Memory-Constrained Communication Minimization for a Class of Array Computations*

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

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

July 24, 2002

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.

*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.

1. Introduction

The development of high-performance parallel programs for scientific applications is usually very 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 [8], in which many computationally intensive components are expressible as a set of tensor contractions. We are developing a synthesis system that will transform an input specification 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 [17, 22], loop fusion and array contraction for memory space minimization [22, 21], tiling and data locality optimization [4, 5], space-time trade-off optimization [6], and data partitioning for communication minimization [18, 19]. 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 [17] that is very efficient in practice. The operation-minimization procedure results in the creation of intermediate temporary arrays.

Often, these intermediate arrays that help in reducing the computational cost 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 fusions for a given operator tree that minimizes the space required for all intermediate arrays after fusion [21, 20].

We have also previously addressed the problem of communication optimization in the context of the operator trees [18, 19]. An efficient polynomial-time dynamic programming algorithm was developed for the determination of optimal distributions of the various arrays through the evaluation of the operator tree so as to minimize interprocessor communication overhead. However, that model did not consider the effects of loop fusion for memory minimization. As we elaborate later with examples, it is not feasible to simply apply the previously developed loop fusion algorithm and the previous communication minimization algorithm (in either order) to optimize for the parallel context when memory size constraints are severe. For many computations of interest to quantum chemists, the unoptimized form of the computation could require in excess of hundreds of terabytes of memory. Therefore, the following optimization problem is of great interest: given a set of computations expressed as a sequence of tensor contractions (explained later on), an empirically derived measure of the communication cost for a given target computer, and a specified limit on the amount of available memory on each processor, re-structure the computation so as to minimize the total execution time while staying within the available memory. In this paper, we present a framework that we have developed to address this problem. The memory-constrained communication minimization algorithm we develop here will be incorporated into the synthesis system being developed.

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 [23, 26]. 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.

The paper is organized as follows. In the next section, we elaborate on the computational context of interest and the pertinent optimization issues. Section 3 presents our multi-dimensional processor model, 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 [14]. 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:

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

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:

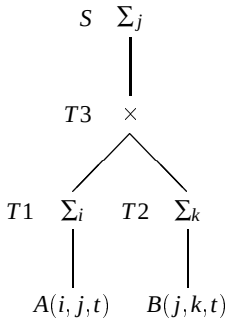
- a multiplication formula of the form: $Tr(\ldots) = X(\ldots) \times Y(\ldots)$, or
- a summation formula of the form: $Tr(\ldots) = \sum_i X(\ldots)$,

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

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

(a) Formula sequence



(b) Binary tree representation

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

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:

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

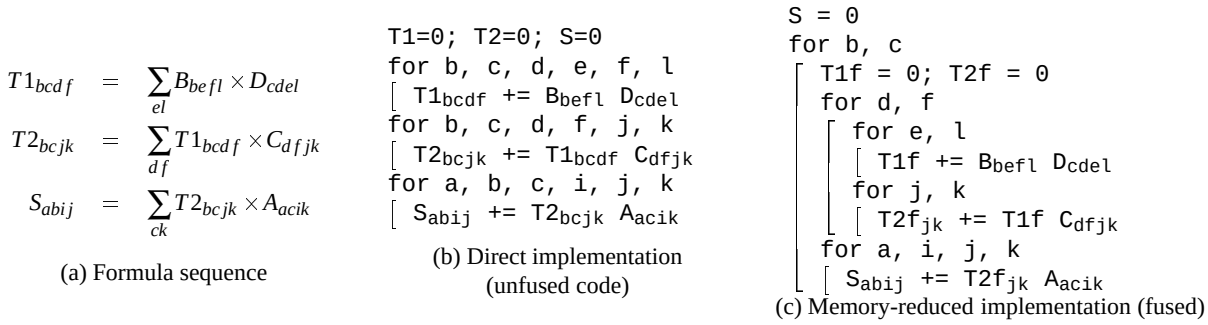
This corresponds to the formula sequence shown in Fig. 2(a) and can be directly translated into code as shown in Fig. 2(b). This form only requires $6N^6$ operations. However, additional space is required to store temporary arrays $T1$ and $T2$. Often, the space requirements for the temporary arrays 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 [17, 18] 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 [22, 21, 20].

