Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints[*]

Daniel Cociorva, Xiaoyang Gao, Sandhya Krishnan, Gerald Baumgartner,
Chi-Chung Lam, P. Sadayappan
Department of Computer and Information Science
The Ohio State University
{cociorva,gaox,krishnas,gb,clam,saday}@cis.ohio-state.edu

J. Ramanujam
Department of Electrical and Computer Engineering
Louisiana State University
jxr@ece.lsu.edu

Abstract

The accurate modeling of the electronic structure of atoms and molecules involves computationally intensive tensor contractions involving large multi-dimensional arrays. The efficient computation of complex tensor contractions usually requires the generation of temporary intermediate arrays. These intermediates could be extremely large, but they can often be generated and used in batches through appropriate loop fusion transformations. To optimize the performance of such computations on parallel computers, the total amount of inter-processor communication must be minimized, subject to the available memory on each processor. In this paper, we address the memory-constrained communication minimization problem in the context of this class of computations. Based on a framework that models the relationship between loop fusion and memory usage, we develop an approach to identify the best combination of loop fusion and data partitioning that minimizes inter-processor communication cost without exceeding the per-processor memory limit. The effectiveness of the developed optimization approach is demonstrated on a computation representative of a component used in quantum chemistry suites.

1 Introduction

The development of high-performance parallel programs for scientific applications is often very tediuos and time consuming. The time to develop an efficient parallel program for a computational model can be a primary limiting factor in the rate of progress of the science. Our overall goal is to develop a program synthesis system to facilitate the rapid development of high-performance parallel programs for a class of scientific computations encountered in quantum chemistry. The domain of our focus is electronic structure calculations, as exemplified by coupled cluster methods [4], in which many computationally intensive components are expressible as a set of tensor contractions (explained later with an example). We are developing a program synthesis system that will transform an algebraic formula expressed in a high-level notation into efficient parallel code tailored to the characteristics of the target architecture.

A number of compile-time optimizations are being incorporated into the program synthesis system. These include algebraic transformations to minimize the number of arithmetic operations [13, 16], loop fusion and array contraction for memory space minimization [15, 16], tiling and data locality optimization [2], and space-time trade-off optimization [3]. Since the problem of determining the set of algebraic transformations to minimize operation count was found to be NP-complete, we developed a pruning search procedure [13] that is very efficient in practice. The operation-minimization procedure results in the creation of intermediate temporary arrays. Instead of directly computing the result using the input arrays, the number of operations can often be significantly reduced by suitable choice of some intermediate arrays (an example is provided later). However, these intermediate arrays that help in reducing the computational cost can create a problem with the memory required. Loop fusion was found to be effective in significantly reducing the total memory requirement. However, since some fusions could prevent other fusions, the choice of the optimal set of fusion transformations is important. So we addressed the problem of finding the choice of loop fusions for a given operator tree that minimizes the space required for all intermediate arrays after fusion [15, 14].

In this paper we address the optimization of a parallel implementation of this class of computations. If memory were abundant, the issue would be that of determining the optimal distributions/re-distributions of the various multi-dimensional arrays and distributed implementations of the collection of tensor contractions, which are essentially generalized matrix products on higher dimensional arrays (an example is provided later). We use a generalization of Cannon's matrix multiplication algorithm [12] as the basis for the individual contractions.

[*]Supported in part by the National Science Foundation through the Information Technology Research program (CHE-0121676 and CHE-0121706), and NSF grants CCR-0073800 and EIA-9986052.

For many problems of practical interest to quantum chemists, the available memory even on clusters is insufficient to implement the operation-minimal form of the computation, unless array contractions through loop fusion are performed. Hence the problem that we address is the following: given a set of computations expressed as a sequence of tensor contractions, and a specified limit on the amount of available memory on each processor, determine the set of loop fusions and choice of array distributions that will minimize the communication overhead for a parallel implementation of the contractions, based on Cannon's algorithm. In this paper, we present a framework that we have developed to solve this problem. Experimental results on a cluster are provided, that demonstrate the effectiveness of the developed algorithm.

The computational structures that we target arise in scientific application domains that are extremely compute-intensive and consume significant computer resources at national supercomputer centers. They are present in various computational chemistry codes such as ACES II, GAMESS, Gaussian, NWChem, PSI, and MOLPRO. In particular, they comprise the bulk of the computation with the coupled cluster approach to the accurate description of the electronic structure of atoms and molecules [17, 20]. Computational approaches to modeling the structure and interactions of molecules, the electronic and optical properties of molecules, the heats and rates of chemical reactions, etc., are very important to the understanding of chemical processes in real-world systems.

This paper is organized as follows. In the next section, we elaborate on the computational context of interest and the pertinent optimization issues. Section 3 discusses the interaction between distribution of arrays and loop fusion, and describes our algorithm for the memory-constrained communication minimization problem. Section 4 presents results from the application of the new algorithm to an example abstracted from NWChem [9]. We discuss related work in Section 5. Conclusions are provided in Section 6.

