Memory-Optimal Evaluation of Expression Trees Involving Large Objects[*]

Chi-Chung Lam[*] Daniel Cociorva[**] Gerald Baumgartner[*] P. Sadayappan[*]

[*] Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
{clam, gb, saday}@cis.ohio-state.edu

[**] Department of Physics
The Ohio State University, Columbus, OH 43210
cociorva@pacific.mps.ohio-state.edu

March 27, 2001

Technical Report OSU-CISRC-5/99-TR13, Revised 3/01
Computer and Information Science Department, The Ohio State University

Abstract

The need to evaluate expression trees involving large objects arises in scientific computing applications such as electronic structure calculations. Often, the tree node objects are so large that only a subset of them can fit into memory at a time. This paper addresses the problem of finding an evaluation order of the nodes in a given expression tree that uses the least amount of memory. We present an algorithm that finds an optimal evaluation order in $\Theta(n \log^2 n)$ time for an n-node expression tree and prove its correctness.

1 Introduction

This paper addresses the problem of finding an evaluation order of the nodes in a given expression tree that minimizes memory usage. The expression tree must be evaluated in some bottom-up order, i.e., the evaluation of a node cannot precede the evaluation of any of its children. The nodes of the expression tree are large data objects whose sizes are given. If the total size of the data objects is so large that they cannot all fit into memory at the same time, space for the data objects has to be allocated and deallocated dynamically. Due to the parent-child dependence relation, a data object cannot be deallocated until its parent node data object has been evaluated. The objective is to minimize the maximum memory usage during the evaluation of the entire expression tree.

This problem arises, for example, in optimizing a class of loop calculations implementing multi-dimensional integrals of the products of several large input arrays to compute the electronic properties of semiconductors and metals [3, 9]. One such application is the calculation of electronic properties of MX materials with the inclusion of many-body effects [2]. MX materials are linear chain compounds with alternating transition-metal atoms (M = Ni, Pd, or Pt) and halogen atoms (X = Cl, Br, or I). The following multi-dimensional integral

[*]Supported in part by the National Science Foundation under grant DMR-9520319.

computes the susceptibility in momentum space for the determination of the self-energy of MX compounds in real space.

$$\chi_{\mathbf{G},\mathbf{G}'}(\mathbf{k}, i\tau) = i \sum_{\mathbf{R}''L'',\mathbf{R}'L'} D^{\mathbf{k},\mathbf{G}}_{\mathbf{R}''L'',\mathbf{R}'L'}(i\tau)\ D^{*\mathbf{k},\mathbf{G}'}_{\mathbf{R}''L'',\mathbf{R}'L'}(-i\tau)$$

where

$$D^{\mathbf{k},\mathbf{G}}_{\mathbf{R}''L'',\mathbf{R}'L'}(i\tau) = \frac{1}{\sqrt{\Omega}} \int_{\Omega} d\mathbf{r}\ e^{-i(\mathbf{k}+\mathbf{G})\mathbf{r}} \sum_{\mathbf{R}'''L'''} \theta_{\mathbf{R}'L',\mathbf{R}'''L'''}(\mathbf{r})\ G_{\mathbf{R}''L'',\mathbf{R}'''L'''}(i\tau)$$

and

$$\theta_{\mathbf{R}'L',\mathbf{R}'''L'''}(\mathbf{r}) = \Psi_{\mathbf{R}'L'}(\mathbf{r} - \mathbf{R}')\ \Psi^{*}_{\mathbf{R}'''L'''}(\mathbf{r} - \mathbf{R}''')$$

In the above equations, Ψ is the localized basis function, G is the orbital projected Green function for electrons and holes, and D is an intermediate array computed using a fast Fourier transform (FFT). The interpretation of the variables and their ranges are given in Table 1. The computation can be expressed in a canonical form as the following multi-dimensional summation of the product of several arrays, where Ψ is written as a two-dimensional array Y.

$$\begin{aligned}\sum_{r,r1,RL,RL1,RL2,RL3} \quad & Y[r,RL] \times Y[r,RL2] \times Y[r1,RL3] \times Y[r1,RL1] \\ & \times G[RL1,RL,t] \times G[RL2,RL3,t] \\ & \times exp[k,r] \times exp[G,r] \times exp[k,r1] \times exp[G1,r1]\end{aligned}$$

When expressed in this form, as a multi-dimensional summation of the primary inputs to the computation, no intermediate quantities such as D are explicitly computed. Although it requires less memory than the previous form, it is not computationally attractive since the number of arithmetic operations is enormous (multiplying the extents of the various summation indices gives $O(10^{22})$ operations). The number of operations can be considerably reduced by applying algebraic properties to factor out terms that are independent of some of the summation indices, thereby computing some intermediate results that can be stored and reused instead of being recomputed many times. There are many different ways of applying algebraic laws of distributivity and associativity, resulting in different alternative computational structures. Using simple examples, we briefly touch upon some of the optimization issues that arise, and then proceed to discuss in detail the problem of memory-optimal evaluation of expression trees.

In previous work [4, 5], we have addressed the problem of minimizing the number of arithmetic operations by applying the algebraic laws of commutativity, associativity, and distributivity. Consider, for example, the multi-dimensional integral shown in Figure 1(a). If implemented directly as expressed (i.e., as a single set of perfectly-nested loops), the computation would require $2 \times N_i \times N_j \times N_k \times N_l$ arithmetic operations to compute. However, assuming associative reordering of the operations and use of the distributive law of multiplication over addition is satisfactory for the floating-point computations, the above computation can be rewritten in various ways. One equivalent form that only requires $2 \times N_j \times N_k \times N_l + 2 \times N_j \times N_k + N_i \times N_j$ operations is given in Figure 1(b). It expresses the sequence of steps in computing the multi-dimensional integral as a sequence of formulae. Each formula computes some intermediate result and the last formula gives the final result. A sequence of formulae can also be represented as an expression tree in which the leaf nodes are input arrays, the internal nodes are intermediate results, and the root node is the final integral. For instance, Figure 1(c) shows the expression tree corresponding to the example formula sequence. This problem has been proved to be NP-complete and a pruning search algorithm was proposed.