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 [22, 18]. However, the issue of memory space requirements was not addressed. In practice, many of the com-



(a) Formula sequence

(b) Direct implementation (unfused code)

(c) Memory-reduced implementation (fused)

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

putations 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 and computational costs while staying within the available memory in implementing the computation on a message-passing parallel computer. Section 3.1 introduces a multi-dimensional processor model used to represent the computational space. Section 3.2 discusses the combined effects of loop fusions and array/operation partitioning on communication cost, computational cost, and memory usage. An integrated algorithm for solving this problem is presented in Section 3.3.

3.1. Preliminaries: A Multi-Dimensional Processor Model

A logical view of the processors as a multi-dimensional grid is used, where each array can be distributed or replicated along one or more of the 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 empirical characterization of the cost of redistribution between different distributions is performed on the target system.

Let p_d be the number of processors on the d-th dimension of an n-dimensional processor array, so that the number of processors is $p_1 \times p_2 \times \ldots \times p_n$. We use an n-tuple to denote the partitioning or *distribution* of the elements of a data array on an n-dimensional processor array. The d-th position in an n-tuple α, denoted $\alpha[d]$, corresponds to the d-th processor dimension. Each position may be one of the following: an index variable distributed along that processor dimension, a '*' denoting replication of data along that processor dimension, or a '1' denoting that only the first processor along that processor dimension is assigned any data. If an index variable appears as an array subscript but not in the n-tuple, then the corresponding dimension of the array is not distributed. Conversely, if an index variable appears in the n-tuple but not in the array, then the data are replicated along the corresponding processor dimension, which is the same as replacing that index variable with a '*'.

As an example, suppose 128 processors form a 4-dimensional $2 \times 2 \times 4 \times 8$ array. For the array $B(b,e,f,l)$ in Fig. 2(a), the 4-tuple $\langle b,e,*,1\rangle$ specifies that the first and the second dimensions of B are distributed along the first and second processor dimensions respectively (the third and fourth dimensions of B are not distributed), and that data are replicated along the third processor dimension and are assigned only to processors whose fourth processor dimension equals 1. Thus, a processor whose id is P_{z_1,z_2,z_3,z_4} will be assigned a portion of B specified

by $B(myrange(z_1,N_b,p_1),myrange(z_2,N_e,p_2),1:N_f,1:N_l)$ if $z_4 = 1$ and no part of B otherwise, where $myrange(z,N,p)$ is the range $(z-1)\times N/p+1$ to $z\times N/p$.

We assume the data-parallel programming model and do not consider distributing the computation of different formulae on different subsets of processors. A child array (or a part of it) is redistributed before the evaluation of its parent if their distributions do not match. For instance, suppose the arrays $B(b,e,f,l)$ and $D(c,d,e,l)$ have distributions $\langle b,e,*,1\rangle$ and $\langle c,d,*,1\rangle$ respectively. If we want $T1$ to have the distribution $\langle c,d,f,1\rangle$ when evaluating $T1(b,c,d,f) = \sum_{e,l} B(b,e,f,l)\times D(c,d,e,l)$, B would have to be redistributed from $\langle b,e,*,1\rangle$ to $\langle *,*,f,1\rangle$ because the two distributions do not match. But since for $D(c,d,e,l)$, the distribution $\langle c,d,*,1\rangle$ is the same as $\langle c,d,f,1\rangle$, D is not redistributed.

