# Memory-optimal evaluation of expression trees involving large objects☆

Chi-Chung Lam[a,1], Thomas Rauber[b], Gerald Baumgartner[c,*], Daniel Cociorva[a,2], P. Sadayappan[a]

[a] *Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA*
[b] *University of Bayreuth, Lehrstuhl für Angewandte Informatik II, D-95447 Bayreuth, Germany*
[c] *Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*

**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. We demonstrate the utility of our algorithm using representative equations from quantum chemistry.

*Keywords:* expression tree, evaluation order, memory minimization, register allocation

## 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 the accurate modeling of the electronic structure of atoms and molecules in quantum chemistry [1, 2] as well as in some computational physics codes modeling the electronic properties of semiconductors and metals [3, 4, 5]. Computational approaches to modeling the structure and interactions of molecules, the electronic and optical

☆This work was supported in part by the National Science Foundation under grants DMR-9520319, CHE-0121676, CNS-0509467, and CCF-0541409.

*Corresponding author. Tel.: +1 225 578 2191; fax: +1 225 578 1465.
[1] Current address: Chemical Abstracts Service, Columbus, OH 43210, USA.
[2] Current address: Department of Chemical Physiology, The Scripps Research Institute, La Jolla, CA 92037, USA.

properties of molecules, the heat and rate of chemical reactions, etc., are crucial to the understanding of chemical processes in real-world systems. Examples of applications include combustion and atmospheric chemistry, chemical vapor deposition, protein structure and enzymatic chemistry, and industrial chemical processing. The computational domain that we consider is also extremely compute-intensive and consumes significant computer resources at national supercomputer centers. Many of these codes are limited in the size of the problem that they can currently solve because of memory and performance limitations.

In this class of computations, the final result to be computed can be expressed in terms of tensor contractions, essentially a collection of multi-dimensional summations of the product of several input arrays. Due to commutativity, associativity, and distributivity, there are many different ways to compute the final result, and they could differ widely in the number of floating point operations required. Consider the following expression:

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel}$$

where typical index ranges are on the order of tens to a few thousands. If this expression is directly translated into code (with ten nested loops, for indices $a - l$), the total number of arithmetic operations required will be $4 \times N^{10}$ if the range of each index $a - l$ is $N$. Instead, the same expression can be rewritten by use of associative and distributive laws [6, 7, 8]:

$$S_{abij} = \sum_{ck} \left( \sum_{df} \left( \sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik}$$

This corresponds to the formula sequence shown in Fig. 1(a) and can be directly translated into code as shown in Fig. 1(b). This form only requires $6 \times N^6$ operations. However, additional space is required to store temporary arrays $T1$ and $T2$. Often, the space requirements for the temporary arrays poses a serious problem. For this example, abstracted from a quantum chemistry model, the array extents along indices $a - d$ are the largest, while the extents along indices $i - l$ are the smallest. Therefore, the size of temporary array $T1$ would dominate the total memory requirement.

Thus, although the latter form is far more economical in terms of the number of operations, its implementation will require the use of temporary intermediate arrays to hold the partial results of the parenthesized array subexpressions.

One approach to reducing the memory requirements for the computation is through loop fusion. By merging the common outer loops of the producer and consumer loop nests for an intermediate array, the dimensions corresponding to the fused loops can be eliminated from the intermediate array. In our example, loop fusion allows $T1$ to be reduced to a scalar and $T2$ to a 2-dimensional array without changing the number of operations, as illustrated in Fig. 1(c). Since different fusion choices are often not mutually compatible, it is necessary to enumerate all fusion choices to find the loop structure that minimizes the memory requirements [9, 10, 11].

If even after loop fusion, some intermediates do not fit in memory, it is necessary to tile these intermediates and move tiles in and out of disk [12, 13].

While loop fusion usually results in large memory reductions, in the context of tensor contractions it has a detrimental effect: it reduces temporal and spatial locality. As a result, the computation can become significantly more expensive. E.g., while the tensor contractions in Fig. 1(b) can be implemented using index permutations and BLAS matrix multiplications, which use the cache effectively, this is not possible anymore with the fused code in Fig. 1(c).

$$T1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel}$$
$$T2_{bcjk} = \sum_{df} T1_{bcdf} \times C_{dfjk}$$
$$S_{abij} = \sum_{ck} T2_{bcjk} \times A_{acik}$$

(a) Formula sequence

```
T1=0; T2=0; S=0;
for b, c, d, e, f, l
| T1bcdf += Bbefl Dcdel
for b, c, d, f, j, k
| T2bcjk += T1bcdf Cdfjk
for a, b, c, i, j, k
| Sabij += T2bcjk Aacik
```