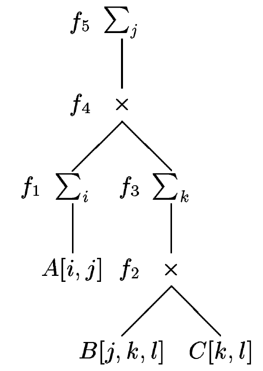
Once the operation-minimal form is determined, the next step is to implement it as some loop structure. A straightforward way is to generate a sequence of perfectly nested loops, each evaluating an intermediate result. However, in practice, the input arrays and the intermediate arrays are often so large that they cannot all fit into available memory. The minimization of memory usage is thus desirable. By fusing loops it is possible to reduce the dimensionality of some of the intermediate arrays [6, 7]. Furthermore, it is not necessary to keep all arrays allocated for the duration of the entire computation. Space allocated for an intermediate array can be deallocated as soon as the operation using the array has been performed. When allocating and deallocating arrays dynamically, the evaluation order affects the maximum memory requirement. In

$$W[k] \quad = \quad \textstyle\sum_{(i,j,l)} A[i,j] \times B[j,k,l] \times C[k,l]$$

(a) A multi-dimensional integral

$$\begin{array}{lcl} f_1[j] & = & \sum_i A[i,j] \\ f_2[j,k,l] & = & B[j,k,l] \times C[k,l] \\ f_3[j,k] & = & \sum_l f_2[j,k,l] \\ f_4[j,k] & = & f_1[j] \times f_3[j,k] \\ W[k] = f_5[k] & = & \sum_j f_4[j,k] \end{array}$$

(b) A formula sequence for computing (a)



(c) An expression tree representation of (b)

Figure 1: An example multi-dimensional integral and two representations of a computation.

this paper, we focus on the problem of finding an evaluation order of the nodes in a given expression tree that minimizes dynamic memory usage. A solution to this problem would allow the automatic generation of efficient code for evaluating expression trees, e.g., for computing the electronic properties of MX materials. We believe that the solution we develop here may have applicability to other areas such as database query optimization and data mining.

As an example of the memory usage optimization problem, consider the expression tree shown in Fig. 2. The size of each data object is shown alongside the corresponding node label. Before evaluating a data object, space for it must be allocated. This space can be deallocated only after the evaluation of its parent is complete. There are many allowable evaluation orders of the nodes. One of them is the post-order traversal $\langle A, B, C, D, E, F, G, H, I \rangle$ of the expression tree. It has a maximum memory usage of 45 units. This occurs during the evaluation of H, when F, G, and H are in memory. Other evaluation orders may use more memory or less memory. Finding the optimal order $\langle C, D, G, H, A, B, E, F, I \rangle$, which uses 39 units of memory, is not trivial.

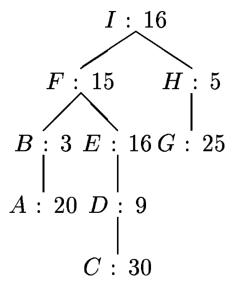


Figure 2: An example expression tree

A simpler problem related to the memory usage minimization problem is the register allocation problem in which the sizes of all nodes are unity. It has been addressed in [8, 10] and can be solved in $\Omega(n)$ time, where n is the number of nodes in the expression tree. But if the expression tree is replaced by a directed acyclic graph (in which all nodes are still of unit size), the problem becomes NP-complete [11]. The algorithm in [10] for expression trees of unit-sized nodes does not extend directly to expression trees having nodes of different sizes. Appel and Supowit [1] generalized the register allocation problem to higher degree expression trees of arbitrarily-sized nodes. However, the problem they addressed is slightly different from ours in that, in their problem, space for a node is not allocated during its evaluation. Also, they restricted their attention to solutions that evaluate subtrees contiguously, which is sub-optimal in some cases. We are not aware of any existing algorithm to the memory usage optimization problem considered in this paper.

The rest of this paper is organized as follows. In Section 2, we formally define the memory usage optimization problem and make some observations about it. Section 3 presents an efficient algorithm for finding the optimal evaluation order for an expression tree. In Section 4, we prove that the algorithm finds a solution in $\Theta(n \log^2 n)$ time for an n-node expression tree. The correctness of the algorithm is proved in Section 5. Section 6 provides conclusions.

2 Problem Statement

We address the problem of optimizing the memory usage in the evaluation of a given expression tree whose nodes correspond to large data objects of various sizes. Each data object depends on all its children (if any), and thus can be evaluated only after all its children have been evaluated. We assume that the evaluation of each node in the expression tree is atomic. Space for each data object is dynamically allocated/deallocated in its entirety. Leaf node objects are created or read in as needed; internal node objects must be allocated before their evaluation begins; and each object must remain in memory until the evaluation of its parent is completed. The goal is to find an evaluation order of the nodes that uses the least amount of memory. Since an evaluation order is also a traversal of the nodes, we will use these two terms interchangeably.