3.2. Interaction Between Array Partitioning 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 to 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 does not change the communication volume, but increases the number of messages, and therefore the start-up communication cost, and (iii) fusion and communications 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/p_d. Let $DistSize(v,\alpha,f)$ be the size on each processor of array v, which has fusion f with its parent and distribution α. We have

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

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, $Set(f)$ is the set of fused indices for fusion f, and

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

In our example, assume that $N_a = N_b = N_c = N_d = 1000$, $N_e = N_f = 70$, and $N_j = N_k = N_l = 30$. These are index ranges typical of the quantum chemistry calculations of interest, and are used elsewhere in the paper in relation to this example. If the array $B(b,e,f,l)$ has distribution $\langle b,e,*,1\rangle$ and fusion $\langle bf\rangle$ with T_2, then the size of B on each processor whose fourth dimension equals one would be $N_e/2\times N_l = 1050$ words, since the e and l dimensions are the only unfused dimensions, and the e dimension is distributed onto 2 processors. Note that if array v undergoes redistribution from α to β, the array size on each processor after redistribution is $DistSize(v,\beta,f)$, which could be different from $DistSize(v,\alpha,f)$, the size before redistribution.

(ii) **Loop fusion increases communication cost.** The initial and final distributions of an array v determines the communication pattern and whether v needs redistribution, while loop fusions change the number of times array v is redistributed and the size of each message. Let v be an array that needs to be redistributed. If node v is not fused with its parent, array v is redistributed only once. Fusing a loop with index t between node v and its parent puts the collective communication code for redistribution inside the loop. Thus, the number of redistributions is increased by a factor of N_t/p_d if the t-dimension of v is distributed along the d-th processor dimension and by a factor of N_t if the t-dimension of v is not distributed. In other words, loop fusions cannot reduce communication cost. Instead, the number of messages increases with loop fusion, while the total volume of communication stays the same. Therefore, the communication cost increases, due to higher start-up costs. Consider the computation sequence presented in Fig. 3(a), where the array $C(i,k)$ is first "produced" from $A(i,j)$ and $B(j,k)$, and then "consumed" to produce $E(i,l)$. For this simple example, we assume that the computation is executed in parallel on 4 processors, with a one-dimensional logical processor view. Figure 3(b) shows the pseudo-code in the absence of fusion: the array $C(i,k)$ is re-distributed from $\langle k\rangle$ to $\langle l\rangle$ only once. In the presence of fusion, where the i-loop is the outermost loop, the dimensionality of the array C is reduced to $C(k)$, but the re-distribution is performed N_i times. The pseudo-code in Fig. 3(c) illustrates this effect.

(iii) **Potential conflict between array distribution and loop fusion. Solution of the conflict by virtual partitioning.** For the fusion of a loop between nodes u and v to be possible, the loop 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. Let us consider again the computation given in Figure 3(a), with a different distribution of the array $C(i,k)$ at the two nodes: assume that we have a

$$C(i,k) = \sum_j A(i,j) \times B(j,k)$$
$$E(i,l) = \sum_k C(i,k) \times D(k,l)$$

(a) Formula sequence

```
for i = 1, Ni
  for k = (z-1) * Nk/4 + 1, z * Nk/4
    for j = 1, Nj
      C(i,k) += A(i,j) * B(j,k)
Redistribute C(i,k) from <k> to <l>=<*>
for i = 1, Ni
  for l = (z-1) * Nl/4 + 1, z * Nl/4
    for k = 1, Nk
      E(i,l) += C(i,k) * D(k,l)
```

(b) Before loop fusion

```
for i = 1, Ni
  Initialize C(k) to zero
  for k = (z-1) * Nk/4 + 1, z * Nk/4
    for j = 1, Nj
      C(k) += A(i,j) * B(j,k)
  Redistribute C(k) from <k> to <l>=<*>
  for  l = (z-1) * Nl/4 + 1, z * Nl/4
    for  k = 1, Nk
      E(i,l) += C(k) * D(k,l)
```