2 Elaboration of problem addressed

In the class of computations considered, the final result to be computed can be expressed as multi-dimensional summations of the product of several input arrays. Due to commutativity, associativity, and distributivity, there are many different ways to obtain the same final result and they could differ widely in the number of floating point operations required. Consider the following example:

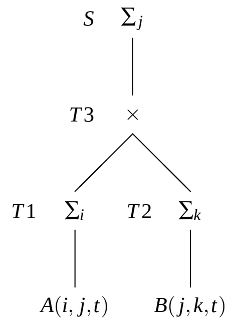
$$S(t) = \sum_{i,j,k} A(i,j,t) \times B(j,k,t) \tag{1}$$

If implemented directly as expressed above, the computation would require $2N_iN_jN_kN_t$ arithmetic operations to compute. However, assuming associative reordering of the operations and use of distributive law of multiplication over addition is acceptable for the floating-point computations, the above computation can be rewritten in various ways. One equivalent form that only requires $N_iN_jN_t + N_jN_kN_t + 2N_jN_t$ operations is as shown in Fig. 1(a).

Generalizing from the above example, we can express multi-dimensional integrals of products of several input arrays as a sequence of formulae. Each formula produces some intermediate array and the last formula gives the final result. A formula is either:

$$\begin{aligned} T1(j,t) &= \sum_i A(i,j,t) \\ T2(j,t) &= \sum_k B(j,k,t) \\ T3(j,t) &= T1(j,t) \times T2(j,t) \\ S(t) &= \sum_j T3(j,t) \end{aligned}$$

(a) Formula sequence



(b) Binary tree representation

Figure 1: A formula sequence and its binary tree representation.

- a multiplication formula of the form: $Tr(\ldots) = X(\ldots) \times Y(\ldots)$, or
- a summation formula of the form: $Tr(\ldots) = \sum_i X(\ldots)$,

where the terms on the right hand side represent input arrays or intermediate arrays produced by a previously defined formula. Let IX, IY and ITr be the sets of indices in $X(\ldots)$, $Y(\ldots)$ and $Tr(\ldots)$, respectively. For a formula to be well-formed, every index in $X(\ldots)$ and $Y(\ldots)$, except the summation index in the second form, must appear in $Tr(\ldots)$. Thus $IX \cup IY \subseteq ITr$ for any multiplication formula, and $IX - \{i\} \subseteq ITr$ for any summation formula. Such a sequence of formulae fully specifies the multiplications and additions to be performed in computing the final result.

A sequence of formulae can be represented graphically as a binary tree to show the hierarchical structure of the computation more clearly. In the binary tree, the leaves are the input arrays and each internal node corresponds to a formula, with the last formula at the root. An internal node may either be a multiplication node or a summation node. A multiplication node corresponds to a multiplication formula and has two children which are the terms being multiplied together. A summation node corresponds to a summation formula and has only one child, representing the term on which summation is performed. As an example, the binary tree in Fig. 1(b) represents the formula sequence shown in Fig. 1(a).

The operation-minimization procedure discussed above usually results in the creation of intermediate temporary arrays. Sometimes these intermediate arrays that help in reducing the computational cost create a problem with the memory capacity required. For example, consider the following expression:

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

If this expression is directly translated to code (with ten nested loops, for indices $a-l$), the total number of arithmetic operations required will be $4N^{10}$ if the range of each index $a-l$ is N. Instead, the same expression can be rewritten by use of associative and distributive laws as the following:

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

This corresponds to the formula sequence shown in Fig. 2(a) and can be directly translated into code as shown in Fig. 2(b). This form only requires $6N^6$ 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.

We have previously shown that the problem of determining the operator tree with minimal operation count is NP-complete, and have developed a pruning search procedure [13] that is very efficient in practice. For the above example, although the latter form is far more economical in terms of the number of arithmetic operations, its implementation will require the use of temporary intermediate arrays to hold the partial results of the parenthesized array subexpressions. Sometimes, the sizes of intermediate arrays needed for the "operation-minimal" form are too large to even fit on disk.

A systematic way to explore ways of reducing the memory requirement for the computation is to view it in terms of potential loop fusions. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array which is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and thus reduces the memory requirements. For the example considered, the application of fusion is illustrated in Fig. 2(c). By use of loop fusion, for this example it can be seen that $T1$ can actually be reduced to a scalar and $T2$ to a 2-dimensional array, without changing the number of arithmetic operations.

For a computation comprised of a number of nested loops, there will generally be a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. In prior work, we have addressed the problem of finding the choice of fusions for a given operator tree that minimizes the total space required for all arrays after fusion [16, 15, 14].