We define the problem formally as follows:

Given a tree T and a size $v.size$ for each node $v \in T$, find a computation of T that uses the least memory, i.e., an ordering $P = \langle v_1, v_2, \ldots, v_n \rangle$ of the nodes in T, where n is the number of nodes in T, such that

1. for all v_i, v_j, if v_i is the parent of v_j, then $i > j$; and
2. $\max_{v_i \in P}\{\text{himem}(v_i, P)\}$ is minimized, where

$$\begin{aligned}\text{himem}(v_i, P) &= \text{lomem}(v_{i-1}, P) + v_i.size \\ \text{lomem}(v_i, P) &= \begin{cases} \text{himem}(v_i, P) - \sum_{\{\text{child } v_j \text{ of } v_i\}} v_j.size & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases}\end{aligned}$$

Here, $\text{himem}(v_i, P)$ is the memory usage during the evaluation of v_i in the traversal P, and $\text{lomem}(v_i, P)$ is the memory usage upon completion of the same evaluation. These definitions reflect that we need to allocate space for v_i before its evaluation, and that after evaluation of v_i, the space allocated to all its children may be released. For instance, consider the post-order traversal $P = \langle A, B, C, D, E, F, G, H, I \rangle$ of the expression tree shown in Fig. 2. During and after the evaluation of A, A is in memory. So, $\text{himem}(A, P) = \text{lomem}(A, P) = A.size = 20$. For evaluating B, we need to allocate space for B, thus $\text{himem}(B, P) = \text{lomem}(A, P) + B.size = 23$. After B is obtained, A can be deallocated, giving $\text{lomem}(B, P) = \text{himem}(B, P) - A.size = 3$. The memory usage for the rest of the nodes is determined similarly and shown in Fig. 3.

The post-order traversal of the given expression tree, however, is not optimal in memory usage. For this example, none of the traversals that visit all nodes in one subtree before visiting another subtree is optimal. There are four such traversals: $\langle A, B, C, D, E, F, G, H, I \rangle$, $\langle C, D, E, A, B, F, G, H, I \rangle$, $\langle G, H, A, B, C, D, E, F, I \rangle$, and $\langle G, H, C, D, E, A, B, F, I \rangle$. If we follow the traditional wisdom of visiting the subtree with the higher memory usage first, as in the Sethi-Ullman algorithm [10], we obtain the best of these four traversals, which is $\langle G, H, C, D, E, A, B, F, I \rangle$. Its overall memory usage is 44 units, as shown in Fig. 4(a), and is not optimal.

Variable		Range
RL	Orbital	10^3
r	Discretized points in real space	10^5
τ	Time step	10^2
k	K-point in a irreducible Brilloin zone	10
G	Reciprocal lattice vector	10^3

Table 1: Variables in an example physics computation.

Node	himem	deallocate	lomem
A	$0 + 20 = 20$	-	$20 - 0 = 20$
B	$20 + 3 = 23$	A	$23 - 20 = 3$
C	$3 + 30 = 33$	-	$33 - 0 = 33$
D	$33 + 9 = 42$	C	$42 - 30 = 12$
E	$12 + 16 = 28$	D	$28 - 9 = 19$
F	$19 + 15 = 34$	B, E	$34 - 3 - 16 = 15$
G	$15 + 25 = 40$	-	$40 - 0 = 40$
H	$40 + 5 = 45$	G	$45 - 25 = 20$
I	$20 + 16 = 36$	F, H	$36 - 15 - 5 = 16$
max	45		

Figure 3: Memory usage of a post-order traversal of the expression tree in Fig. 2

Node	himem	lomem
G	25	25
H	30	5
C	35	35
D	44	14
E	30	21
A	41	41
B	44	24
F	39	20
I	36	16
max	44	

(a) A better traversal

Node	himem	lomem
C	30	30
D	39	9
G	34	34
H	39	14
A	34	34
B	37	17
E	33	24
F	39	20
I	36	16
max	39	

(b) The optimal traversal

Figure 4: Memory usage of two different traversals of the expression tree in Fig. 2

The optimal traversal is $\langle C, D, G, H, A, B, E, F, I\rangle$ and uses 39 units of memory (see Fig. 4(b)). Notice that it 'jumps' back and forth between the subtrees. Therefore, any algorithm that only considers traversals that visit subtrees contiguously may not produce an optimal solution.

The memory usage optimization problem has an interesting property: an expression tree or a subtree may have more than one optimal traversal. For example, for the subtree rooted at F, the traversals $\langle C, D, E, A, B, F\rangle$ and $\langle C, D, A, B, E, F\rangle$ both use the least memory space of 39 units. One might attempt to take two optimal subtree traversals, one from each child of a node X, merge them together optimally, and then append X to form a traversal for X. But, this resulting traversal may not be optimal for X. Continuing with the above example, if we merge together $\langle C, D, E, A, B, F\rangle$ and $\langle G, H\rangle$ (which are optimal for the subtrees rooted at F and H, respectively) and then append I, the best we can get is a sub-optimal traversal $\langle G, H, C, D, E, A, B, F, I\rangle$ that uses 44 units of memory (see Fig. 4(a)). However, the other optimal traversal $\langle C, D, A, B, E, F\rangle$ for the subtree rooted at F can be merged with $\langle G, H\rangle$ to form $\langle C, D, G, H, A, B, E, F, I\rangle$ (with I appended), which is an optimal traversal of the entire expression tree. Thus, locally optimal traversals may not be globally optimal. In the next section, we present an efficient algorithm that finds traversals which are not only locally optimal but also globally optimal.

3 An Efficient Algorithm

We now present an efficient divide-and-conquer algorithm that, given an expression tree whose nodes are large data objects, finds an evaluation order of the tree that minimizes the memory usage. For each node in the expression tree, it computes an optimal traversal for the subtree rooted at that node. The optimal subtree traversal that it computes has a special property: it is not only locally optimal for the subtree, but also globally optimal in the sense that it can be merged together with globally optimal traversals for other subtrees to form an optimal traversal for a larger tree that is also globally optimal. As we have seen in Section 2, not all locally optimal traversals for a subtree can be used to form an optimal traversal for a larger tree.

