergraduates (REU).

S. Sahni et al. (Eds.) HiPC 2002, LNCS 2552, pp. 63–75, 2002.

for a significant number of processors brings us much closer to the ideal parallel computer envisioned over 20 years ago by theoreticians, the *Parallel Random Access Machine (PRAM)* (see [13, 19]) and thus may enable us at last to take advantage of 20 years of research in PRAM algorithms for various irregular computations. Moreover, as supercomputers increasingly use SMP clusters, SMP computations will play a significant role in supercomputing. For instance, much attention has been devoted lately to OpenMP [17], that provides compiler directives and runtime support to reveal algorithmic concurrency and thus takes advantage of the SMP architecture; and to mixed-mode programming, that combines message-passing style between cluster nodes (using MPI) and shared-memory style within each SMP (using OpenMP or POSIX threads).

While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work—synchronization cannot be taken for granted and the number of processors is far smaller than that assumed in PRAM algorithms. The significant feature of SMPs is that they provide much faster access to their shared-memory than an equivalent message-based architecture. Even the largest SMP to date, the recently delivered 106-processor Sun Fire Enterprise 15000 (E15K), has a worst-case memory access time of 450ns (from any processor to any location within its 576GB memory); in contrast, the latency for access to the memory of another processor in a distributed-memory architecture is measured in tens of μs. In other words, message-based architectures are two orders of magnitude slower than the largest SMPs in terms of their worst-case memory access times.

The largest SMP architecture to date, the Sun E15K [5] (a system three- to five-times faster than its predecessor, the E10K [4]), uses a combination of data crossbar switches, multiple snooping buses, and sophisticated cache handling to achieve UMA across the entire memory. Of course, there remains a large difference between the access time for an element in the local processor cache (around 10ns) and that for an element that must be obtained from memory (at most 450ns)—and that difference increases as the number of processors increases, so that cache-aware implementations are even more important on large SMPs than on single workstations. Fig. 1 illustrates the memory access behavior of the Sun E10K (right) and its smaller sibling, the E4500 (left), using a single processor to visit each 32-bit node in a circular array. We chose patterns of access with a fixed stride, in powers of 2 (labeled `C, stride`), as well as a random access pattern (labeled `R`). The data clearly show the effect of addressing outside the on-chip cache (the first break, at a problem size of size greater than 2^{12} words, or 16KB — the size of L1 cache) and then outside the L2 cache (the second break, at a problem size of greater than 2^{20} words, or 4MB). The uniformity of access times was impressive—standard deviations around our reported means are well below 10 percent. Such architectures make it possible to design algorithms targeted specifically at SMPs.

Arithmetic Expression Evaluation (*AEE*) has important uses in a wide-range of problems ranging from computer algebra and evaluation of logical queries to compiler design. Its classical parallel formulation uses the technique of tree

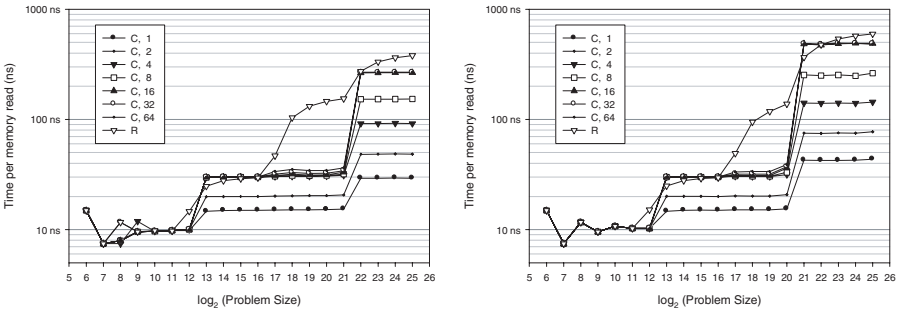


Fig. 1. Memory access (read) time using one 400 MHz UltraSPARC II processor of a Sun E4500 (left) and an E10K (right) as a function of array size for various strides

contraction when the expression is represented as a tree with a constant at each leaf and an operator at each internal vertex. AEE involves computing the value of the expression at the root of the tree. Hence, AEE is a direct application of the well studied tree contraction technique, a systematic way of shrinking a tree into a single vertex.