A data-parallel implementation of the unfused code for computing S_{abij} would involve a sequence of three steps, each corresponding to one of the loops in Fig. 2(b). The communication cost incurred will clearly depend on the way the arrays A, B, C, D, $T1$, $T2$, and S are distributed. We have previously considered the problem of minimization of communication with such computations [16]. However, the issue of memory space requirements was not addressed. In practice, many of the computations of interest in quantum chemistry require impractically large intermediate arrays in the unfused operation-minimal form. Although the collective memory of parallel machines is very large, it is nevertheless insufficient to hold the full intermediate arrays for many computations of interest. Thus, array contraction through loop fusion is essential in the parallel context too. However, it is not satisfactory to first find a communication-minimizing data/computation distribution for the unfused form, and then apply fusion transformations to minimize memory for that parallel form. This is because 1) fusion changes the communication cost, and 2) it may be impossible to find a fused form that fits within available memory, due to constraints imposed by the chosen data distribution on possible fusions. In this paper we address this problem of finding suitable fusion transformations and data/computation partitioning that minimize communication costs, subject to limits on available per-processor memory.

3 Memory-constrained communication minimization

Given a sequence of formulae, we now address the problem of finding the optimal partitioning of arrays and operations among the processors and the loop fusions on each processor in order to minimize inter-processor communication while staying within the available memory in implementing the computation on a message-passing parallel computer. Section 3.1 introduces a two-dimensional logical processor model used to represent the computational space, and presents a generalized Cannon's algorithm for tensor contractions. Section 3.2 discusses the combined effects of loop fusions and array/operation partitioning on communication cost and memory usage. An integrated algorithm for solving this problem is presented in Section 3.3.

3.1 Preliminaries: a generalization of Cannon's algorithm for tensor contractions

Since primitive tensor contractions are essentially generalized multi-dimensional matrix multiplications, we choose to use the memory efficient Cannon algorithm [12] as the primary template. The generalization of Cannon's algorithm to multi-dimensional arrays proceeds as follows. A logical view of the P processors as a two-dimensional $\sqrt{P} \times \sqrt{P}$ grid is used, and each array is fully distributed along the two processor dimensions. As will be clear later on, the logical view of the processor grid does not impose any restriction on the actual physical interconnection topology of the processor system, since an empirical characterization of the cost of redistribution between different distributions is performed on the target system.

We use a pair of indices to denote the partitioning or *distribution* of the elements of a data array on a two-dimensional processor array. The d-th position in a pair α, denoted $\alpha[d]$, where d can be either 1 or 2, corresponds to the d-th processor dimension. Each position is an index variable distributed along that processor dimension. As an example, suppose 16 processors form a two-dimensional 4×4 logical array. For the array $B(b,e,f,l)$ in Fig. 2(a), the pair $\alpha = \langle b, f \rangle$ specifies that the first (b) and the third (f) dimensions of B are distributed along the first and second processor dimensions respectively, and that the second (e) and fourth (l) dimensions of B are not distributed. Thus, a processor whose id is P_{z_1,z_2}, with z_1 and z_2 between 1 and 4, will be assigned a portion of B specified by $B(myrange(z_1,N_b,4), 1:N_e, myrange(z_2,N_f,4), 1:N_l)$, where $myrange(z,N,p)$ is the range $(z-1) \times N/p + 1$ to $z \times N/p$.

We assume a data-parallel programming model, with parallelism being exploited within each operator of an operator

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

(a) Formula sequence

```
T1=0; T2=0; S=0
for b, c, d, e, f, l
[ T1bcdf += Befl 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 += Befl Dcdel
    for j, k
      T2fjk += T1f Cdfjk
  for a, i, j, k
    Sabij += T2fjk Aacik
```

(c) Memory-reduced implementation (fused)

Figure 2: Example illustrating use of loop fusion for memory reduction.

tree, and do not consider distributing the computation of different formulae on different subsets of processors. A tensor contraction formula can be expressed as a generalized matrix multiplication C(I,J) += A(I,K) * B(K,J), where I, J, and K represent index collections or index sets. This observation follows from a special property of tensor contractions: all the indices appearing on the left-hand side must appear on the right-hand side only *once* (index sets I and J, for A and B, respectively), and all summation indices must appear on both right-hand side arrays (index set K). For example, the tensor contraction $T1(b,c,d,f) = \sum_{e,l} B(b,e,f,l) \times D(c,d,e,l)$ is characterized by the index sets I = $\{b,f\}$, J = $\{c,d\}$, and K = $\{e,l\}$.

We generalize Cannon's algorithm for multi-dimensional arrays as follows. A triplet $\{i,j,k\}$ formed by one index from each index set I, J, and K defines a distribution $\langle i,j\rangle$ for the result array C, and distributions $\langle i,k\rangle$ and $\langle k,j\rangle$ for the input arrays A and B, respectively. In addition, one of the 3 indices $\{i,j,k\}$ is chosen as the "rotation index," along which the processor communication takes place. For example, in the traditional Cannon algorithm for matrix multiplication, the summation index k plays that role; blocks of the input arrays A and B are rotated among processors, and each processor holds a different block of A and B and the same block of C after each rotation step. At every step, processors multiply their local blocks of A and B, and add the result to the block of C.