(b) Direct implementation (unfused code)

```
S = 0;
for b, c
| T1f = 0; T2f = 0;
| for d, f
| | for e, l
| | | T1f += Bbefl Dcdel
| | for j, k
| | | T2fjk += T1f Cdfjk
| for a, i, j, k
| | Sabij += T2fjk Aacik
```

(c) Memory-reduced implementation (fused)

Figure 1: Example illustrating use of loop fusion for memory reduction

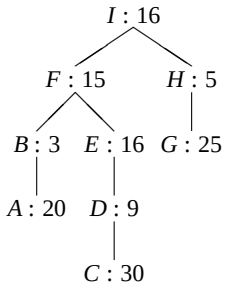



Figure 2: An example expression tree

Since space allocated for an intermediate array can be deallocated as soon as the operation using the array has been performed, the evaluation order affects the maximum memory requirements as well [14, 11]. Our strategy is, therefore, to attempt to reduce the memory requirements by improving the evaluation order first in the hope of avoiding loop fusion. Even if loop fusion and disk I/O become necessary, the improved evaluation order can reduce the memory pressure and lead to less disk I/O.

In this paper, we focus on the problem of finding an evaluation order of the nodes in a given expression tree that minimizes the dynamic memory usage. A solution to this problem would result in the generation of more efficient code for evaluating expression trees, e.g., for computing tensor contraction expressions.

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. The data objects for leaf nodes are assumed to be generated or read in as needed. Therefore, space for them is allocated and deallocated in the same way. 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.

A simpler problem related to the memory usage minimization problem is the register allocation problem for binary expression trees in which the sizes of all nodes are unity. It has been addressed in [15, 16] and can be solved in $\Theta(n)$ 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 [17]. The algorithm in [16] for expression trees of unit-sized nodes does not extend directly to expression trees having nodes of different sizes. Appel and Supowit [18] generalized the register allocation problem to higher degree expression trees of arbitrarily-sized nodes. However, they restricted their attention to solutions that evaluate subtrees contiguously, which could result in non-optimal solutions (as we show through an example in Section 2). The generalized register allocation problem for expression trees was addressed in the context of vector machines in [19, 20].

Our approach is different from out-of-core solutions. There the problem is to minimize disk I/O under a given memory constraint (not to minimize memory) by restructuring the computation or with the help of data structures. We attempt to avoid an out-of-core computation by minimizing memory. If this is unsuccessful, we need to resort to other techniques, such as tiling, for producing an out-of-core solution [12, 13].

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. In Section 6, we show experimental results. Section 7 contains conclusions and discusses possible future work.

## 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. 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. Similarly, a leaf node object is allocated before it is created or read from disk and deallocated after 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

| 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

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

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

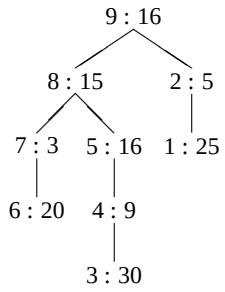
Here, $\text{himem}(v_i, P)$ is the memory usage during the evaluation of $v_i$ in the traversal $P$, and $\text{lomem}(v_i, P)$ is the memory usage upon completion of the same evaluation. These definitions reflect that we need to allocate space for $v_i$ before its evaluation, and that after evaluation 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. In 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 [16], 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, and is not optimal. The optimal traversal, which uses only 39 units of memory, is $\langle C, D, G, H, A, B, E, F, I\rangle$ (see Fig. 5). 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)

| 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) Memory usage

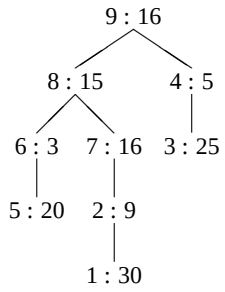



(b) Order of node visits

Figure 4: Best traversal with contiguous evaluation of subtrees of the expression tree in Fig. 2

| 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 | |

(a) Memory usage




(b) Order of node visits

Figure 5: Optimal traversal of the expression tree in Fig. 2

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). 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 that 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