(c) After loop fusion

Figure 3: An example of the increase in communication cost due to loop fusion.

```
for i = 1, Ni
  for k = (z-1) * Nk/4 + 1, z * Nk/4
    for j = 1, Nj
      C(i,k) += A(i,j) * B(j,k)
Redistribute C(i,k) from <k> to <i>
for i = (z-1) * Ni/4 + 1, z * Ni/4
  for l = 1, Nl
    for k = 1, Nk
      E(i,l) += C(i,k) * D(k,l)
```

(a) Before virtualization

```
for i = (z-1) * Ni/4 + 1, z * Ni/4
  for ii = 1, 4
    for k = (z-1) * Nk/4 + 1, z * Nk/4
      for j = 1, Nj
        C(ii,k) += A(i + (ii-1) * Ni/4,j) * B(j,k)
  Redistribute C(ii,k) from <k> to <i>=<ii>
  for l = 1, Nl
    for k = 1, Nk
      E(i,l) += C(1,k) * D(k,l)
```

(b) After virtualization

Figure 4: An example of the increase in loop fusion due to a virtual process view.

$\langle k \rangle$ distribution at the first node, and a $\langle i \rangle$ distribution at the second node. The pseudo-code for this computation on 4 processors is presented in Fig. 4(a). Fusion of the *i*-loop is no longer possible, due to the different loop ranges at the two nodes. However, we can overcome this problem by taking a virtualized view of the computation on a larger set of virtual processors, mapped onto the actual physical processors. Consider a virtual partitioning of the computation and split the *i*-loop into two loops, *i* and *ii*. (see the pseudo-code in Fig. 4(b)). With this modification, the outermost *i*-loop can be fused, and the size of the array C is reduced from $N_i \times N_k$ to $4N_k$.

This transformation of the *i*-loop is presented graphically in Fig. 5. At the first node (where it is produced), the array C is distributed among the 4 processors along the k dimension ($\langle k \rangle$ distribution, or vertical partitioning in the Figure). In addition, each physical processor can be further viewed as 4 "virtual processors", as showed by the horizontal virtual partitioning lines in Fig. 5. The purpose of the virtual partitioning along the *i* dimension at the first (produce) node is to match the actual *i* partitioning at the second (consume) node and allow for fusion of the *i*-loop. Fusion of the *i*-loop no longer produces a one-dimensional $C(k)$ array in this case. Each processor stores the equivalent of 4 such arrays, corresponding to the 4 virtual processors. In Fig. 5, the elements stored on processor P_0, before and after re-distribution, are represented by shaded areas.

In general, the virtual partitioning of the computation depends on the distribution at the nodes involved. Let u and v be two nodes in the operator tree T that have a common loop index t. The t-loop is distributed onto p_u processors at node u and onto p_v processors at node v. Let p_{virtual} be lowest common multiple of p_u and p_v. With these notations, the t-loop can be virtually partitioned by a factor of p_{virtual}/p_u at the u node, and by a factor of p_{virtual}/p_v at the v node. The resulting virtual partitions along the t dimension at the u and v nodes become identical, allowing for loop fusion.

Virtual partitioning is essential for the success of our combined loop fusion — data distribution approach. Since both fusion and distribution impose constraints on the array dimensions, the potential for conflict is enormous. In practice, unless we allow virtual partitioning, we often find that optimal array distribution for minimizing inter-processor communication precludes effective memory reduction by fusion. The number of compatible loop fusion and array distribution configurations is very limited. Virtual partitioning relaxes the mutual constraints imposed by the loop fusion and data distribution, allowing for the optimal solution(s) to be found.