Due to the symmetry of the problem, any of the 3 indices $\{i,j,k\}$ can be chosen as the rotation index, so that it is possible to keep any one of the arrays in a fixed distribution and communicate ("rotate") the other two arrays. Therefore, the number of distinct communication patterns within the generalized Cannon's algorithm framework is given by 3× NI NJ NK, where NI is defined as the number of indices in the index set I. The communication costs of the tensor contraction depend on the distribution choice $\{i,j,k\}$ and the choice of rotation index.

In addition to the communication of array blocks during the rotation phase of the Cannon algorithm, array redistribution may be necessary between the Cannon steps. For instance, suppose the arrays $B(b,e,f,l)$ and $D(c,d,e,l)$ have initial distributions $\langle b,f\rangle$ and $\langle e,c\rangle$ respectively. If we want $T1$ to have the distribution $\langle b,c\rangle$ when evaluating $T1(b,c,d,f) = \sum_{e,l} B(b,e,f,l) \times D(c,d,e,l)$, B would have, for example, to be re-distributed from $\langle b,f\rangle$ to $\langle b,e\rangle$ for the generalized Cannon algorithm to be possible. But since the initial distribution $\langle e,c\rangle$ of $D(c,d,e,l)$ is the same as the distribution required to perform the Cannon rotations, no redistribution is necessary for array D.

3.2 Interaction between data distribution and loop fusion

The partitioning of data arrays among the processors and the fusions of loops on each processor are inter-related. Although in our context there are no constraints on loop fusion due to data dependences (there are never any fusion-preventing dependences), there are constraints and interactions with array distribution: (*i*) both affect memory usage, by fully collapsing array dimensions (fusion) or by reducing them (distribution), (*ii*) loop fusion can increase both the number of messages, and therefore the start-up communication cost, and the communication volume, and (*iii*) fusion and communication patterns may conflict, resulting in mutual constraints. We discuss these issues next.

(*i*) **Memory usage and array distribution.** The memory requirements of the computation depend on both loop fusion and array distribution. Fusing a loop with index t between a node v and its parent eliminates the t-dimension of array v. If the t-loop is not fused but the t-dimension of array v is distributed along the d-th processor dimension, then the range of the t-dimension of array v on each processor is reduced to $N_t/\sqrt{P}$. Let $DistSize(v,\alpha,f)$ be the size on each processor of array v, with distribution α and fusion f with its parent, where f is described by a set of fused indices. We have

$$DistSize(v,\alpha,f) = \prod_{i \in v.dimens} DistRange(i,v,\alpha,f)$$

where $v.dimens = v.indices - \{v.sumindex\}$ is the array dimension indices of v before loop fusions, $v.indices$ is the set of loop indices for v including the summation index $v.sumindex$ if v is a summation node, and

$$DistRange(i,v,\alpha,x) = \begin{cases} 1 & \text{if } i \in x \\ N_i/\sqrt{P} & \text{if } i \notin x \text{ and } i = \alpha[d] \\ N_i & \text{if } i \notin x \text{ and } i \notin \alpha \end{cases}$$

In our example, assume $P = 16$ and that $N_a = N_b = N_c = N_d = 480$, $N_e = N_f = 64$, and $N_j = N_k = N_l = 32$. If the array $T1(b,c,d,f)$ has distribution $\langle b,f\rangle$ and fusion $\langle c\rangle$ with $T2$, then the size of B on each processor would be $N_b/\sqrt{P} \times 1 \times N_d \times N_f/\sqrt{P} = 921,600$ words, or 7.2MB per processor.

(*ii*) **Loop fusion increases communication cost.** The distribution of an array v determines the communication pattern, while loop fusions change the number of times array v is communicated and the size of each message. Let v be an array that is communicated (rotated) in a generalized Cannon algorithm tensor contraction. If node v is not fused with its parent, array v is fully rotated only once, in $\sqrt{P}$ rotation steps. Fusing a loop with index t between node v and its parent puts the collective communication code inside the t loop. Thus, the number of communication steps is increased by a factor of $N_t/\sqrt{P}$ if the t-dimension of v is distributed, and by a factor of N_t if the t-dimension of v is not distributed.

(iii) **Potential conflict between array distribution and loop fusion.** For the fusion of a loop between nodes u and v to be possible, its index must either be undistributed at both u and v, or be distributed onto the same number of processors at u and at v. Otherwise, the range of the loop at node u would be different from that at node v, preventing fusion of the loops. This condition introduces an important mutual constraint in the search for the optimal fusion and distribution. It shows that independently finding the optimal fusion configuration and the optimal data distribution pattern may not result in a valid overall solution to the problem of memory-constrained communication minimization. The next section presents an integrated algorithm that provides the solution to this problem.

3.3 Memory-constrained communication minimization algorithm

In this section, we present an algorithm addressing the communication minimization problem with memory constraint. In practice, the arrays involved are often too large to fit into the available memory even after partitioning among the processors. We assume the input arrays can be distributed initially among the processors in any way at zero cost. We do not require the final results to be distributed in any particular way. Our approach works regardless of whether any initial or final data distribution is given.