The algorithm stores a traversal not as an ordered list of nodes, but as an ordered list of *indivisible units* or *elements*. Each element contains an ordered list of nodes with the property that there necessarily exists some globally optimal traversal of the entire tree wherein this sequence appears undivided. Therefore, as we show later, inserting any node in between the nodes of an element does not lower the total memory usage. An element initially contains a single node. But as the algorithm goes up the tree merging traversals together and appending new nodes to them, elements may be appended together to form new elements containing a larger number of nodes. Moreover, the order of indivisible units in a traversal stays invariant, i.e., the indivisible units must appear in the same order in some optimal traversal of the entire expression tree. This means that indivisible units can be treated as a whole and we only need to consider the relative order of indivisible units from different subtrees.

Each element (or indivisible unit) in a traversal is a (*nodelist*, *hi*, *lo*) triple, where *nodelist* is an ordered list of nodes, *hi* is the maximum memory usage during the evaluation of the nodes in *nodelist*, and *lo* is the memory usage after those nodes are evaluated. Using the terminology from Section 2, *hi* is the highest himem among the nodes in *nodelist*, and *lo* is the lomem of the last node in *nodelist*. The algorithm always maintains the elements of a traversal in decreasing *hi* and increasing *lo* order, which implies in order of decreasing *hi-lo* difference. In Section 5, we prove that arranging the indivisible units in this order minimizes memory usage.

Before formally describing the algorithm, we illustrate how it works with an example. Consider the expression tree shown in Fig. 2. We visit the nodes in a bottom-up order. Since A has no children, the optimal traversal for the subtree rooted at A, denoted by $A.seq$, is $\langle (A, 20, 20)\rangle$, meaning that 20 units of memory are needed during the evaluation of A and immediately afterwards. To form $B.seq$, we take $A.seq$, append a new element $(B, 3+20, 3)$ (the *hi* of which is adjusted by the *lo* of the preceding element), and get $\langle (A, 20, 20), (B, 23, 3)\rangle$. Whenever two adjacent elements are not in decreasing *hi* and increasing *lo* order, we combine them into one element by concatenating the *nodelists* and taking the highest *hi* and the second *lo*. Thus, $B.seq$ becomes $\langle (AB, 23, 3)\rangle$. (For clarity, we write *nodelists* in a sequence as strings). Here, A and B form an indivisible unit, implying that B must follow A in some optimal traversal of the entire expression tree. The maximum memory usage during the evaluation of this indivisible unit is 23 and the result of the

Node v	Subtree traversals merged in decreasing *hi-lo* order	Optimal traversal $v.seq$
A	$\langle(A, 20, 20)\rangle$	$\langle(A, 20, 20)\rangle$
B	$\langle(A, 20, 20), (B, 23, 3)\rangle$	$\langle(AB, 23, 3)\rangle$
C	$\langle(C, 30, 30)\rangle$	$\langle(C, 30, 30)\rangle$
D	$\langle(C, 30, 30), (D, 39, 9)\rangle$	$\langle(CD, 39, 9)\rangle$
E	$\langle(CD, 39, 9), (E, 25, 16)\rangle$	$\langle(CD, 39, 9), (E, 25, 16)\rangle$
F	$\langle(CD, 39, 9), (AB, 32, 2), (E, 28, 19), (F, 34, 15)\rangle$	$\langle(CD, 39, 9), (ABEF, 34, 15)\rangle$
G	$\langle(G, 25, 25)\rangle$	$\langle(G, 25, 25)\rangle$
H	$\langle(G, 25, 25), (H, 30, 5)\rangle$	$\langle(GH, 30, 5)\rangle$
I	$\langle(CD, 39, 9), (GH, 39, 14), (ABEF, 39, 20), (I, 36, 16)\rangle$	$\langle(CDGHABEFI, 39, 16)\rangle$

Figure 5: Optimal traversals for the subtrees in the expression tree in Fig. 2

evaluation occupies 3 units of memory. Similarly, we get $E.seq = \langle(CD, 39, 9), (E, 25, 16)\rangle$. Note that these two adjacent elements cannot be combined because they are already in decreasing *hi* and increasing *lo* order. For node F, which has two children B and E, we merge $B.seq$ and $E.seq$ by the order of decreasing *hi-lo* difference. The merged elements are in the order $(CD, 39, 9)$, $(AB, 23+9, 3+9)$, and finally $(E, 25+3, 16+3)$ with the *hi* and *lo* values adjusted by the *lo* value of the last merged element from the other subtree. They are the three elements in *F.seq* after the merge as no elements are combined so far. Then, we append to *F.seq* the new element $(F, 15+19, 15)$ for the root of the subtree. The new element is combined with the last two elements in *F.seq* to ensure that the elements are in decreasing *hi-lo* difference. Hence, *F.seq* becomes $\langle(CD, 39, 9), (ABEF, 34, 15)\rangle$, a sequence of only two indivisible units. The optimal traversals for the other nodes are computed in the same way and are shown in Fig. 5. At the end, the algorithm returns the optimal traversal $\langle C, D, G, H, A, B, E, F, I\rangle$ for the entire expression tree (see Fig. 4(b)).

Fig. 6 shows the algorithm. The input to the algorithm (the **MinMemTraversal** procedure) is an expression tree T, in which each node v has a field $v.size$ denoting the size of its data object. The procedure performs a bottom-up traversal of the tree and, for each node v, computes an optimal traversal $v.seq$ for the subtree rooted at v. The optimal traversal $v.seq$ is obtained by optimally merging together the optimal traversals $u.seq$ from each child u of v, and then appending v. At the end, the procedure returns a concatenation of all the *nodelists* in $T.root.seq$ as the optimal traversal for the given expression tree. The memory usage of the optimal traversal is $T.root.seq[1].hi$.