Miller and Reif [16] designed an exclusive-read exclusive-write (EREW) PRAM algorithm for evaluating any arithmetic expression of size n, which runs in $O(\log n)$ time using $O(n)$ processors (with $O(n \log n)$ work). Subsequently Cole and Vishkin [6] and Gibbons and Rytter [8] independently developed $O(\log n)$-time $O(n/\log n)$-processors (with $O(n)$ work) EREW PRAM algorithms. Kosaraju and Delcher [15] developed a simplified version of the algorithm in [8] which runs in the same time-processor bounds ($O(\log n)$-time $O(n/\log n)$-processors (with $O(n)$ work) on the EREW PRAM). Recently, several researchers in [3, 7] present theoretic observation that this classical PRAM algorithm for tree contraction on a tree T with n vertices can run on the Coarse-Grained Multicomputer (CGM) parallel machine model with p processors in $O(\log p)$ communication rounds with $O\left(\frac{n}{p}\right)$ local computation per round.

2 Related Experimental Work

Several groups have conducted experimental studies of graph algorithms on parallel architectures (for example, [11, 12, 14, 18, 20, 9]). However, none of these related works use test platforms that provide a true, scalable, UMA shared-memory environment and still other studies have relied on *ad hoc* hardware [14]. Thus ours is the first study of speedup for over tens of processors (and promise to scale over a significant range of processors) on a commercially available platform. In a recent work of ours ([2]) we study the problem of decomposing graphs with the ear decomposition using similar shared-memory platforms.

Our work in this paper focuses on a parallel implementation of the tree contraction technique specific to the application of expression evaluation on symmetric multiprocessors (SMPs). The implementation is based on the classic PRAM algorithm (e.g., see [15] and [13]). We begin with the formal description of the parallel algorithm implemented. Next we detail the implementation including the empirical and testing environment and present results of the experiments. The last section provides our conclusions and future work.

3 The Expression Evaluation Problem

AEE is the problem of computing the value of an expression that is represented as a tree with a constant at each leaf and an operator at each internal vertex of the tree. For a parallel formulation, we use the tree contraction technique to shrink the tree to its root. For simplicity, our study is restricted to expressions with binary associative operators. Hence, the trees are binary with each vertex u excluding the root of the tree having only one sibling $sib(u)$. This technique can also be used with expressions having non-binary operators by considering unary operators with their identity elements and converting general trees resulting from ternary operators to binary trees as a preprocessing phase of the algorithm [13]. For the sake of discussion and without loss of generality, let us assume that the internal vertices contain either the addition operator $+$ or the multiplication operator $\times$.

The simple parallel solution of evaluating each subexpression (two sibling leaves and their parent) in parallel and setting the parents of the vertices evaluated equal to the value of the subexpression until the root is reached works well when the tree is well-balanced. In the extreme case of a "caterpillar" tree, when the tree is a long chain with leaves attached to it, this solution requires a linear number of iterations similar to the sequential solution. Hence, an optimal solution should ease the above restriction that each vertex must be fully evaluated before its children can be removed.

The *rake* operation is used to remove the vertices from the tree, thus contracting it. Let T be a rooted binary tree with root r and let $p(v)$, $sib(v)$ and $p(p(v))$ represent the parent, sibling, and grandparent of a vertex v respectively. The rake operation when applied to a leaf v ($p(v) \neq r$) of the tree removes v and $p(v)$ from T and connects $sib(v)$ to $p(p(v))$. The rake operation is illustrated in Fig. 2.