The main idea of this method is to search among all combinations of loop fusions and array distributions to find one that has minimal total communication and computational cost and uses no more than the available memory. In this section, we present a dynamic programming algorithm for this purpose.

Let $RCost(localsize, \alpha, i)$ be the communication cost in rotating the blocks of an α- distributed array, with *localsize* elements distributed on each processor, along the i index of the array. We empirically measure *RCost* for each distribution α and each position of the index i, and for several different *localsizes* on the target parallel computer. We generate this data by measuring the communication times for different array distributions and array sizes on the target computer (in our case, an Intel Itanium cluster). Although generating the characterization is somewhat laborious, once a characterization file is completed, it can be used to predict, by interpolation or extrapolation, the communication times for arbitrary array distributions and sizes. Let $RotateCost(v, \alpha, i, f)$ denote the communication cost for the array v, which has fusion f with its parent, distribution α, and is rotated along the i index. It can be expressed as:

$$RotateCost(v, \alpha, i, f) = \\ MsgFactor(v, \alpha, f) \times RCost(DistSize(v, \alpha, f), \alpha, i),$$

where

$$MsgFactor(v, \alpha, f) = \prod_{j \in v.dimens} LoopRange(j, v, \alpha, f)$$

and

$$LoopRange(j, v, \alpha, f) = \begin{cases} 1 & \text{if } j \notin f \\ N_j/\sqrt{P} & \text{if } j \in f \text{ and } j = \alpha[d] \\ N_j & \text{if } j \in f \text{ and } j \notin \alpha \end{cases}$$

Finally, we define $Cost(v, \alpha)$ to be the total cost for the subtree rooted at v with distribution α. After transforming the given sequence of formulae into an expression tree T (see Section 2), we initialize $Cost(v, \alpha) = 0$ for each leaf node v in T and each distribution α. For each internal node u and each distribution α, we can calculate $Cost(u, \alpha)$ according to the following procedure: let u be a node with two children v and v'. Let β and γ be the distributions of v and v', respectively. In order to perform the generalized Cannon matrix multiplication, the distributions β, γ, and α have to be compatible, *i.e.* of the form $\beta = \langle i, k \rangle$, $\gamma = \langle k, j \rangle$, and $\alpha = \langle i, j \rangle$. Thus,

$$Cost(u, \alpha) = \min_{\beta, \gamma, l}\{Cost(v, \beta) + Cost(v', \gamma) + RotateCost(u, \alpha, l, \emptyset)\}.$$

With these definitions, the bottom-up dynamic programming algorithm proceeds as follows. At each node v in the expression tree T, we consider all combinations of array distributions for v and loop fusions between v and its parent. The array size and communication cost are determined according to the equations in sections 3.2 and 3.3. At each node v, a set of solutions is formed. Each solution contains the final distribution of v, the loop nesting at v, the loop fusion between v and its parent, the total communication cost, and the memory usage for the subtree rooted at v. A solution s is said to be inferior to another solution s' if they have the same final distribution, s has less potential fusions with v's parent than s', $s.cost \geq s'.cost$, and the memory usage of s is higher than that of s'. An inferior solution and any solution that uses more memory than available can be pruned. At the root node of T, the only two remaining criteria are the communication cost and the memory usage of the solutions. After pruning the solutions whose memory usage exceeds the memory limit, we pick the solution with the lowest communication cost as the optimal solution for the entire tree.

The algorithm is exhaustive, searching through the entire space of array distributions and loop fusions, and discarding only those partial solutions that cannot become optimal. It always finds an optimal solution if there is one. Although the complexity of the algorithm is exponential in the number of index variables and the number of solutions could in theory grow exponentially with the number of index variables, the number of index variables in practical applications is usually small and there is indication that the pruning is effective in keeping the size of the solution set in each node small.

4 An application example

In this section, we present an application example of the memory-constrained communication minimization algorithm. Consider again the sequence of computations in Fig. 2(a), representative of the multi-dimensional tensor contractions often present in quantum chemistry codes:

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

The sizes of the array dimensions are chosen to be compatible with the dimensions found in typical chemistry problems, where they represent occupied or virtual orbital spaces: $N_j = N_k = N_l = 32$, $N_a = N_b = N_c = N_d = 480$, and $N_e = N_f = 64$. The input arrays A, B, C, and D, and the output array S are assumed to be fully stored in memory, while the intermediate arrays $T1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel}$ and $T2_{bcjk} = \sum_{df} T1_{bcdf} \times C_{dfjk}$ can be partially stored through the use of loop fusion (see Fig. 2(c)).