| Node $v$ | Subtree traversals merged in decreasing $hi - lo$ order | Optimal traversal $v.seq$ |
|---|---|---|
| $A$ | $\langle(A, 20, 20)\rangle$ | $\langle(A, 20, 20)\rangle$ |
| $B$ | $\langle(A, 20, 20), (B, 23, 3)\rangle$ | $\langle(AB, 23, 3)\rangle$ |
| $C$ | $\langle(C, 30, 30)\rangle$ | $\langle(C, 30, 30)\rangle$ |
| $D$ | $\langle(C, 30, 30), (D, 39, 9)\rangle$ | $\langle(CD, 39, 9)\rangle$ |
| $E$ | $\langle(CD, 39, 9), (E, 25, 16)\rangle$ | $\langle(CD, 39, 9), (E, 25, 16)\rangle$ |
| $F$ | $\langle(CD, 39, 9), (AB, 32, 12), (E, 28, 19), (F, 34, 15)\rangle$ | $\langle(CD, 39, 9), (ABEF, 34, 15)\rangle$ |
| $G$ | $\langle(G, 25, 25)\rangle$ | $\langle(G, 25, 25)\rangle$ |
| $H$ | $\langle(G, 25, 25), (H, 30, 5)\rangle$ | $\langle(GH, 30, 5)\rangle$ |
| $I$ | $\langle(CD, 39, 9), (GH, 39, 14), (ABEF, 39, 20), (I, 36, 16)\rangle$ | $\langle(CDGHABEFI, 39, 16)\rangle$ |

Figure 6: Optimal traversals for the subtrees in the expression tree in Fig. 2

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 indivisible unit 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 and in the same order. E.g., if there is an optimal traversal of the entire expression tree in which $B$ is evaluated immediately following $A$, the algorithm will construct an indivisible unit containing the nodes $AB$. (For clarity, we write node lists in an indivisible unit as strings). Therefore, as we show later, inserting any node in between the nodes of an indivisible unit 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*.

For example, the traversal $\langle(CD, 39, 9), (E, 25, 16)\rangle$ indicates that the maximum memory usage during the evaluation of $C$ and $D$ is 39 units of memory and that the result of the evaluation occupies 9 units. Similarly, during the evaluation of $E$, 25 units are needed and the result occupies 16 units. While $C$ and $D$ can be evaluated consecutively in an optimal traversal of the entire expression tree, nodes from other subtrees might be evaluated between $D$ and $E$.

The algorithm always maintains the elements of a traversal in decreasing *hi* and increasing *lo* order, which implies in order of decreasing $hi - lo$. 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 and compute the optimal traversals for the subtrees rooted at each node as shown in Fig. 6. 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 $\langle(AB, 23, 3)\rangle$. 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 $E.seq$ in the order of decreasing $hi - lo$. 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 have been 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 $hi - lo$ order. 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. 6. 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. 5).

We can visualize the elements in a traversal as follows. The first element in a traversal contains the nodes from the beginning of the traversal up to and including the last node that has the lowest lomem after the last node that has the highest himem. The highest himem and the lowest lomem become the *hi* and the *lo* of the first indivisible unit, respectively. The same rules are recursively applied to the remainder of the traversal to form the second element, and so on. This ensures that the indivisible units in the traversal are in decreasing *hi* and increasing *lo* order.

Fig. 7 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 $v$. For simplicity, we present the algorithm using sequences of (*nodelist*, *hi*, *lo*) triples as the data structure for traversals. We will present a more efficient data structure in the next section.