The value of a vertex v, denoted by $val(v)$ is defined as the value of the subexpression (subtree rooted) at v. The value of a leaf is simply the constant value stored in the leaf. To accomplish partial evaluation, each vertex v of the tree T is associated with a label (a_v, b_v) such that the contribution of a vertex to its parent's value is given by the expression $a_v val(v) + b_v$. The label of each vertex is initialized to $(1, 0)$. Let u be an internal vertex of the tree such that u holds the operator $\oplus \in \{+, \times\}$ and has left child v and right child w. The value of vertex u is given by $val(u) = (a_v val(v) + b_v) \oplus_u (a_w val(w) + b_w)$. The value contributed by u to the vertex $p(u)$ is given by $E = a_u val(u) + b_u =$

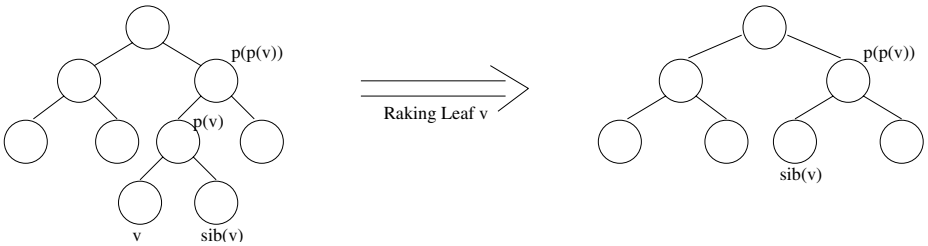


Fig. 2. Rake of leaf v: removes vertices v and $p(v)$ from the tree and makes the sibling $sib(v)$ of vertex v, the child of grandparent $p(p(v))$ of v

$a_u[(a_v val(v) + b_v) \oplus_u (a_w val(w) + b_w)] + b_u$. Say v is the left leaf of u. On raking leaf v, E is simplified to a linear expression in the unknown value of w, namely $val(w)$. The labels of w are then updated to maintain the value contributed to $p(u)$. The augmented rake operation that modifies the sibling labels as described above maintains the value of the arithmetic expression. The original binary tree is contracted to a three-vertex tree with root r and two leaves v_1 and v_2 by applying the augmented rake operation repeatedly and concurrently to the tree. The value of the arithmetic expression is then given by the value of the root which is given by $val(T) = val(r) = (a_{v_1} val(v_1) + b_{v_1}) \oplus_r (a_{v_2} val(v_2) + b_{v_2})$.

The next section formally describes the PRAM algorithm in detail.

4 Review of the PRAM Algorithm for Expression Evaluation

The PRAM algorithm consists of three main steps. The first step identifies and labels the leaves. The second step contracts the input binary tree to a three-vertex tree using the augmented *rake* operation. The resultant tree contains the root and the left- and right-most leaves of the original tree with labels suitably updated to maintain the value of the arithmetic expression. The third step computes the final value of the arithmetic expression from the three-vertex binary tree as described in the previous section. The concurrent rake of two leaves that are siblings or whose parents are adjacent must be avoided as this would lead to an inconsistent resultant structure. Hence, a careful application of the rake operation for tree contraction is required.

Alg. 4 evaluates a given arithmetic expression using the PRAM model (from [13]). After Step (2) of Alg. 4, we have a three-vertex tree T' with a root r holding an operator $\oplus_r$ and two leaves, the left- and right-most leaves, u and v containing the constants c_u and c_v with labels (a_u, b_u) and (a_v, b_v), respectively. These labels hold the relevant information from the vertices that have been raked. Hence, the value of the subexpression at the root r is the value of the given arithmetic expression. This value is given by

$$val(T) = val(T') = val(r) = (a_u c_u + b_u) \oplus_r (a_v c_v + b_v). \quad (1)$$

Data : (1) A rooted binary tree T such that each vertex v has the labels (a_v, b_v) initialized to $(1,0)$, (2) each non-leaf vertex has exactly two children, and (3) for each vertex different from the root, the parent $p(v)$, the sibling $sib(v)$ and an indication if it is a left or right child of its parent.