The **MergeSeq** procedure optimally merges two given traversals $S1$ and $S2$ and returns the merged result S. $S1$ and $S2$ are subtree traversals of two children nodes of the same parent. The optimal merge is performed in a fashion similar to merge-sort. Elements from $S1$ and $S2$ are scanned sequentially and appended into S in the order of decreasing *hi-lo* difference. This order guarantees that the indivisible units are arranged to minimize memory usage. Since $S1$ and $S2$ are formed independently, the *hi-lo* values in the elements from $S1$ and $S2$ must be adjusted before they can be appended to S. The amount of adjustment for an element from $S1$ ($S2$) equals the *lo* value of the last merged element from $S2$ ($S1$), which is kept in variable *base1* (*base2*).

The **AppendSeq** procedure appends the new element specified by the triple (*nodelist*, *hi*, *lo*) to the given traversal S. Before the new element E is appended to S, it is combined with elements at the end of S whose *hi* is not higher than $E.hi$ or whose *lo* is not lower than $E.lo$. The combined element has the concatenated *nodelist* and the highest *hi* but the original $E.lo$.

This algorithm has the property that the traversal it finds for a subtree T' is not only optimal for T' but must also appear as a subsequence in some optimal traversal for any larger tree that contains T' as a subtree. For example, *E.seq* is a subsequence in *F.seq*, which is in turn a subsequence in *I.seq* (see Fig. 5).

```
MinMemTraversal ($T$):
    foreach node $v$ in some bottom-up traversal of $T$
      $v.seq = \langle\rangle$
      foreach child $u$ of $v$
        $v.seq = \textbf{MergeSeq}(v.seq, u.seq)$
      if $|v.seq| > 0$ then      // $|x|$ is the length of $x$
        $base = v.seq[|v.seq|].lo$
      else
        $base = 0$
      AppendSeq ($v.seq, \langle v\rangle, v.size + base, v.size$)
    $nodelist = \langle\rangle$
    for $i = 1$ to $|T.root.seq|$
      $nodelist = nodelist + T.root.seq[i].nodelist$      // + is concatenation
    return $nodelist$      // memory usage is $T.root.seq[1].hi$

MergeSeq ($S1, S2$):
    $S = \langle\rangle$
    $i = j = 1$
    $base1 = base2 = 0$
    while $i \leq |S1|$ or $j \leq |S2|$
      if $j > |S2|$ or ($i \leq |S1|$ and $S1[i].hi - S1[i].lo > S2[j].hi - S2[j].lo$) then
        AppendSeq ($S, S1[i].nodelist, S1[i].hi + base1, S1[i].lo + base1$)
        $base2 = S1[i].lo$
        $i$++
      else
        AppendSeq ($S, S2[j].nodelist, S2[j].hi + base2, S2[j].lo + base2$)
        $base1 = S1[j].lo$
        $j$++
    end while
    return $S$

AppendSeq ($S$, $nodelist$, $hi$, $lo$):
    $E = (nodelist, hi, lo)$      // new element to append to $S$
    $i = |S|$
    while $i \geq 1$ and ($E.hi \geq S[i].hi$ or $E.lo \leq S[i].lo$)
      // combine $S[i]$ with $E$
      $E = (S[i].nodelist + E.nodelist, \max(S[i].hi, E.hi), E.lo)$
      remove $S[i]$ from $S$
      $i$−−
    end while
    $S = S + E$      // $|S|$ is now $i + 1$
```

Figure 6: Procedure for finding an memory-optimal traversal of an expression tree

4 Complexity of the Algorithm

The time complexity of our algorithm is $\Theta(n \log^2 n)$ for an n-node expression tree. We represent a sequence of indivisible units as a red-black tree with the indivisible units at the leaves. The tree is sorted by decreasing *hi-lo* difference. In addition, the leaves are linked in sorted order in a doubly-linked list, and a count of the number of indivisible units in the sequence is maintained.

The cost of constructing the final evaluation order consists of the cost for building sequences of indivisible units and the cost for combining indivisible units into larger indivisible units. For finding an upper bound on the cost of the algorithm, the worst-case cost for building sequences can be analyzed separately from the worst-case cost for combining indivisible units.

The sequence for a leaf node of the expression tree can be constructed in constant time. For a unary interior node, we simply append the node to the sequence of its subtree, which costs $O(\log n)$ time. For an m-ary interior node, we merge the sequences of the subtrees by inserting the nodes from the smaller sequences into the largest sequence. Inserting a node into a sequence represented as a red-black tree costs $O(\log n)$ time. Since we always insert the nodes of the smaller sequences into the largest one, every time a given node of the expression tree gets inserted into a sequence the size of the sequence containing this node at least doubles. Each node, therefore, can be inserted into a sequence at most $O(\log n)$ times, with each insertion costing $O(\log n)$ time. The cost for building the traversal for the entire expression tree is, therefore, $O(n \log^2 n)$.

Two individual indivisible units can be combined in constant time. When combining two adjacent indivisible units within a sequence, one of them must be deleted from the sequence and the red-black tree must be rebalanced, which costs $O(\log n)$ time. Since there can be at most $n-1$ of these combine operations, the total cost is $O(n \log n)$. The cost of the whole algorithm is, therefore, dominated by the cost for building sequences, which is $O(n \log^2 n)$.