The optimal traversal $v.seq$ for a node $v$ is obtained by optimally merging together the optimal traversals $u.seq$ for all children $u$ 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 of sequences 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 $hi - lo$. This order guarantees that the indivisible units are arranged to minimize memory usage. Since $S1$ and $S2$ are formed independently, the *hi* and *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*)

```
MinMemTraversal (T):
    foreach node v in some bottom-up traversal of T
      v.seq = ⟨⟩
      foreach child u of v       // merge traversals from all children
        v.seq = 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, ⟨v⟩, v.size + base, v.size)       // append parent node v
    nodelist = ⟨⟩
    for i = 1 to |T.root.seq|       // flatten sequence of nodelists
      nodelist = nodelist + T.root.seq[i].nodelist       // + is concatenation
    return nodelist       // memory usage is T.root.seq[1].hi

MergeSeq (S1, S2):
    S = ⟨⟩       // S will hold the merge result
    i = j = 1
    base1 = base2 = 0
    while i ≤ |S1| or j ≤ |S2|
      if j > |S2| or (i ≤ |S1| and S1[i].hi − S1[i].lo > S2[j].hi − S2[j].lo) then
        // S2 is exhausted or S1[i] has a larger hi − lo than S2[j]
        // append indivisible unit from S1 to S
        AppendSeq (S, S1[i].nodelist, S1[i].hi + base1, S1[i].lo + base1)
        base2 = S1[i].lo
        i++
      else
        // S1 is exhausted or S2[j] has a larger or equal hi − lo than S1[i]
        // append indivisible unit from S2 to S
        AppendSeq (S, S2[j].nodelist, S2[j].hi + base2, S2[j].lo + base2)
        base1 = S2[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 ≥ 1 and (E.hi ≥ S[i].hi or E.lo ≤ S[i].lo)
      // combine S[i] with E to keep S in decreasing hi and increasing lo order
      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 7: Procedure for finding an memory-optimal traversal of an expression tree

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. 6).

## 4. Complexity of the algorithm

In the previous section, our algorithm has been presented using an abstract sequence notation so that the properties of sequences of indivisible units are easy to see. If sequences were implemented as doubly-linked lists and merged sequentially, the worst-case complexity for an unbalanced $n$-node tree would be $O(n^2)$, since for every node a linear-time merge operation would be performed.

With a more efficient data structure, the time complexity of our algorithm (the **MemMinTraversal** procedure) 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 in decreasing $hi - lo$ order. 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 traversing the expression tree, the cost for building sequences of indivisible units, and the cost for combining indivisible units into larger indivisible units. The cost for traversing the expression tree is linear, since **MemMinTraversal** visits each node in the expression tree only once. 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 call the **AppendSeq** procedure to append an element containing the node to the sequence of its subtree, which costs $O(\log n)$ time for inserting the element in the red-black tree.

For an $m$-ary interior node, the **MergeSeq** procedure must 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 the **AppendSeq** procedure 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 re-balanced, 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, indivis-

ible 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 sum of the sizes of the two nodes 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 that are maintained by the algorithm. 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 as Theorem 4 the optimality of the traversal found by 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 1.** *Let $v$ be any node in an expression tree, $S = v.\text{seq}$, and $P$ be the traversal represented by $S$ of the subtree rooted at $v$, i.e., $P = S[1].\text{nodelist} + \cdots + S[|S|].\text{nodelist}$. The algorithm maintains the following invariants:*

> *For all $1 \le i \le |S|$, let $S[i].\text{nodelist} = \langle v_1, v_2, \ldots, v_n \rangle$ and $v_m$ be the last node in $S[i].\text{nodelist}$ that has the maximum himem value, i.e., for all $k < m$, $himem(v_k, P) \le himem(v_m, P)$ and for all $k > m$, $himem(v_k, P) < himem(v_m, P)$. Then, we have,*
>
> 1. $S[i].\text{hi} = himem(v_m, P)$,
> 2. $S[i].\text{lo} = lomem(v_n, P)$,
> 3. *for all* $m \le k \le n$, $lomem(v_k, P) \ge lomem(v_n, P)$,
> 4. *for all* $1 \le j < i$,
>    (a) *for all* $1 \le k \le n$, $S[j].\text{hi} > himem(v_k, P)$,
>    (b) *for all* $1 \le k \le n$, $S[j].\text{lo} < lomem(v_k, P)$,
>    (c) $S[j].\text{hi} > S[i].\text{hi}$, *and*
>    (d) $S[j].\text{lo} < S[i].\text{lo}$.

Proof. The above invariants are true by construction. The invariants (1) and (2) are maintained by adding the appropriate *base* value to the *hi* and *lo* arguments in the calls of **AppendSeq**. The invariant (3) is maintained by **AppendSeq**, and (4) is maintained by the merge order in **MergeSeq** and by **AppendSeq**. □