Result : The value of the arithmetic expression represented by the tree T

begin

1. Label the leaves consecutively in order from left to right, excluding the left- and right-most leaves, and store the labeled leaves in an array A of size n.
2. **for** $\lceil \log(n+1) \rceil$ *iterations* **do**
   - 2.1. Apply the augmented rake operation that modifies the sibling's labels concurrently to all the elements of A_{odd} that are left children.
   - 2.2. Apply the augmented rake operation that modifies the sibling's labels concurrently to the rest of the elements in A_{odd}.
   - 2.3. Set $A := A_{even}$.
3. Compute the value of the arithmetic expression from the three-vertex binary tree.

end

Algorithm 1: PRAM Algorithm for AEE

Alg. 4 ensures that two leaves that are siblings or whose parents are adjacent are never raked concurrently (for proof refer to [15]). Hence the rake operations are applied correctly and the algorithm is correct. Step (1), labeling the leaves of the tree, can be implemented using the Euler tour technique and using the optimal list ranking algorithm to obtain the labels. Hence this step takes $O(\log n)$ time using $O(n)$ operations. Given an array A, A_{odd} and A_{even} (containing the odd and even indexed elements of A respectively) can be obtained in $O(1)$ time using a linear number of operations. The augmented rake operations that modify the sibling labels in Steps (2.1) and (2.2) are done in parallel and hence take $O(1)$ time each. Step (2.3) takes $O(1)$ time. The number of operations required by each iteration is $O(|A|)$, where $|A|$ represents the current size of the array A. Since the size of the array A decreases by half each iteration, the total number of operations required by Step (2) is $O\left(\sum_i (n/2^i)\right) = O(n)$. Hence, this step takes $O(\log n)$ time using $O(n)$ operations. Finally, computing the value of the arithmetic expression from the three-vertex binary tree in Step (3) takes $O(1)$ time and $O(1)$ work. Therefore, arithmetic expression evaluation takes $O(\log n)$ time using $O(n)$ operations on an EREW PRAM.

Note that the *rake* operation used for contracting the tree does not create any new leaves. Hence, the data copy in Step (2.3) of Alg. 4 can be completely avoided by replacing Step (2) with Alg. 4. For further details refer to [15]. This modified algorithm has the same complexity bounds as the original algorithm.

5 SMP Algorithm for AEE

This section begins with the description of the programming environment used for the implementation. The methodology for converting a PRAM algorithm to a practical SMP algorithm is then described. The actual implementation details are then given. This section ends with the description of the cost model used for analyzing the SMP algorithm and a thorough analysis of the algorithm.

SMP Libraries Our practical programming environment for SMPs is based upon the SMP Node Library component of SIMPLE [1], that provides a portable framework for describing SMP algorithms using the single-program multiple-data (SPMD) program style. This framework is a software layer built from POSIX threads that allows the user to use either the already developed SMP primitives or the direct thread primitives. We have been continually developing and improving this library over the past several years and have found it to be portable and efficient on a variety of operating systems (e.g., Sun Solaris, Compaq/Digital UNIX, IBM AIX, SGI IRIX, HP-UX, and Linux). The SMP Node Library contains a number of SMP node algorithms for barrier synchronization, broadcasting the location of a shared buffer, replication of a data buffer, reduction, and memory management for shared-buffer allocation and release. In addition to these functions, we have control mechanisms for contextualization (executing a statement on only a subset of processors), and a *parallel do* that schedules n independent work statements implicitly to p processors as evenly as possible.

The PRAM model assumes the availability of as many processors as needed and a synchronous mode of operation. However, SMPs have a limited number of processors and barriers have to be explicitly used to enforce synchronization. Hence the original PRAM algorithm for AEE can be converted to an SMP algorithm by balancing the work between the available processors and synchronizing the processors between the various steps and substeps of the algorithm. These barriers enforce synchronization and hence the algorithm works correctly on a SMP.

Implementation Details Labeling the leaves is done as follows. Each original tree edge is converted into a pair of directed arcs and weighted as follows.