Combining indivisible units into larger ones reduces the number of elements in the sequences and, therefore, the time required for merging and combining sequences. In the best case, indivisible units are always combined such that each sequence contains a single element. In this case, the algorithm only takes linear time.

In the worst case, a degenerate expression tree consists of small nodes and large nodes such that every small node has as its only child a large node. A pair of such nodes will form an indivisible unit with the *hi* being the size of the large node and the *lo* being the size of the small node. Such a tree can be constructed such that these indivisible units will not further combine into larger indivisible units. In this case, the algorithm will result in a sequence of $n/2$ indivisible units containing two nodes each. If such a degenerate expression tree is also unbalanced, the algorithm requires $\Omega(n \log^2 n)$ time for computing the optimal traversal.

5 Correctness of the Algorithm

We now show the correctness of the algorithm. The proof proceeds as follows. Lemma 1 characterizes the indivisible units in a subtree traversal by providing some invariants. Lemma 2 shows that, once formed, each indivisible unit can be considered as a whole. Lemma 3 deals with the optimal ordering of indivisible units from different subtrees. Finally, using the three lemmas we prove the correctness of the algorithm by arguing that any traversal of an expression tree can be transformed in a series of steps into the optimal traversal found by the algorithm without increasing memory usage.

The first lemma establishes some important invariants about the indivisible units in an ordered list $v.seq$ that represents a traversal. The *hi* value of an indivisible unit is the highest himem of the nodes in the indivisible unit. The *lo* value of an indivisible unit is the lomem of the last node in the indivisible unit. Given a sequence of nodes, an indivisible unit extends to the last node with the lowest lomem following the last node with the highest himem. In addition, the indivisible units in a sequence are in decreasing *hi* and increasing *lo* order.

Lemma 5.1 Let v be any node in an expression tree, $S = v.seq$, and P be the traversal represented by S of the subtree rooted at v, i.e., $P = S[1].nodelist + \cdots + S[|S|].nodelist$. The algorithm maintains the following invariants:

For all $1 \leq i \leq |S|$, let $S[i].\mathit{nodelist} = \langle v_1, v_2, \ldots, v_n \rangle$ and v_m be the last node in $S[i].\mathit{nodelist}$ that has the maximum himem value, i.e., for all $k < m$, $\mathrm{himem}(v_k, P) \leq \mathrm{himem}(v_m, P)$ and for all $k > m$, $\mathrm{himem}(v_k, P) < \mathrm{himem}(v_m, P)$. Then, we have,

1. $S[i].hi = \mathrm{himem}(v_m, P)$,
2. $S[i].lo = \mathrm{lomem}(v_n, P)$,
3. for all $m \leq k \leq n$, $\mathrm{lomem}(v_k, P) \geq \mathrm{lomem}(v_n, P)$,
4. for all $1 \leq j < i$,
   (a) for all $1 \leq k \leq n$, $S[j].hi > \mathrm{himem}(v_k, P)$,
   (b) for all $1 \leq k \leq n$, $S[j].lo < \mathrm{lomem}(v_k, P)$,
   (c) $S[j].hi > S[i].hi$, and
   (d) $S[j].lo < S[i].lo$.

Proof

The above invariants are true by construction. □

The second lemma asserts the 'indivisibility' of an indivisible unit by showing that unrelated nodes inserted in between the nodes of an indivisible unit can always be moved to the beginning or the end of the indivisible unit without increasing memory usage. Thus, once an indivisible unit is formed, we do not need to consider breaking it up later. This lemma allows us to treat each traversal as a sequence of indivisible units (each containing one or more nodes) instead of a list of the individual nodes.

Lemma 5.2 Let v be a node in an expression tree T, $S = v.\mathit{seq}$, and P be a traversal of T in which the nodes from $S[i].\mathit{nodelist}$ appear in the same order as they are in $S[i].\mathit{nodelist}$, but not contiguously. Then, any nodes that are in between the nodes in $S[i].\mathit{nodelist}$ can always be moved to the beginning or the end of $S[i].\mathit{nodelist}$ without increasing memory usage, provided that none of the nodes that are in between the nodes in $S[i].\mathit{nodelist}$ are ancestors or descendants of any nodes in $S[i].\mathit{nodelist}$.

Proof

Let $S[i].\mathit{nodelist} = \langle v_1, \ldots, v_n \rangle$, v_0 be the node before v_1 in S, v_m be the node in $S[i].\mathit{nodelist}$ such that for all $k < m$, $\mathrm{himem}(v_k, P) \leq \mathrm{himem}(v_m, P)$ and for all $k > m$, $\mathrm{himem}(v_m, S) > \mathrm{himem}(v_k, S)$. Let $v'_1, \ldots, v'_b$ be the 'foreign' nodes, i.e., the nodes that are in between the nodes in $S[i].\mathit{nodelist}$ in P, with $v'_1, \ldots, v'_a$ (not necessarily contiguously) before v_m and $v'_{a+1}, \ldots, v'_b$ (not necessarily contiguously) after v_m in P. Let Q be the traversal obtained from P by removing the nodes in $S[i].\mathit{nodelist}$. We construct another traversal P' of T from P by moving $v'_1, \ldots, v'_a$ to the beginning of $S[i].\mathit{nodelist}$ and $v'_{a+1}, \ldots, v'_b$ to the end of $S[i].\mathit{nodelist}$. In other words, we replace $\langle v_1, \ldots, v'_1, \ldots, v'_a, \ldots, v_m, \ldots, v'_{a+1}, \ldots, v'_b, \ldots, v_n \rangle$ in P with $\langle v'_1, \ldots, v'_a, v_1, \ldots, v_m, \ldots, v_n, v'_{a+1}, \ldots, v'_b, \rangle$ to form P'.