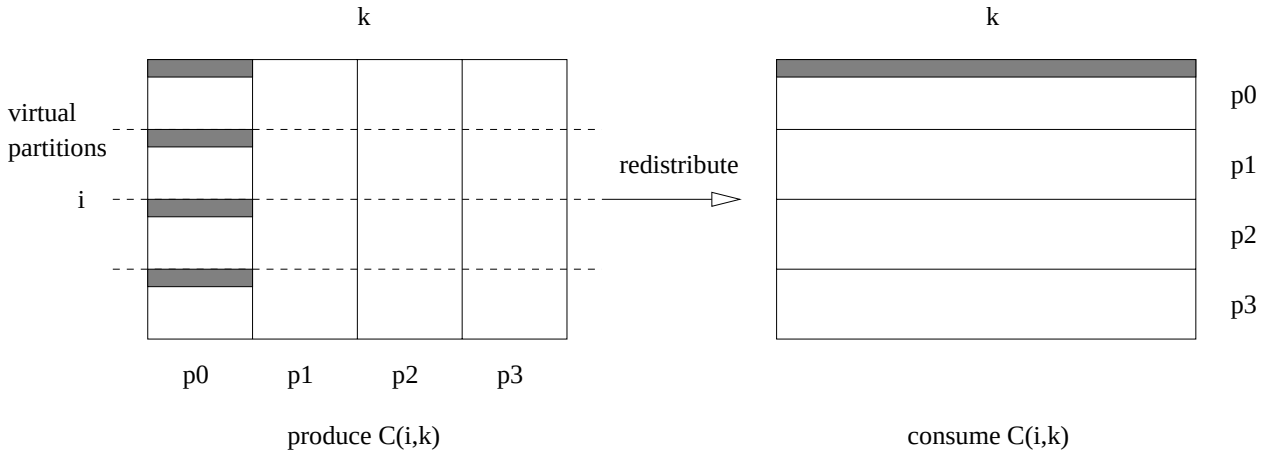


Figure 5: Virtual partitioning of an array.

3.3. Memory-Constrained Communication Minimization Algorithm

In this section, we present an algorithm addressing the communication minimization problem with memory constraint. Previously, we have solved the communication minimization problem but without considering loop fusion or memory usage [18]. 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, as long as they are not replicated. 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. A dynamic programming algorithm for this purpose is given in this section.

Let $Mcost(localsize, \alpha, \beta)$ be the communication cost in moving the elements of an array, with *localsize* elements distributed on each processor, from an initial distribution α to a final distribution β. We empirically measure *Mcost* for each possible non-matching pair of α and β and for several different *localsizes* on the target parallel computer. Let $MoveCost(v, \alpha, \beta, f)$ denote the communication cost in redistributing the elements of array v, which has fusion f with its parent, from an initial distribution α to a final distribution β. It can be expressed as:

$$MoveCost(v, \alpha, \beta, f) = MsgFactor(v, \alpha, Set(f)) \times Mcost(DistSize(v, \alpha, Set(f)), \alpha, \beta),$$

where

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

and

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

Let $CalcCost(v, \gamma)$ be the computational cost in calculating an array v with γ as the distribution of v. Note that the computational cost is unaffected by loop fusions. For multiplication and for summation where the summation index is not distributed, the computational cost for v can be quantified as the total number of operations for v divided by the number of processors working on distinct parts of v. In our example in Fig. 2(a), if the array $T1(b,c,d,f)$ has distribution $\langle c,d,f,1\rangle$, its computational cost would be $N_b \times N_c \times N_d \times N_e \times N_f \times N_l / p_1 / p_2 / p_3 = 9.1875 \times 10^{12}$ multiply-add operations on each participating processor. Formally,

$$CalcCost(v, \gamma) = \frac{\prod_{i \in v.indices} N_i}{\prod_{\gamma[d] \in v.dimens} p_d}$$

For the case of summation where the summation index $i = v.sumindex$ is distributed, partial sums of v are first formed on each processor and then either consolidated on one processor along the i-dimension or replicated on all processors along the same processor dimension. We denote by *CalcCost1* and *MoveCost1* the computational and communication costs for forming the sum without replication, and by *CalcCost2* and *MoveCost2* those with replication.