2. **for** *phase i from* 0 *to* $\lceil \log_2(n-2) \rceil$ **do**
   2.1. Apply the augmented rake operation that modifies the sibling's labels concurrently to all the leaves in locations $2^i, 3\cdot 2^i, 5\cdot 2^i, 7\cdot 2^i, \ldots$ of A that are left children.
   2.2. Apply the augmented rake operation that modifies the sibling's labels concurrently to all the leaves in locations $2^i, 3\cdot 2^i, 5\cdot 2^i, 7\cdot 2^i, \ldots$ of A that are right children.

Algorithm 2: Modified Step (2) of PRAM Algorithm for AEE

A directed arc leaving a leaf vertex is assigned a weight of one; all other arcs are given a weight of zero. The Euler tour of the given tree is then constructed. This Euler tour is represented by a successor array with each node storing the index of its successor. This is followed by list ranking to obtain a consecutive labeling of the leaves. Our implementation uses the SMP list ranking algorithm and implementation developed by Helman and JáJá [10] that performs the following main steps:

1. Finding the head h of the list which is given by $h = (n(n-1)/2 - Z)$ where Z is the sum of successor indices of all the nodes in the list.
2. Partitioning the input list into s sublists by randomly choosing one splitter from each memory block of $n/(s-1)$ nodes, where s is $\Omega(p \log n)$, where p is the number of processors. Corresponding to each of these sublists is a record in an array called *Sublists*. (Our implementation uses $s = 8p$.)
3. Traversing each sublist computing the prefix sum of each node within the sublists. Each node records its sublist index. The input value of a node in the *Sublists* array is the sublist prefix sum of the last node in the previous *Sublists*.
4. The prefix sums of the records in the *Sublists* array are then calculated.
5. Each node adds its current prefix sum value (value of a node within a sublist) and the prefix sum of its corresponding *Sublists* record to get its final prefix sums value. This prefix sum value is the required label of the leaves.

For a detailed description of the above steps refer to [10]. Pointers to the leaf vertices are stored in an array A. This completes Step (1) of AEE algorithm.

Given the array A, the rake operation is applied by updating the necessary pointers in the tree's data structure. The concurrent rake operations in Steps (2.1) and (2.2) are handled similarly. Each processor is responsible for an equal share (at most $\lceil R/p \rceil$) of the R leaves in A_{odd}, and rakes the appropriate leaves in each of these two steps, with a barrier synchronization following each step. Step (2.3) copies the remaining leaves (A_{even}) into another array B. Thus iterations of the *for* loop alternate between using the A and B arrays for the rake operation and store the even leaves in the other array at Step (2.3) of the iteration. Finally, the value of the given expression is computed from the resultant three-vertex binary tree. Hence this implementation, while motivated by the PRAM algorithm, differs substantially in its SMP implementation due to the limited number of processors and need for explicit synchronization. In addition, to achieve high-performance, our SMP implementation must make good use of cache.

Alternatively, we could use Alg. 4 for the second step that eliminates the data copy between arrays A and B. While this modification uses less space (only a single array of leaf pointers), it has the disadvantage that due to the increasing strides of reading the leaves from the A array, more cache misses are likely.

SMP Cost Model We use the SMP complexity model proposed by Helman and JáJá [10] to analyze our shared memory algorithms. In the SMP complexity model, we measure the overall complexity of the algorithm by the

triplet (M_A, M_E, T_C). The term M_A is simply a measure of the number of non-contiguous main memory accesses, where each such access may involve an arbitrary-sized contiguous blocks of data. M_E is the maximum amount of data exchanged by any processor with main memory. T_C is an upper bound on the local computational complexity of any of the processors and is represented by the customary asymptotic notation. M_A, M_E are represented as approximations of the actual values. In practice, it is often possible to focus on either M_A or M_E when examining the cost of algorithms.