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 has been formed as part of the traversal of a subtree, we do not need to consider breaking it up later for inserting nodes from another subtree. This lemma allows the **MemMinTraversal** algorithm 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 2.** *Let $v$ be a node in an expression tree $T$, $S = v$.*seq*, and $P$ be a traversal of $T$ in which the nodes from $S[i]$.*nodelist *appear in the same order as they are in $S[i]$.*nodelist*, but not contiguously. Then, any nodes that are in between the nodes in $S[i]$.*nodelist *can always be moved to the beginning or the end of $S[i]$.*nodelist *without increasing memory usage, provided that none of the nodes that are in between the nodes in $S[i]$.*nodelist *are ancestors or descendants of any nodes in $S[i]$.*nodelist*.*

Proof. Let $S[i].nodelist = \langle v_1, \ldots, v_n \rangle$, $v_0$ be the node before $v_1$ in $S$, $v_m$ be the node in $S[i].nodelist$ such that for all $k < m$, $\text{himem}(v_k, P) \leq \text{himem}(v_m, P)$ and for all $k > m$, $\text{himem}(v_m, S) > \text{himem}(v_k, S)$. Let $v'_1, \ldots, v'_b$ be the 'foreign' nodes, i.e., the nodes that are in between the nodes in $S[i].nodelist$ 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 $S[i].nodelist$. We construct another traversal $P'$ of $T$ from $P$ by moving $v'_1, \ldots, v'_a$ to the beginning of $S[i].nodelist$ and $v'_{a+1}, \ldots, v'_b$ to the end of $S[i].nodelist$. 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 $P'$.

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 $\text{himem}(v_m, P') = \text{himem}(v_m, S) + \text{lomem}(v'_a, Q) = \text{himem}(v_m, P)$.
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 $\text{himem}(v_k, S) \leq \text{himem}(v_m, S)$ implies that $\text{himem}(v_k, P') = \text{himem}(v_k, S) + \text{lomem}(v'_a, Q) \leq \text{himem}(v_m, S) + \text{lomem}(v'_a, Q) = \text{himem}(v_m, P)$.
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 $1 \leq k \leq m$, $\text{lomem}(v_0, S) < \text{lomem}(v_k, S)$ (by invariant 4(b) in Lemma 1) implies $\text{himem}(v'_j, P') = \text{himem}(v'_j, Q) + \text{lomem}(v_0, S) \leq \text{himem}(v'_j, P)$.
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 $m \leq k \leq n$, $\text{lomem}(v_k, S) \geq \text{lomem}(v_n, S)$ (by invariant 3 in Lemma 1) implies $\text{himem}(v'_j, P') = \text{himem}(v'_j, Q) + \text{lomem}(v_n, S) \leq \text{himem}(v'_j, P)$.

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 not increase as a result of moving them, the overall maximum memory usage cannot increase. □

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

| 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 8: Memory usage comparison of two traversals in Lemma 3

**Lemma 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 *uses the least memory.*

Proof. Let $M$ be a merge of $S$ and $S'$ that is not in the order of decreasing *hi* − *lo*. 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. 8.

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* 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. □

**Theorem 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 indirectly 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 $hi - lo$. 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. □

| Equation | terms | left-to-right | right-to-left | optimal | opt. time |
|---|---|---|---|---|---|
| Example from Figure 1(b) | 1 | 1.6GB | 2.0GB | 1.6GB | 0.023ms |
| CCSD Energy, small | 3 | 1,624B | 1,608B | 1,608B | 0.025ms |
| CCSD Singles, small | 14 | 6,888B | 6,080B | 6,000B | 0.063ms |
| CCSD Doubles, small | 31 | 17,968B | 32,800B | 15,800B | 0.161ms |
| CCSD Energy, large | 3 | 400MB | 400MB | 400MB | 0.026ms |
| CCSD Singles, large | 14 | 600MB | 600MB | 600MB | 0.065ms |
| CCSD Doubles, large | 31 | 1.8GB | 2.2GB | 1.4GB | 0.159ms |
| CCSDTQ Lambda 1 | 264 | 8.0EB | 8.0EB | 8.0EB | 2.761ms |

Table 1: Memory usage for different traversals and the running time of our optimization algorithm

$$R^{p1p2}_{h1h2} = v\_vvoo^{p1p2}_{h1h2} - t\_vo^{p1}_{h7} * (v\_ovoo^{h7p2}_{h1h2} - 0.5 * t\_vo^{p2}_{h8} * (v\_oooo^{h7h8}_{h1h2} + (v\_ooov^{h7h8}_{h1p3} - 0.5 * v\_oovv^{h7h8}_{p3p4} * t\_vo^{p4}_{h1}) * t\_vo^{p3}_{h2} + 0.5 * t\_vvoo^{p5p6}_{h1h2} * v\_oovv^{h7h8}_{p5p6}) + t\_vo^{p3}_{h2} * (v\_ovov^{h7p2}_{h1p3} - 0.5 * t\_vo^{p4}_{h1} * v\_ovvv^{h7p2}_{p3p4}) - (f\_ov^{h7}_{p3} - t\_vo^{p4}_{h4} * v\_oovv^{h4h7}_{p3p4}) * t\_vvoo^{p2p3}_{h1h2} - t\_vvoo^{p2p7}_{h2h4} * (v\_ooov^{h4h7}_{h1p7} + t\_vo^{p3}_{h1} * v\_oovv^{h4h7}_{p3p7}) + 0.5 * t\_vvoo^{p3p4}_{h1h2} * v\_ovvv^{h7p2}_{p3p4}) + t\_vo^{p3}_{h2} * (v\_vvov^{p1p2}_{h1p3} - 0.5 * t\_vo^{p4}_{h1} * v\_vvvv^{p1p2}_{p3p4}) + (f\_oo^{h6}_{h1} + (f\_ov^{h6}_{p6} + t\_vo^{p4}_{h4} * v\_oovv^{h4h6}_{p4p6}) * t\_vo^{p6}_{h1} - t\_vo^{p4}_{h4} * v\_ooov^{h4h6}_{h1p4} - 0.5 * t\_vvoo^{p4p5}_{h1h5} * v\_oovv^{h5h6}_{p4p5}) * t\_vvoo^{p1p2}_{h2h6} + (f\_vv^{p2}_{p3} - t\_vo^{p4}_{h4} * v\_ovvv^{h4p2}_{p3p4} - 0.5 * t\_vvoo^{p2p4}_{h4h5} * v\_oovv^{h4h5}_{p3p4}) * t\_vvoo^{p1p3}_{h1h2} + 0.5 * t\_vvoo^{p1p2}_{h6h8} * (v\_oooo^{h6h8}_{h1h2} + t\_vo^{p6}_{h2} * (v\_ooov^{h6h8}_{h1p6} + 0.5 * t\_vo^{p4}_{h1} * v\_oovv^{h6h8}_{p4p6}) - 0.5 * t\_vvoo^{p3p4}_{h1h2} * v\_oovv^{h6h8}_{p3p4}) + t\_vvoo^{p1p3}_{h2h3} * (v\_ovov^{h3p2}_{h1p3} - t\_vo^{p5}_{h1} * v\_ovvv^{h3p2}_{p3p5} - 0.5 * t\_vvoo^{p2p5}_{h1h5} * v\_oovv^{h3h5}_{p3p5}) + 0.5 * t\_vvoo^{p3p4}_{h1h2} * v\_vvvv^{p1p2}_{p3p4}$$