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 $Cost(v, \alpha)$ for each leaf node v in T and each distribution α as follows:

$$Cost(v,\alpha) = \begin{cases} 0 & \text{if } NoRep(\alpha) \\ \min_{\beta}\{MoveCost(v,\beta,\alpha,\emptyset)\} & \text{otherwise,} \end{cases}$$

where $NoRep(\alpha)$ is a predicate meaning α involves no replication.

For each internal node u and each distribution α, we can calculate $Cost(u,\alpha)$ according to the following procedure:
Case (a): u is a multiplication node with two children v and v'. We need both v and v' to have the same distribution, say γ, before u can be formed. After the multiplication, the product could be redistributed if necessary. Thus,

$$\begin{aligned} Cost(u,\alpha) &= \min_{\gamma}\{Cost(v,\gamma) + Cost(v',\gamma) \\ &+ CalcCost(u,\gamma) + MoveCost(u,\gamma,\alpha,\emptyset)\}. \end{aligned}$$

Case (b): u is a summation node over index i and with a child v, which may have any distribution γ. If $i \in \gamma$, each processor first forms partial sums of u and then we either combine the partial sums on one processor along the i dimension or replicate them on all processors along that processor dimension. Afterwards, the sum could be redistributed if necessary. Thus,

$$\begin{aligned} Cost(u,\alpha) &= \min_{\gamma}\{Cost(v,\gamma) + \min_{j=1,2}(CalcCost_j(u,\gamma) \\ &+ MoveCost_j(u,\gamma,\alpha,\emptyset))\}. \end{aligned}$$

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. If loop fusion of the same index t between v and its parent is not possible because of different distribution ranges, then a virtual processor view is considered in order to allow the fusion. The array size, communication cost, and computational cost are determined according to the equations in Sections 3.1 and 3.3. If the size of an array before and after redistribution is different, the higher of the two should be used in determining memory usage. 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 and computational 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', $s.totalcost \geq s'.totalcost$, 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 total cost and the memory usage of the solutions. The set of solutions is ordered in increasing memory usage and decreasing cost. The solution with the lowest total cost and whose memory usage is below the available memory limit is the optimal solution for the entire tree.

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. 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: $N_i = N_j = N_k = N_l = 40$, $N_a = N_b = N_c = N_d = 1000$, and $N_e = N_f = 70$.

As an example, we investigate the parallel execution of this calculation on 32 processors of a Cray T3E, assuming 512MB of memory available at each node, and on 16 processors of an Intel Itanium cluster, assuming 2GB of memory available at each node. The best partitioning of the algorithm depends on the number of processors and the amount of memory available. It also depends on the empirical characterization data that we use to describe the communication costs of a given machine. We generated this data by measuring the communication times for each possible non-matching pair of array distributions and different array sizes for both the Cray T3E and the 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.

Tables 1 and 2 present the solutions of the memory-constrained communication minimization algorithm on the Cray T3E and Itanium cluster, respectively.

For the system of 32 processors of the Cray T3E, the optimal logical view of the processor space is found to be a two-dimensional 4×8 distribution. Table 1 shows 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 their re-distribution. The final distribution is defined in the same way for both input and intermediate arrays: it is the distribution at the multiplication node at which the array is used or consumed. The initial distribution is defined differently for input and intermediate arrays: it is the distribution at the leaf node for an input array, and the distribution at the multiplication node where the array is generated, or produced, for

Full array	Reduced array	Initial dist.	Final dist.	Memory/processor	Comm. cost
$D(c,d,e,l)$	$D(c,e,l)$	$\langle c,e\rangle$	$\langle *,*\rangle$	22.4MB	552.8 sec.
$B(b,e,f,l)$	$B(b,e,f,l)$	$\langle b,f\rangle$	$\langle b,f\rangle$	49.0MB	0
$C(d,f,j,k)$	$C(f,j,k)$	$\langle j,f\rangle$	$\langle *,*\rangle$	0.9MB	362.3 sec.
$A(a,c,i,k)$	$A(c,i,k)$	$\langle i,c\rangle$	$\langle *,*\rangle$	12.8MB	460.9 sec.
$T1(b,c,d,f)$	$T1(b,c,f)$	$\langle b,f\rangle$	$\langle b,c\rangle$	17.5MB	791.8 sec.
$T2(b,c,j,k)$	$T2(b,c,j,k)$	$\langle b,c\rangle$	$\langle b,j\rangle$	400.0MB	20.5 sec.
$S(a,b,i,j)$	$S(b,i,j)$	$\langle b,j\rangle$	$\langle b,j\rangle$	0.4MB	0