Algorithmic Analysis The first step of the algorithm uses the List Ranking algorithm developed by Helman and JáJá [10]. For $n > p^2 \ln n$, we would expect in practice list ranking to take

$$T(n,p) = (M_A(n,p); T_C(n,p)) = \left(\frac{n}{p}, \mathrm{O}\left(\tfrac{n}{p}\right)\right). \tag{2}$$

Tree contraction requires $\mathrm{O}(\log n)$ iterations, where iteration i, for $1 \leq i \leq \log n$ rakes $n/2^i$ leaves concurrently. Since each rake operation requires a constant number of memory accesses of unit size, the tree contraction takes

$$T(n,p) = (M_A(n,p); T_C(n,p)) = \left(\log n + \frac{n}{p}, \mathrm{O}\left(\log n + \tfrac{n}{p}\right)\right). \tag{3}$$

For $n > p \log n$, this simplifies to the same complexity as in Eq. 2.

Thus, for $n > p^2 \lg n$, evaluating arithmetic expressions with $\mathrm{O}(n)$ vertices on a p processor shared-memory machine takes

$$T(n,p) = (M_A(n,p); T_C(n,p)) = \left(\frac{n}{p}, \mathrm{O}\left(\tfrac{n}{p}\right)\right). \tag{4}$$

6 Experimental Results

This section summarizes the experimental results of our implementation. We tested our shared-memory implementation on the Sun HPC 10000, a UMA shared memory parallel machine with 64 UltraSPARC II processors and 64 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. The clock speed of each processor is 400 MHz.

Experimental Data. In order to test the performance of our algorithm, we designed a collection of tree generators that when given an integer d, could generate arbitrarily large regular and irregular rooted, binary trees of $n = 2^d - 1$ vertices. Each generated tree is a strict binary tree, meaning that each vertex has either both subtrees empty or neither subtree empty. Thus, each test tree has n vertices and $(n+1)/2$ leaves. Let the level of a vertex be the number of edges on the path between it and the root. We then generate the following three kinds of strict binary trees:

FULL: These are full, binary trees with 2^l internal vertices at each level l, for $0 \leq l < d-1$, and $(n+1)/2$ leaves at the last level $l = d-1$.

CAT: These "caterpillar" trees are basically long chains of vertices with one leaf attached at each internal vertex, except for two leaves at the bottom (level $(n-1)/2$). Hence, the root is at level 0, and each level l, for $1 \leq l \leq (n-1)/2$, has exactly two vertices.

RAN: Unlike the full and the caterpillar trees, this is an irregular class of binary tree with no specific structure. We randomly create this input using an unrake-like (reverse rake) operation. Each tree is constructed by randomly choosing one of the vertices v in the current tree (except the root) and introducing a new vertex u and a leaf w at the chosen vertex position. More specifically, u becomes the child of $p(v)$ replacing v and w and v become the new children of u. The sub-tree rooted at v remains unaltered. We initially start with a three-vertex tree containing the root and two leaves and build the tree randomly until the tree contains a total of n vertices.

Alg. 4 consists of four main substeps, namely 1) the *Euler tour* computation, 2) *list ranking*, 3) *copying leaves* in the array A, and 4) the *rake* operation that contracts the tree to a three-vertex tree and computes the final value of the arithmetic expression. The running times of our implementation of Alg. 4 for various sizes of **FULL**, **CAT** and **RAN** trees on different numbers of processors are plotted in Fig. 3. The experiments were performed for $d \leq 25$ with the number p of processors equal to 2, 4, 8, 16, and 32, and using 32-bit integers. For $d = 25$ we also give a step-by-step breakdown of the execution time. In all cases, for a particular class of input tree, the implementation performs faster as more processors are employed. The regular trees (**FULL** and **CAT**) represent special cases of expressions and run faster than the irregular input (**RAN**) that represents the most common class of expressions that have an irregular structure. Note that the irregular trees have a longer running time (primarily due to cache misses since the same number of rake operations per iteration are performed), but exhibit a nearly linear relative speedup. This performance matches our analysis (Eq. 4).

Finally, we have implemented a linear sequential time, non-recursive version of expression evaluation to provide sequential timing comparisons. It is important to note that the hidden asymptotic constant associated with the sequential code is very small (close to one), while the parallel code has a much larger hidden constant. Even so, our parallel code is faster than the sequential version when enough processors are used, and in the most general case, the irregular input trees.