The traversals P and P' differ in memory usage only at the nodes $\{v_1, \ldots, v_n, v'_1, \ldots, v'_b\}$. P' does not use more memory than P because:

1. The memory usage for P' at v_m is the same as the memory usage for P at v_m since $\mathrm{himem}(v_m, P') = \mathrm{himem}(v_m, S) + \mathrm{lomem}(v'_a, Q) = \mathrm{himem}(v_m, P)$.

2. For all $1 \leq k \leq n$, the memory usage for P' at v_k is no higher than the memory usage for P at v_k, since $\mathrm{himem}(v_k, S) \leq \mathrm{himem}(v_m, S)$ implies that $\mathrm{himem}(v_k, P') = \mathrm{himem}(v_k, S) + \mathrm{lomem}(v'_a, Q) \leq \mathrm{himem}(v_m, S) + \mathrm{lomem}(v'_a, Q) = \mathrm{himem}(v_m, P)$.

3. For all $1 \leq j \leq a$, the memory usage for P' at v'_j is no higher than the memory usage for P at v'_j, since for all $1 \leq k \leq m$, $\mathrm{lomem}(v_0, S) < \mathrm{lomem}(v_k, S)$ (by invariant 4(b) in Lemma 1) implies $\mathrm{himem}(v'_j, P') = \mathrm{himem}(v'_j, Q) + \mathrm{lomem}(v_0, S) \leq \mathrm{himem}(v'_j, P)$.

4. For all $a < j \leq b$, the memory usage for P' at v'_j is no higher than the memory usage for P at v'_j, since for all $m \leq k \leq n$, $\mathrm{lomem}(v_k, S) \geq \mathrm{lomem}(v_n, S)$ (by invariant 3 in Lemma 1) implies $\mathrm{himem}(v'_j, P') = \mathrm{himem}(v'_j, Q) + \mathrm{lomem}(v_n, S) \leq \mathrm{himem}(v'_j, P)$.

element	himem	lomem
⋮	⋮	⋮
$S'[j]$	$H'_j + L_{i-1}$	$L'_j + L_{i-1}$
$S[i]$	$H_i + L'_j$	$L_i + L'_j$
⋮	⋮	⋮

(a) Sequence M

element	himem	lomem
⋮	⋮	⋮
$S[i]$	$H_i + L'_{j-1}$	$L_i + L'_{j-1}$
$S'[j]$	$H'_j + L_i$	$L'_j + L_i$
⋮	⋮	⋮

(b) Sequence M'

Figure 7: Memory usage comparison of two traversals in Lemma 5.3

Since the memory usage of any node in $v_1, \ldots, v_n$ after moving the foreign nodes cannot exceed that of v_m, which remains unchanged, and the memory usage of the foreign nodes does increase as a result of moving them, the overall maximum memory usage cannot increase. $\square$

The next lemma deals with the ordering of indivisible units. It shows that arranging indivisible units from different subtrees in the order of decreasing *hi-lo* difference minimizes memory usage, since two indivisible units that are not in that order can be interchanged in the merged traversal without increasing memory usage.

Lemma 5.3 Let v and v' be two nodes in an expression tree that are siblings of each other, $S = v.seq$, and $S' = v'.seq$. Then, among all possible merges of S and S', the merge that arranges the elements from S and S' in the order of decreasing *hi-lo* difference uses the least memory.

Proof

Let M be a merge of S and S' that is not in the order of decreasing *hi-lo* difference. Then there exists an adjacent pair of elements, one from each of S and S', that are not in that order. Without loss of generality, we assume the first element is $S'[j]$ from S' and the second one is $S[i]$ from S. Consider the merge M' obtained from M by interchanging $S'[j]$ and $S[i]$. To simplify the notation, let $H_r = S[r].hi$, $L_r = S[r].lo$, $H'_r = S'[r].hi$, and $L'_r = S'[r].lo$. The memory usage of M and M' differs only at $S'[j]$ and $S[i]$ and is compared in Fig. 7.

The memory usage of M at the two elements is $\max(H'_j + L_{i-1}, H_i + L'_j)$ while the memory usage of M' at the same two elements is $\max(H'_j + L_i, H_i + L'_{j-1})$. Since the two elements are out of order, the *hi-lo* difference of $S'[j]$ must be less than that of $S[i]$, i.e., $H'_j - L'_j < H_i - L_i$. This implies $H_i + L'_j > H'_j + L_i$. Invariant 4 in Lemma 1 gives us $L'_j > L'_{j-1}$, which implies $H_i + L'_j > H_i + L'_{j-1}$. Thus, $\max(H'_j + L_{i-1}, H_i + L'_j) \geq H_i + L'_j > \max(H'_j + L_i, H_i + L'_{j-1})$. Therefore, M' cannot use more memory than M. By switching all adjacent pairs in M that are out of order until no such pair exists, we get an optimal order without increasing memory usage. $\square$

Theorem 5.4 Given an expression tree, the algorithm presented in Section 3 computes a traversal that uses the least memory.

Proof

We prove the correctness of the algorithm by describing a procedure that transforms any given traversal to the traversal found by the algorithm without increase in memory usage in any transformation step. Given a traversal P for an expression tree T, we visit the nodes in T in a bottom-up manner and, for each non-leaf node v in T, we perform the following steps:

1. Let T' be the subtree rooted at v and P' be the minimal substring of P that contains all the nodes from $T' - \{v\}$. In the following steps, we will rearrange the nodes in P' such that the nodes that form an indivisible unit in $v.seq$ are contiguous and the indivisible units are in the same order as they are in $v.seq$.