Table 1: Loop fusions, memory requirements and communication costs on 32 processors of a Cray T3E for the arrays presented in Fig. 2(a).

an intermediate array. The total memory requirement of an array is defined as the largest memory usage of the two distributions (initial and final).

The optimal solution has the a and d loops fused, each across its own range: the fusion of the d-loop reduces C, D, and $T1$ to three-dimensional arrays, while the fusion of the a-loop reduces A and S to 3-dimensional arrays as well. Notice that B and $T2$ are the only four-dimensional arrays left, and, consequently, they have the largest storage requirements of all arrays: 49MB per processor and 400MB per processor, respectively. The total memory requirements for the solution of the example are 503MB per processor, within the imposed limit of 512MB. Notice that further memory reduction is possible, for example, by partially fusing the c-loop and collapsing D and $T1$ to two-dimensional arrays. However, this is unnecessary, as the communication cost of the computation would increase, and nothing can be gained by further memory reduction.

Based on the empirical characterization data of the Cray T3E, the total communication cost for this example is 2188 seconds, or 0.61 hours. Most of this load can be attributed to the re-distribution of the arrays A, C, D, and $T1$. Since they are collapsed onto three dimensions for better memory management, they have to be partially re-distributed at each iteration of the fused loop, resulting in large message-passing start-up costs.

Table 2 presents the solution of the algorithm for a system of 16 processors on the Itanium cluster. The optimal logical view of the processor space is found to be a two-dimensional 4×4 distribution. The total memory requirement of the optimal solution is 1.77GB per processor, which is within the 2GB memory limit. The total communication cost is 3076 seconds, or 0.85 hours. The optimal distributions of the arrays are different for the two cases presented here (see Tables 1 and 2).

It is important to note that a decoupled approach of first performing loop fusion followed by array distribution fails to provide a feasible solution in this example. In particular, minimizing the communication cost without taking memory usage into account produces a final distribution $\langle a,b\rangle = \langle *,*\rangle$ for the array $T2(b,c,j,k)$. The array $T2$ would be replicated on all processors, resulting in a memory usage of 12.8GB per processor. Reduction from this amount is possible by fusion, but the constraints imposed by the communication-optimal solution do not permit effective memory reduction. In this example, starting from the unfused communication-optimal solution, no loop fusion structure exists that can bring the memory usage under the limit. Only an integrated approach to memory reduction and communication minimization is able to provide a solution.

5. Related Work

Much work has been done on improving locality and parallelism by using loop fusion. Kennedy and McKinley [15] 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 [9] discusses the complexity of maximal fusion of parallel loops. Recently, Kennedy [16] has developed a fast algorithm that allows accurate modeling of data sharing as well as the use of fusion-enabling transformations. Ding [10] illustrates the use of loop fusion in reducing storage requirements through an example, but does not provide a general solution. Singhai and McKinley [32] 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.