As an example, we investigate the parallel execution of this calculation on an Intel Itanium cluster with 2 processors per node and 4GB of memory available at each node. We first consider the use of 32 nodes in the computation, for a total of 64 processors and 128GB available physical memory. We then consider the execution of the same calculation on only 8 nodes, resulting in a total of 16 processors and 32GB of available memory. Since different amounts of memory are available under the two scenarios, the communication optimal solutions are different. In particular, we show by this example that memory constraints can lead to counter-intuitive trends in communication costs: for a given problem size, as the number of available nodes decreases, more loop fusions are necessary to keep the problem in the available memory, resulting in higher communication costs.

Tables 1 and 2 present the solutions of the memory-constrained communication minimization algorithm on the Itanium cluster for 64 and 16 processors, respectively.

For the system of 64 processors (32 nodes) on the Itanium cluster, the logical view of the processor space is a two-dimensional 8×8 distribution. For the 16 processor (8 node) case, the logical processor view is a 4×4 distribution. Tables 1 and 2 show the full four-dimensional arrays involved in the computation, their reduced (fused) representations, their initial and final distributions, their memory requirements, and the communication costs involved in the Cannon rotations in the initial and final distributions. The initial distribution is defined as the distribution at the multiplication node where the array is generated, or produced, and the definition applies only to intermediate ($T1$, $T2$) or output (S) arrays. The final distribution is defined as the distribution at the multiplication node at which the array is used or consumed; this definition applies to the input arrays A, B, C, and D, as well as to the intermediate arrays $T1$ and $T2$.

For the 64 processor system, the available 128GB of memory are enough to fully store all the arrays in the computation, and no fusion is necessary. Indeed, Table 1 shows the "reduced arrays" being the same as the full arrays; since the array elements are double precision quantities, the total memory requirements for the sum of all arrays is $\approx$ 65.3GB, or $\approx$ 2.04GB/node, which is within the 4GB available memory limit per node, even allowing for an extra 115.2MB temporary send/receive buffer (the size of the largest message to be transmitted, for the array D). The array $T1$ requires the largest amount of space (55.3GB, or $\approx$ 1.73GB/node), and dominates the memory requirements of the problem. However, the optimal solution does not require communication of $T1$, and each processor needs access only to its own data.

For the optimal solution presented in Table 1, the initial and final distributions of the intermediate arrays are the same; this implies that no array re-distribution is performed between the Cannon matrix multiplication steps. Hence, all the communication costs in the problem result exclusively from the alignment and rotation of the multi-dimensional matrix blocks during the Cannon matrix multiplication steps. For each step, the 2 smaller arrays are communicated: in the first step, $T1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel}$, the arrays B and D are rotated; in the second step, $T2_{bcjk} = \sum_{df} T1_{bcdf} \times C_{dfjk}$, C and $T2$ are rotated, while in the final step, $S_{abij} = \sum_{ck} T2_{bcjk} \times A_{acik}$, any 2 arrays can be rotated for the same cost, and we choose A and $T2$. Table 1 shows the communication costs for these Cannon rotations. A value of 0 means that the array is not communicated during a matrix multiplication step.

The total communication cost is the sum of all the costs in Table 1: 98.0 seconds. This represents only 7.0% of the total 1403.4 second running time of the calculation on the 64 processor system. If memory limits were not an issue, we would expect the fraction of communication time to the total running time to decrease even further for a system with a smaller number of nodes. However, the memory constraints are very important in this class of problems, and their impact on communication costs is very significant, as we show in the next example.

Table 2 presents the solution of the algorithm for a system of 8 nodes (16 processors, 32GB of memory) on the Itanium cluster. In this case, the total memory requirements of the problem, $\approx$ 65.3GB, are larger than the available space, so loop fusion for memory reduction is necessary. This is achieved by fusing the f loop and reducing the array $T1(b,c,d,f)$ (55.3GB, or 6.9GB/node) to $T1(b,c,d)$, (864MB, or 108MB/node). The total memory requirement of the optimal solution is now $\approx$ 10.8GB, or 1.35GB/node, which is within the 4GB/node memory limit, even allowing for an extra 230.4MB temporary send/receive buffer (the size of the largest messages to be transmitted, for the arrays A and $T2$).

For the optimal solution presented in Table 2, the initial and final distributions of the intermediate arrays are again the same, so no array re-distribution is performed between the Cannon matrix multiplication steps. Like in the previous example, the inter-processor communication happens only during the matrix multiplication steps. For the first 2 steps, the arrays that do not contain the fused index f are not communicated, in order to minimize the communication cost: in the first step, $T1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel}$, the arrays $B(b,e,f = \text{constant},l)$ and $T1(b,c,d)$ are rotated for each iteration of f, while $D(c,d,e,l)$ is not communicated. The communication of B and $T1$ results in multiple smaller messages and higher start-up costs; in the second step, $T2_{bcjk} = \sum_{df} T1_{bcdf} \times C_{dfjk}$, $C(d, f = \text{constant}, j,k)$ and $T1(b,c,d)$ are rotated for each iteration of f, while $T2(b,c,j,k)$ is not communicated. In the final step, $S_{abij} = \sum_{ck} T2_{bcjk} \times A_{acik}$, any 2 arrays can be rotated for the same cost, because no fusion is involved, and the arrays have the same size; we again choose to communicate A and $T2$, while keeping S in a fixed distribution. Table 2 shows the communication costs for these Cannon rotations. A value of 0 means that the array is not communicated during a matrix multiplication step.

