Efficient Synthesis of Out-of-Core Algorithms Using a Nonlinear Optimization Solver*

Sandhya Krishnan[1] Sriram Krishnamoorthy[1]
Gerald Baumgartner[1] Chi-Chung Lam[1]
J. Ramanujam[2] P. Sadayappan[1]
Venkatesh Choppella[3]

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

[2]Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803, USA.
jxr@ece.lsu.edu

[3]Indian Institute of Information Technology and Management, Kerala
Technopark, Thiruvananthapuram, Kerala 695 581, INDIA.
choppell@iiitmk.ac.in

Abstract

We address the problem of efficient out-of-core code generation for a special class of imperfectly nested loops encoding tensor contractions. These loops operate on arrays too large to fit in physical memory. The problem involves determining optimal tiling and placement of disk I/O statements. This entails a search in an explosively large parameter space. We formulate the problem as a non-linear optimization problem and use a discrete constraint solver to generate optimized out-of-core code. Measurements on sequential and parallel versions of the generated code demonstrate the effectiveness of the proposed approach.

1 Introduction

Many scientific and engineering applications need to operate on data sets that are too large to fit in the physical memory of the machine. The application we consider is the implementation of electronic structure calculations in quantum chemistry [41, 18]. These calculations employ multi-dimensional tensors involved in contractions (generalized matrix multiplications) that are often much larger than available physical memory. In such situations, it is necessary to develop so-called *out-of-core* algorithms that explicitly orchestrate the movement of subsets of data between main memory and secondary disk storage. These algorithms ensure that data is operated in chunks small enough to fit within the system's physical memory but large enough to minimize the cost of moving data between disk and main memory.

The problem addressed in this paper is the following: We are given an imperfectly nested loop structure containing a collection of tensor contraction computations expressed in "abstract" form, that is, without concern for whether the arrays can fit within available physical memory. The problem consists of generating a "concrete" form of the code by suitably tiling the loops and inserting the necessary disk I/O statements so as to minimize the cost of disk I/O. In the case of parallel code generation, it also involves distributing the workload among processors and inserting the requisite communication calls. The search space of possible placements of the disk I/O statements and possible combinations of tile sizes is explosively large. We formulate the problem as a non-linear optimization problem and use a general-purpose discrete constraint solver to generate optimized out-of-core

*Supported in part by the National Science Foundation through the Information Technology Research program (CHE-0121676)

code.

The paper is organized as follows: In Sec. 2, we explain the context in which the data locality optimization is being performed. In Sec. 3, we review related work in the area. In Sec. 4, we describe the Discrete Constrained Search (DCS) solver [43] and outline the steps used to convert the abstract code specification into concrete code. Our experimental results in Sec. 5 demonstrate that the DCS-based approach to out-of-core code generation is orders of magnitude faster that our previous approach [10, 38]. This enables us to deal with more complex higher order coupled cluster methods, for which our previous approach becomes impractical.

2 The Computational Context

The optimization presented in this paper has been implemented as part of the Tensor Contraction Engine (TCE) [13, 10], a domain-specific compiler for ab initio quantum chemistry calculations. The TCE takes as input a high-level specification of a computation expressed as a set of tensor contraction expressions and transforms it into efficient parallel code. Several compile-time optimizations are incorporated into the TCE: algebraic transformations to minimize operation counts [6, 7], loop fusion to reduce memory requirements [3, 5, 4], space-time trade-off optimization [8], communication minimization [11], and data locality optimization [10, 9] of memory-to-cache traffic.

A tensor contraction expression is comprised of a collection of multi-dimensional summations of the product of several input arrays. As an example, consider the following contraction, used often in quantum chemistry calculations to transform a set of two-electron integrals from an atomic orbital (AO) basis to a molecular orbital (MO) basis:

$$B(a,b,c,d) = \sum_{p,q,r,s} C1(s,d) \times C2(r,c) \times C3(q,b) \times C4(p,a) \times A(p,q,r,s)$$

This contraction is referred to as a four-index transform. Here, $A(p,q,r,s)$ is a four-dimensional input array initially stored on disk, and $B(a,b,c,d)$ is the transformed output array to be placed on disk at the end of the computation. The arrays $C1$ through $C4$ are called transformation matrices. In practice, these four arrays are identical; we identify them by different names only in order to be able to distinguish them in the text.