In Fig. 4 we compare the performance of the *rake* substep, that is, Step (2) in Algs. 4 and 4 for $d = 25$ on a range of processors. These experimental results confirm that the improved cache hit rate in Alg. 4 overcomes the time spent copying data between the A and B leaf pointer arrays. The improvement, though, is only noticeable when the input trees are regular.

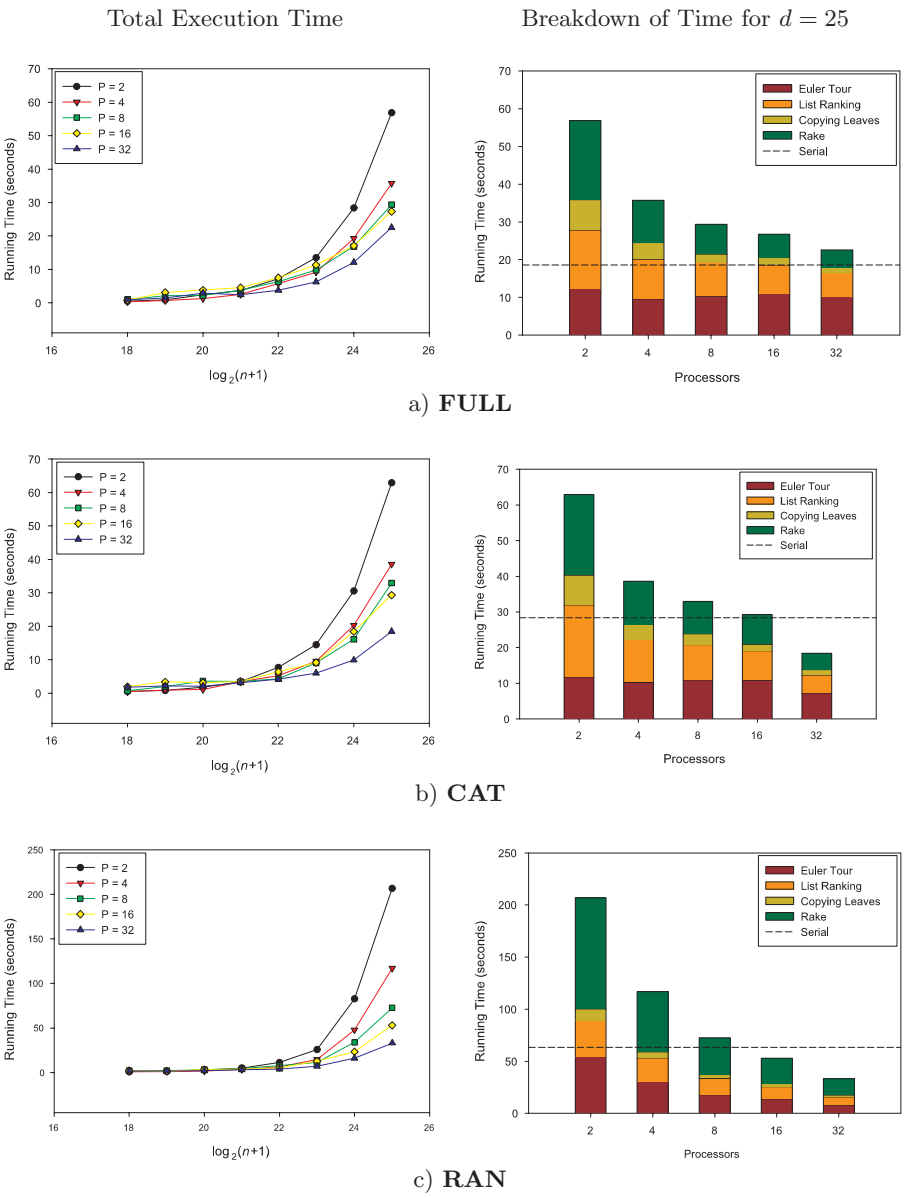
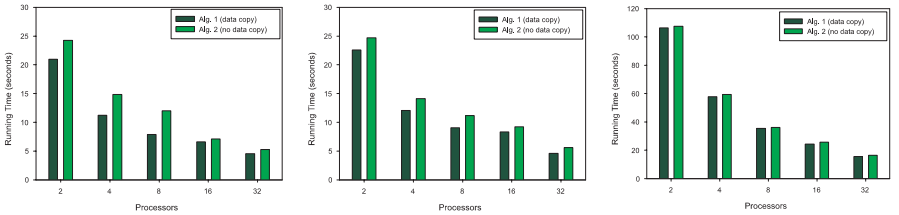