Full array	Reduced array	Initial dist.	Final dist.	Memory /processor	Comm. cost
$D(c,d,e,l)$	$D(c,e,l)$	$\langle e,l\rangle$	$\langle *,*\rangle$	22.4MB	704.8 sec.
$B(b,e,f,l)$	$B(b,e,f,l)$	$\langle f,b\rangle$	$\langle f,b\rangle$	98.0MB	0
$C(d,f,j,k)$	$C(f,j,k)$	$\langle j,f\rangle$	$\langle *,*\rangle$	0.9MB	389.7 sec.
$A(a,c,i,k)$	$A(c,i,k)$	$\langle c,k\rangle$	$\langle *,*\rangle$	12.8MB	546.0 sec.
$T1(b,c,d,f)$	$T1(b,c,f)$	$\langle f,b\rangle$	$\langle c,b\rangle$	35.0MB	1391.7 sec.
$T2(b,c,j,k)$	$T2(b,c,j,k)$	$\langle c,b\rangle$	$\langle j,b\rangle$	800.0MB	43.9 sec.
$S(a,b,i,j)$	$S(a,b,i,j)$	$\langle j,b\rangle$	$\langle j,b\rangle$	800.0MB	0

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

Traditional compiler research does not address this use of loop fusion because this problem does not arise with manually-produced programs.

Gao et al. [13] 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.

Loop fusion in the context of delayed evaluation of array expressions in compiling APL programs has been discussed by Guibas and Wyatt [12]. As part of their algorithm, a general buffering mechanism is devised to save portions of a sub-expression that will be repeatedly needed, to avoid re-computation. They considered loop fusion without any loop reordering; and their work is not aimed at minimizing array sizes. Lewis et al. [24] discusses the application of fusion directly to array statements in languages such as F90 and ZPL. Callahan et al. [3] present a technique to convert array references to scalar accesses in innermost loops.

There has been some recent work on using loop fusion for memory reduction for sequential execution. Fraboulet et al. [11] use loop alignment to reduce memory requirement between adjacent loops by formulating the one-dimensional version of the problem as a network flow problem; they did look at the effect of their solution on cache behavior or communication. Song [35] and Song et al. [36, 37] present a different network flow formulation of the memory reduction problem and they include a simple model of cache misses as well. 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 tiling for enhancing data locality has been studied extensively [1, 2, 7, 29, 30, 34, 38, 39, 40]. As mentioned earlier, loop fusion has also been used as a means of improving data locality [16, 32, 33, 28, 27]. There has been much less work investigating the use of loop fusion as a means of reducing memory requirements [13, 31]. To the best of our knowledge, loop fusion transformation for memory reduction, in combination with data partitioning for communication minimization in the parallel context, has not been previously considered.

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. 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] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. In *Proc. ACM International Conference on Supercomputing*, Santa Fe, NM, 2000.

[2] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 166–178, Santa Barbara, CA, July 1995.

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

[4] D. Cociorva, J. Wilkins, C. Lam, G. Baumgartner, P. Sadayappan, J. Ramanujam. Loop Optimizations for a Class of Memory-Constrained Computations. In *Proc. 15th ACM Intl. Conf. on Supercomputing*, 2001.

[5] 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.

[6] 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. *Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002, To Appear.

[7] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[8] 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.

[9] 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.

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

[11] A. Fraboulet, G. Huard and A. Mignotte. Loop alignment for memory access optimization. In *Proc. 12th International Symposium on System Synthesis*, pages 71–77, San Jose, California, November 1999.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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, Kluwer Academic, 1997.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proc. 8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, March 1999.

[30] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Proc. SC'00*, Dallas, TX, November 2000.

[31] 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.

[32] S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Proc. Mid-Atlantic Student Workshop on Programming Languages and Systems*, SUNY at New Paltz, April 1996.

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

[34] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, Georgia, May 1–4, 1999.

[35] Y. Song. *Compiler algorithms for efficient use of memory systems.* Ph.D. thesis, Department of Computer Sciences, Purdue University, November 2000.

[36] 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.

[37] Y. Song, C. Wang and Z. Li. Locality enhancement by array contraction. In *Proc. 14th International Workshop on Languages and Compilers for Parallel Computing*, August 2001.

[38] M. Wolf and M. Lam. A data locality optimization algorithm. In *SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[39] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proc. 3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[40] M. Wolfe. *High performance compilers for parallel computing.* Addison Wesley, 1996.