The total communication cost is the sum of all the costs in Table 2: 1907.8 seconds. This represents 27.3% of the total 6983.8 second running time of the calculation on the 16 processor system. This represents a significantly higher percentage than that for the 64 processor system, and it is entirely due to the memory constraints of the problem. In this example, the higher communication overhead for the 16 processor example mostly arises from the rotation of the array $T1$ for each iteration of the f loop.

5 Related work

Much work has been done on improving locality and parallelism by using loop fusion. Kennedy and McKinley [10] presented an algorithm for fusing a collection of loops to minimize the parallel loop synchronization overhead and maximize parallelism. They proved that finding loop fusions that maximize locality is NP-hard. Two polynomial-time algorithms for improving locality were given. Darte [5] discusses the complexity of maximal fusion of parallel loops. Recently,

Table 1: Loop fusions, memory requirements and communication costs on 64 processors (32 nodes) of an Intel Itanium cluster for the arrays presented in Fig. 2(a).

Full array	Reduced array	Initial dist.	Final dist.	Mem./node	Comm. (init.)	Comm. (final)
$D(c,d,e,l)$	$D(c,d,e,l)$	N/A	$\langle d,e\rangle$	115.2MB	N/A	35.7 sec.
$B(b,e,f,l)$	$B(b,e,f,l)$	N/A	$\langle e,b\rangle$	15.4MB	N/A	4.9 sec.
$C(d,f,j,k)$	$C(d,f,j,k)$	N/A	$\langle k,d\rangle$	7.7MB	N/A	2.8 sec.
$A(a,c,i,k)$	$A(a,c,i,k)$	N/A	$\langle a,k\rangle$	57.6MB	N/A	18.3 sec.
$T1(b,c,d,f)$	$T1(b,c,d,f)$	$\langle d,b\rangle$	$\langle d,b\rangle$	1.728GB	0	0
$T2(b,c,j,k)$	$T2(b,c,j,k)$	$\langle k,b\rangle$	$\langle k,b\rangle$	57.6MB	17.8 sec.	18.5 sec.
$S(a,b,i,j)$	$S(a,b,i,j)$	$\langle a,b\rangle$	N/A	57.6MB	0	N/A

Table 2: Loop fusions, memory requirements and communication costs on 16 processors (8 nodes) of an Intel Itanium cluster for the arrays presented in Fig. 2(a).

Full array	Reduced array	Initial dist.	Final dist.	Mem./node	Comm. (init.)	Comm. (final)
$D(c,d,e,l)$	$D(c,d,e,l)$	N/A	$\langle d,e\rangle$	460.8MB	N/A	0
$B(b,e,f,l)$	$B(b,e,f,l)$	N/A	$\langle e,b\rangle$	61.6MB	N/A	25.7 sec.
$C(d,f,j,k)$	$C(d,f,j,k)$	N/A	$\langle k,d\rangle$	30.8MB	N/A	20.8 sec.
$A(a,c,i,k)$	$A(a,c,i,k)$	N/A	$\langle a,k\rangle$	230.4MB	N/A	34.6 sec.
$T1(b,c,d,f)$	$T1(b,c,d)$	$\langle d,b\rangle$	$\langle d,b\rangle$	108.0MB	902.0 sec.	888.5 sec.
$T2(b,c,j,k)$	$T2(b,c,j,k)$	$\langle k,b\rangle$	$\langle k,b\rangle$	230.4MB	0	36.2 sec.
$S(a,b,i,j)$	$S(a,b,i,j)$	$\langle a,b\rangle$	N/A	230.4MB	0	N/A

Kennedy [11] developed a fast algorithm that allows accurate modeling of data sharing as well as the use of fusion-enabling transformations. Ding [6] illustrates the use of loop fusion in reducing storage requirements through an example, but does not provide a general solution. Singhai and McKinley [24] examined the effects of loop fusion on data locality and parallelism together. They viewed the optimization problem as one of partitioning a weighted directed acyclic graph in which the nodes represent loops and the weights on edges represent the amount of locality and parallelism. Although the problem is NP-hard, they were able to find optimal solutions in restricted cases and heuristic solutions for the general case. However, the work addressed in this paper considers a different use of loop fusion, which is to reduce array sizes and memory usage of automatically synthesized code containing nested loop structures. Traditional compiler research does not address this use of loop fusion because this problem does not arise with manually-produced programs.