Figure 9: The spin-orbital CCSD Doubles equation

## 6. Experiments

We have implemented our algorithm in the Tensor Contraction Engine [21] and tested it on a variety of equations. Table 1 shows the results of several representative equations.

For the example from Figure 1(b), we used the following array dimensions: $V = 100$, $O = 50$.

The Coupled Cluster Singles and Doubles (CCSD) equations are the equations for an important quantum chemistry model. We used the shorter spin-orbital variant of these equations. Each term in these equations is the contraction of up to five two- and four-dimensional tensors. To illustrate the complexity of these equations, Figure 9 shows the factored (parenthesized) CCSD Doubles equation in Einstein notation, where indices that occur twice in a term (once as upper and once as lower index) are implicitly summed over. The *h* indices represent occupied orbitals of range *O*; the *p* indices represent virtual orbitals of range *V*. For our measurements, we used two different dimension sizes: $O = 5$ and $V = 2$ for modeling a water molecule and $O = 50$ and $V = 100$ for modeling a larger molecule.

The Coupled Cluster Singles, Doubles, Triples, and Quadruples (CCSDTQ) equation is the largest equation in the TCE testsuite with 264 terms of up to six two-, four-, six-, and eight-dimensional tensors. We used $O = 10$ and $V = 100$.

For many equations, either a left-to-right or a right-to-left post-order traversal will find a solution that is close to optimal. Which of these traversals works best depends on how the equation was written or which tool generated the equation. A hand-factorized equation, such as the CCSD Doubles equation, typically has less structure than an equation that was generated and, therefore, provides more opportunity for reducing the memory usage. Some equations, such as CCSDTQ, are dominated by a few terms involving very big tensors such that any improvements in the evaluation order are minimal. The optimal traversal for CCSDTQ uses 80GB less memory

than the right-to-left traversal and slightly less memory than the left-to-right traversal, but these differences are insignificant compared to the 8EB term. Similarly, the CCSD Doubles equation would be dominated by the terms involving *v_vvvv* if *V* were orders of magnitude larger than *O*. However, for many interesting tensor sizes the improvement is significant and can result in the computation to fit in memory without resorting to loop fusion.

We measured the performance of our algorithm by averaging the running time over 100 runs. The measurements were performed on a Lenovo Thinkpad T400 with a 2.8GHz Core 2 Duo processor. The results are shown in the right-most column of Table 1.

The implementation of our algorithm simply represents traversals as doubly-linked lists instead of using a balanced tree. Even so, the algorithm computes the optimal traversal for most examples in less than 0.5ms and takes only 2.76ms for CCSDTQ. Our implementation of the $O(n)$ post-order traversal is actually slightly slower in most cases, since we use the same data structure for representing traversals. It takes 2.80ms for CCSDTQ. A simple list of nodes, instead of a list of indivisible units containing one-element node lists, would have been sufficient as the data structure for a post-order traversal. The efficiency of our algorithm compared to the post-order traversal does show, though, that a data structure with a balanced tree will likely only pay off for much larger equations.

## 7. 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.

The problem that has been presented is very similar to that of register allocation for expression trees, except that the definition of the cost functions himem and lomem would be different, since in machine instructions one of the argument registers is typically reused for the result of the instruction. For binary expression trees with unit-sized nodes, our algorithm would find the same evaluation order as the Sethi-Ullman algorithm [16]. Our strategy of evaluating indivisible units in $hi - lo$ order is similar to that of the Appel-Supowit algorithm [18], except that the latter evaluates entire subtree contiguously and then sorts the subtrees in $hi - lo$ order, which is suboptimal, as we have shown in Section 2.