2. First, we sort the components of the indivisible units in $v.seq$ so that they are in the same order as in $v.seq$. The sorting process involves rearranging two kinds of units. The first kind of units

are the indivisible units in $u.seq$ for each child u of v. The second kind of units are the contiguous sequences of nodes in P' which are from $T - T'$. For this sorting step, we temporarily treat each such maximal contiguous sequence of nodes as a unit. For each unit E of the second kind, we take $E.hi = \max_{w \in E}\text{himem}(w, P)$ and $E.lo = \text{lomem}(w_n, P)$ where w_n is the last node in E. The sorting process is as follows.

While there exist two adjacent units E' and E in P' such that E' is before E and $E'.hi - E'.lo < E.hi - E.lo$,

(a) Swap E' and E. By Lemma 3, this does not increase the memory usage.

(b) If two units of the second kind become adjacent to each other as a result of the swapping, combine the two units into one and recompute its new hi and lo.

When the above sorting process finishes, all units of the first kind, which are components of the indivisible units in $v.seq$, are in the order of decreasing hi-lo difference. Since, for each child u of v, indivisible units in $u.seq$ have been in the correct order before the sorting process, their relative order is not changed. The order of the nodes from $T - T'$ is preserved because the sorting process never swaps two units of the second kind. Also, v and its ancestors do not appear in P', and nodes in units of the first kind are not ancestors or descendants of any nodes in units of the second kind. Therefore, the sorting process does not violate parent-child dependencies.

3. Now that the components of the indivisible units in $v.seq$ are in the correct order, we make the indivisible units contiguous using the following combining process.

For each indivisible unit E in $v.seq$,

(a) In the traversal P, if there are nodes from $T - T'$ in between the nodes from E, move them either to the beginning or the end of E as specified by Lemma 2.

(b) Make the contiguous sequence of nodes from E an indivisible unit.

Upon completion, each indivisible unit in $v.seq$ is contiguous in P and the order in P of the indivisible units is the same as they are in $v.seq$. According to Lemma 2, moving 'foreign' nodes out of an indivisible unit does not increase the memory usage. Also, the order of the nodes from $T - T'$ is preserved. Hence, the combining process does not violate parent-child dependencies.

We use induction to show that the above procedure correctly transforms any given traversal P into an optimal traversal found by the algorithm. The induction hypothesis $H(u)$ for each node u is that:

- the nodes in each indivisible unit in $u.seq$ appear contiguously in P and are in the same order as they are in $u.seq$, and
- the order in P of the indivisible units in $u.seq$ is the same as they are in $u.seq$.

Initially, $H(u)$ is true for every leaf node u because there is only one traversal order for a leaf node. As the induction step, assume $H(u)$ is true for each child u of a node v. The procedure rearranges the nodes in P' such that the nodes that form an indivisible unit in $v.seq$ are contiguous in P, the sets of nodes corresponding to the indivisible units are in the same order in P as they are in $v.seq$, and the order among the nodes that are not in the subtree rooted at v is preserved. Thus, when the procedure finishes processing a node v, $H(v)$ becomes true. By induction, $H(T.root)$ is true and a traversal found by the algorithm is obtained. Since any traversal P can be transformed into a traversal found by the algorithm without increasing memory usage in any transformation step, no traversal can use less memory and the algorithm is correct. □

6 Conclusion

In this paper, we have considered the memory usage optimization problem in the evaluation of expression trees involving large objects of different sizes. This problem arose in the context of optimizing electronic structure calculations. The developed solution would apply in any context involving the evaluation of an expression tree, in which intermediate results are so large that it is impossible to keep all of them in memory at the same time. In such situations, it is necessary to dynamically allocate and deallocate space for the intermediate results and to find an evaluation order that uses the least memory. We have developed an efficient algorithm that finds an optimal evaluation order in $\Theta(n \log^2 n)$ time for an expression tree containing n nodes and proved its correctness.

Acknowledgments

We would like to thank Tamal Dey for help with the complexity proof and for proofreading the paper.

References

[1] A. W. Appel and K. J. Supowit. Generalizations of the Sethi-Ullman algorithm for register allocation, *Software—Practice and Experience*, 17-6(1987), 417–421.

[2] W. Aulbur, *Parallel implementation of quasiparticle calculations of semiconductors and insulators*, Ph.D. Dissertation, Ohio State University, Columbus, October 1996.

[3] M. S. Hybertsen and S. G. Louie, Electronic correlation in semiconductors and insulators: band gaps and quasiparticle energies, *Phys. Rev. B*, 34(1986), 5390.

[4] C. Lam, P. Sadayappan, and R. Wenger, On optimizing a class of multi-dimensional loops with reductions for parallel execution, *Parallel Processing Letters*, 7-2(1997), 157–168.

[5] C. Lam, P. Sadayappan, and R. Wenger, Optimization of a class of multi-dimensional integrals on parallel machines, *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[6] C. Lam, P. Sadayappan, D. Cociorva, M. Alouani, and J. Wilkins, Performance optimization of a class of loops involving sums of products of sparse arrays, *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[7] C. Lam, *Performance optimization of a class of loops implementing multi-dimensional integrals*, Technical report no. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University, Columbus, August 1999.

[8] I. Nakata, On compiling algorithms for arithmetic expressions, *Comm. ACM*, 10(1967), 492–494.

[9] H. N. Rojas, R. W. Godby, and R. J. Needs, Space-time method for Ab-initio calculations of self-energies and dielectric response functions of solids, *Phys. Rev. Lett.*, 74(1995), 1827.

[10] R. Sethi, J. D. Ullman, The generation of optimal code for arithmetic expressions, *J. ACM*, 17-1(1970), 715–728.

[11] R. Sethi, Complete register allocation problems, *SIAM J. Computing*, 4-3(1975), 226–248.