Gao et al. [8] studied the contraction of arrays into scalars through loop fusion as a means to reduce array access overhead. They partitioned a collection of loop nests into fusible clusters using a max-flow min-cut algorithm, taking into account the data dependencies. However, their study is motivated by data locality enhancement and not memory reduction. Also, they only considered fusions of conformable loop nests, i.e., loop nests that contain exactly the same set of loops. Song et al. [25] have explored the use of loop fusion for memory reduction for sequential execution. They do not consider trading off memory for recomputation or the impact of data distribution on communication costs while meeting per-processor memory constraints in a distributed memory machine.

Loop fusion in the context of delayed evaluation of array expressions in compiling APL programs has been discussed by Guibas and Wyatt [7]. They considered loop fusion without any loop reordering; and their work is not aimed at minimizing array sizes. Lewis et al. [18] discusses the application of fusion directly to array statements in languages such as F90 and ZPL. Callahan et al. [1] present a technique to convert array references to scalar accesses in innermost loops.

As mentioned earlier, loop fusion has also been used as a means of improving data locality [11, 24, 22, 21]. There has been much less work investigating the use of loop fusion as a means of reducing memory requirements [8, 23]. Another significant way in which our approach differs from other work that we are aware of, is that we attempt global optimization across a collection of loop nests using empirically derived cost models.

6 Conclusion

In this paper, we have addressed a compile-time optimization problem arising in the context of a program synthesis system. The goal of the synthesis system is the facilitation of rapid development of high-performance parallel programs for a class of computations encountered in computational chemistry. These computations are expressible as a set of tensor contractions and arise in electronic structure calculations.

We have described the interactions between distributing arrays on a parallel machine and minimizing memory through loop fusion. We have presented an optimization approach that can serve as the basis for a key component of the system, for minimizing the communication cost on a parallel computer under memory constraints. We have found that the memory constraints of a given problem size generally lead to higher communication costs for systems with lower available memory. The effectiveness of the algorithm was demonstrated by applying it to a computation that is representative of those used in quantum chemistry codes such as NWChem.

References

[1] D. Callahan, S.Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation,* White Plains, NY, June 1990.

[2] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. *Proc. of the Intl. Conf. on High Performance Computing*, Lecture Notes in Computer Science, Vol. 2228, pp. 237–248, Springer-Verlag, 2001.

[3] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[4] T. D. Crawford and H. F. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In *Reviews in Computational Chemistry*, vol. 14, pp. 33–136, Wiley-VCH, 2000.

[5] A. Darte. On the complexity of loop fusion. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'99),* Newport Beach, CA, October 1999.

[6] C. Ding. Improving effective bandwidth through compiler enhancement of global and dynamic cache reuse. Ph.D. Thesis, Rice University, January 2000.

[7] L. Guibas and D. Wyatt. Compilation and delayed evaluation in APL. In *Proc. 5th Annual ACM Symposium on Principles of Programming Languages,* Tucson, Arizona, pp. 1–8, Jan. 1978.

[8] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Languages and Compilers for Parallel Processing,* New Haven, CT, August 1992.

[9] High Performance Computational Chemistry Group. NWChem, A computational chemistry package for parallel computers, Version 3.3, 1999. Pacific Northwest National Laboratory, Richland, WA 99352.

[10] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing,* Portland, OR, pp. 301–320, August 1993.

[11] K. Kennedy. Fast greedy weighted fusion. In *Proc. ACM International Conference on Supercomputing,* Santa Fe, May 2000. Also available as Technical Report CRPC-TR-99789, Center for Research on Parallel Computation (CRPC), Rice University, Houston, TX, 1999.

[12] For a brief description, see V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing,* The Benjamin/Cummings Publishing Company, pp. 171, 1994.

[13] C. Lam, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parallel Processing Letters*, Vol. 7 No. 2, pp. 157–168, 1997.

[14] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *Proc. International Conference on High Performance Computing*, Calcutta, India, December 1999.

[15] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Optimization of memory usage requirement for a class of loops implementing multi-dimensional integrals. In *Languages and Compilers for Parallel Computing,* San Diego, August 1999.

[16] C. Lam. Performance optimization of a class of loops implementing multi-dimensional integrals. Ph.D. Dissertation, Ohio State University, Columbus, August 1999. Also available as Technical Report No. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University.

[17] T. Lee and G. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff (Ed.), *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy,* pages 47–109, 1997.

[18] E. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation,* June 1998.

[19] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. In *Proc. International Conference on Parallel Processing,* pp. II:19–28, Oconomowoc, WI, August 1995.

[20] J. Martin. In *Encyclopedia of Computational Chemistry.* P. Schleyer, P. Schreiner, N. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. Schaefer III (Eds.), Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115–128, 1998.

[21] K. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 9(8):769–787, Aug 1998.

[22] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, July 1996.

[23] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proc. ACM International Conference on Supercomputing,* pages 194–205, Cologne, Germany, June 1991.

[24] S. Singhai and K. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[25] Y. Song, R. Xu, C. Wang and Z. Li. Data locality enhancement by memory reduction. In *Proc. of ACM 15th International Conference on Supercomputing,* pages 50–64, June 2001.