Modern compilers employ register allocators based on graph coloring [22]. While a graph-coloring register allocator is an approximation algorithm, it allows register allocation across multiple expression trees or for directed acyclic graphs. Using the Sethi-Ullman algorithm for determining the evaluation order of individual expressions will minimize the number of simultaneously live temporaries and improve the result of a graph-coloring register allocator [23].

Our algorithm has some limitations. It assumes that the data for leaf nodes is produced or read in, and it minimizes the memory usage for evaluating a single expression tree. If the resulting evaluation order does not fit in memory, we need to resort to loop fusion [9, 14] and tiling [12] for producing an out-of-core solution. For future work, we are planning to explore the use

of **MinMemTraversal** together with a generalized graph-coloring register allocator for producing out-of-core solutions for sequences of expression trees. Such an approach could also allow leaf nodes that are already memory-resident as well as directed acyclic graphs. Similar as with the combination of the Sethi-Ullman algorithm and a graph-coloring register allocator in a traditional compiler, our algorithm could produce an evaluation order for the individual expression trees. Then, following liveness analysis, a graph-coloring register allocator with heuristics for large data objects could decide which nodes need to be spilled, i.e., temporarily stored on disk. Since loop fusion can result in poor temporal locality and tiling results in temporaries being read in repeatedly, a combination of our algorithm with a graph-coloring register allocator has the potential of producing competitive out-of-core solutions.

Another possible generalization of our algorithm is to use it for parallelizing the evaluation of an expression tree by scheduling different indivisible units for evaluation on different (sets of) processors. Since the last node in an indivisible unit has a small lomem, it is a good candidate for sending to another processor, since it will help keep the communication cost small. For achieving load balancing, the algorithm could be turned into a dynamic programming algorithm that computes all solutions with minimal memory usage as well as estimates of the communication and computation cost for each solution. The communication and computation costs would then be used for selecting among multiple solutions with minimal memory usage and for limiting the size of indivisible units to aid in load balancing.

## Acknowledgments

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

## Role of the funding source

This work was supported in part by the National Science Foundation under grants DMR-9520319, CHE-0121676, CNS-0509467, and CCF-0541409. The National Science Foundation had no involvement in any aspect of the research.

## References

[1] T. J. Lee, G. E. Scuseria, Achieving chemical accuracy with coupled cluster theory, in: S. R. Langhoff (Ed.), Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997, pp. 47–109.

[2] J. M. L. Martin, Benchmark studies on small molecules, in: P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, H. F. S. P. Kollman (Eds.), Encyclopedia of Computational Chemistry, Vol. 1, Wiley & Sons, Berne, Switzerland, 1998, pp. 115–128.

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

[4] H. N. Rojas, R. W. Godby, 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.

[5] W. Aulbur, Parallel implementation of quasiparticle calculations of semiconductors and insulators, Ph.D. thesis, The Ohio State University (Oct. 1996).

[6] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. Bernholdt, S. Hirata, C. Lam, R. Pitzer, J. Ramanujam, P. Sadayappan, Automated operation minimization of tensor contraction expressions in electronic structure calculations, in: Proc. 5th International Conference on Computational Science, Vol. 3514 of Lecture Notes in Computer Science, Springer-Verlag, Atlanta, GA, 2005, pp. 155–164.

[7] C. Lam, P. Sadayappan, R. Wenger, On optimizing a class of multi-dimensional loops with reductions for parallel execution, Parall. Process. Lett. 7 (2) (1997) 157–168.
[8] C. Lam, P. Sadayappan, R. Wenger, Optimization of a class of multi-dimensional integrals on parallel machines, in: Eighth SIAM Conference on Parallel Processing for Scientific Computing, Society for Industrial and Applied Mathematics, Minneapolis, MN, 1997.
[9] A. Bibireata, S. Krishnan, D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, V. Choppella, Memory-constrained data locality optimization for tensor contractions, in: Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing, Vol. 2958 of Lecture Notes in Computer Science, Springer-Verlag, College Station, Texas, 2003, pp. 93–108.
[10] C. Lam, D. Cociorva, G. Baumgartner, P. Sadayappan, Memory-optimal evaluation of expression trees involving large objects, in: Proceedings of the 1999 International Conference on High-Performance Computing, Vol. 1746 of Lecture Notes in Computer Science, Springer-Verlag, Calcutta, India, 1999, pp. 103–110.