Fig. 3. Expression evaluation execution time for **FULL**, **CAT**, and **RAN** trees. The left-hand graphs plot the total running time taken for varying input sizes and numbers of processors. The corresponding graphs on the right give a step-by-step breakdown of the running time for a fixed input size ($2^{25} - 1$ vertices) and the number p of processors from 2 to 32



a) **FULL** b) **CAT** c) **RAN**

Fig. 4. Comparison of time for *rake* step for **FULL**, **CAT**, and **RAN** trees for a fixed input size ($2^{25} - 1$ vertices) and the number p of processors from 2 to 32

7 Conclusions

In summary, we present optimistic results that for the first time, show that parallel algorithms for expression evaluation using tree contraction techniques run efficiently on parallel symmetric multiprocessors. Our new implementations scale nearly linearly with the problem size and the number of processors, as predicted by our analysis in Eq. 4. These results provide optimistic evidence that complex graph problems that have efficient PRAM solutions, but no known efficient parallel implementations, may have scalable implementations on SMPs.

References

[1] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *J. Parallel & Distributed Comput.*, 58(1):92–108, 1999. 69

[2] D. A. Bader, A. K. Illendula, B. M. E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G. S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, August 2001. Springer-Verlag. 65

[3] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proc. 24th Int'l Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400, Bologna, Italy, 1997. Springer-Verlag. 65

[4] A. Charlesworth. Starfire: extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998. 64

[5] A. Charlesworth. The Sun Fireplane system interconnect. In *Proc. Supercomputing (SC 2001)*, pages 1–14, Denver, CO, November 2001. 64

[6] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988. 65

[7] F. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002. 65

[8] A. M. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. *Information and Computation*, 81:32–45, 1989. 65

[9] B. Grayson, M. Dahlin, and V. Ramachandran. Experimental evaluation of QSM, a simple shared-memory model. In *Proc. 13th Int'l Parallel Processing Symp. and 10th Symp. Parallel and Distributed Processing (IPPS/SPDP)*, pages 1–7, San Juan, Puerto Rico, April 1999. 65

[10] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag. 70, 71

[11] T.-S. Hsu and V. Ramachandran. Efficient massively parallel implementation of some combinatorial algorithms. *Theoretical Computer Science*, 162(2):297–322, 1996. 65

[12] T.-S. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. In *Proc. 9th Int'l Parallel Processing Symp.*, pages 106–112, Santa Barbara, CA, April 1995. 65

[13] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992. 64, 66, 67

[14] J. Keller, C. W. Keßler, and J. L. Träff. *Practical PRAM Programming*. John Wiley & Sons, 2001. 65

[15] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking (extended abstract). Technical report, The Johns Hopkins University, 1987. 65, 66, 68

[16] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th Ann. IEEE Symp. Foundations of Computer Science (FOCS)*, pages 478–489, Portland, OR, October 1985. IEEE Press. 65

[17] OpenMP Architecture Review Board. OpenMP: A proposed industry standard API for shared memory programming. www.openmp.org, October 1997. 64

[18] M. Reid-Miller. List ranking and list scan on the Cray C-90. *J. Comput. Syst. Sci.*, 53(3):344–356, December 1996. 65

[19] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993. 64

[20] J. Sibeyn. Better trade-offs for parallel list ranking. In *Proc. 9th Ann. Symp. Parallel Algorithms and Architectures (SPAA-97)*, pages 221–230, Newport, RI, June 1997. ACM. 65