[11] C. Lam, Performance optimization of a class of loops implementing multi-dimensional integrals, Ph.D. thesis, The Ohio State University, also available as Technical Report No. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University (Aug. 1999).
[12] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, V. Choppella, Data locality optimization for synthesis of efficient out-of-core algorithms., in: Proceedings of the the International Conference on High-Performance Computing, Vol. 2913 of Lecture Notes in Computer Science, Springer-Verlag, Hyderabad, India, 2003, pp. 406–417.
[13] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, R. Harrison, Towards automatic synthesis of high-performance codes for electronic structure calculations: Data locality optimization, in: Proceedings of the 2001 International Conference on High-Performance Computing, Vol. 2228 of Lecture Notes in Computer Science, Springer-Verlag, Hyderabad, India, 2001, pp. 237–248.
[14] C. Lam, D. Cociorva, G. Baumgartner, P. Sadayappan, Optimization of memory usage requirement for a class of loops implementing multi-dimensional integrals, in: Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing, Vol. 1863 of Lecture Notes in Computer Science, Springer-Verlag, San Diego, CA, 1999, pp. 350–364.
[15] I. Nakata, On compiling algorithms for arithmetic expressions, Commun. ACM 10 (1967) 492–494.
[16] R. Sethi, J. D. Ullman, The generation of optimal code for arithmetic expressions, J. ACM 17 (1) (1970) 715–728.
[17] R. Sethi, Complete register allocation problems, SIAM J. Comput. 4 (3) (1975) 226–248.
[18] A. W. Appel, K. J. Supowit, Generalizations of the Sethi-Ullman algorithm for register allocation, Software: Practice and Experience 17 (6) (1987) 417–421.
[19] T. Rauber, Ein Compiler für Vektorrechner mit optimaler Auswertung von vektoriellen Ausdrucksbäumen, Ph.D. thesis, University Saarbrücken (1990).
[20] T. Rauber, Optimal evaluation of vector expression trees, in: Proceedings of the 5th Jerusalem Conference on Information Technology, IEEE Computer Society Press, Jerusalem, Israel, 1990, pp. 467–473.
[21] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, A. Sibiryakov, Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models, Proceedings of the IEEE 93 (2) (2005) 276–292.
[22] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, P. W. Markstein, Register allocation via coloring, Computer Languages 6 (1981) 47–57.
[23] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, 2nd Edition, Cambridge University Press, Cambridge, United Kingdom, 2002.

**Chi-Chung Lam** graduated from The Ohio State University with a Ph.D. degree in computer and information science in 1999. The title of his dissertation was Performance Optimization of a Class of Loops Implementing Multi-dimensional Integrals. He is currently employed as a Senior Systems Engineer at the American Chemical Society in Columbus, Ohio.

**Thomas Rauber** received his Ph.D. in Computer Science from the University des Saarlandes in 1990. From 1990 to 2002, he has been professor for Computer Science at the Martin-Luther University Halle/Wittenberg, Germany. Since 2002, he is professor at the University of Bayreuth where he hold the chair for parallel and distributed systems. His research interests include parallel and distributed programming, compiler optimizations, and performance modeling for parallel and distributed environments.

**Gerald Baumgartner** received the Dipl.-Ing. degree from the University of Linz, Austria, and M.S. and Ph.D. degrees from Purdue University, all in computer science. He began his academic career at The Ohio State University in 1997. Since 2004 he is at the Department of Computer Science at Louisiana State University. His research interest includes compiler optimizations, the design and implementation of domain-specific and object-oriented languages, desktop grids, and development and testing tools for object-oriented and embedded systems programming.

**Daniel Cociorva** obtained his B.S. and M.S. degrees in Theoretical Physics from University of Lyon, France. He started working in Computational Physics as a graduate student at The Ohio State University in Columbus, Ohio. In his Ph.D. thesis, completed in 2001, he used advanced numerical methods to study properties of interfaces and defects in semiconductor structures. As a postdoctoral associate, he worked on the Tensor Contraction Engine project for automatic code generation and optimization in Quantum Chemistry. He is currently employed as a Bioinformatics Analyst in mass spectrometry proteomics at the Scripps Research Institute in La Jolla, California.

**P. Sadayappan** received the B.Tech. degree from the Indian Institute of Technology, Madras, India, and an M.S. and a Ph.D. from the State University of New York at Stony Brook, all in Electrical Engineering. He is currently a Professor in the Department of Computer Science and Engineering at The Ohio State University. His research interests include Compile-time/Run-time Optimization and Scheduling and Resource Management for Parallel/Distributed Systems.