The indices p, q, r, and s have the same range N, denoting the total number of orbitals, which is equal to $O + V$. O denotes the number of occupied orbitals and V denotes the number of unoccupied (virtual) orbitals. Likewise, the index ranges for a, b, c, and d are the same, and equal to V. Typical values for O range from 10 to 300; the number of virtual orbitals V is usually between 50 and 1000.

The calculation of B is done in four steps to reduce the number of floating point operations from $O(V^4N^4)$ in the initial formula (8 nested loops, for p, q, r, s, a, b, c, and d) to $O(VN^4)$:

$$B(a,b,c,d) = \sum_s C1(s,d) \times \left(\sum_r C2(r,c) \times \left(\sum_q C3(q,b) \times \left(\sum_p C4(p,a) \times A(p,q,r,s) \right) \right) \right)$$

This operation-minimization transformation results in the creation of three intermediate arrays:

$$T1(a,q,r,s) = \sum_p C4(p,a) \times A(p,q,r,s)$$

$$T2(a,b,r,s) = \sum_q C3(q,b) \times T1(a,q,r,s)$$

$$T3(a,b,c,s) = \sum_r C2(r,c) \times T2(a,b,r,s)$$

Assuming that the available memory on the machine running this calculation is less than V^4 (which for $V = 800$ and double precision arrays is about $3TB$), none of A, $T1$, $T2$, $T3$, and B can entirely fit in memory. Therefore, the intermediates $T1$, $T2$, and $T3$ need to be written to disk once they are produced, and read from disk before they are used in the next step. Since none of these arrays can be fully stored in memory, it may not be possible to perform all multiplication operations by reading each element of the input arrays from disk only once. This could result in the amount of disk I/O volume being much larger than the total volume of the data on disk.

For illustration purposes, we focus on the following contraction (a two-index transform):

$$B(m,n) = \sum_{i,j} C1(m,i) \times C2(n,j) \times A(i,j)$$

The operation minimal form of the two-index transform and the corresponding intermediate array are as follows:

$$B(m,n) = \sum_i C1(m,i) \times \left(\sum_j C2(n,j) \times A(i,j) \right)$$

$$T(n,i) = \sum_j C2(n,j) \times A(i,j)$$

Fig. 1 shows an abstract form of the computation of the array B and illustrates how memory requirements for the computation of B may be reduced using loop fusion. The computation is abstract because it can be executed only if the sizes of the arrays are small enough to fit in the available physical memory. The transformation of abstract forms into concrete forms that can be executed is addressed later in Sec. 4. Concrete forms have explicit disk I/O statements between disk-resident arrays and their in-memory counterparts. Fig. 1(a) shows the abstract form of the computation before loop fusion. The computation consists of two loop nests: a first loop that produces the intermediate

```
double T(V,N)

T(*,*) = 0
B(*,*) = 0
FOR i = 1, N
   FOR n = 1, V
      FOR j = 1, N
         T(n,i) += C2(n,j) * A(i,j)
END FOR j,n,i

FOR i = 1, N
   FOR n = 1, V
      FOR m = 1, V
         B(m,n) += C1(m,i) * T(n,i)
END FOR m,n,i
```

(a) Unfused code

```
double T(V,N)

T(*,*) = 0
B(*,*) = 0
FOR i, n, j
   T(n,i) += C2(n,j) * A(i,j)
END FOR j,n,i

FOR i, n, m
   B(m,n) += C1(m,i) * T(n,i)
END FOR m,n,i
```

(b) Compact notation

```
double T

B(*,*) = 0
FOR i, n
   T = 0
   FOR j
      T += C2(n,j) * A(i,j)
   END FOR j

   FOR m
      B(m,n) += C1(m,i) * T
   END FOR m
END FOR n,i
```

(c) Fused code

Figure 1. Example of the use of loop fusion to reduce memory requirements. Loops i and n are fused to reduce T to a scalar.

$T(1:V, 1:N)$, and a second loop that uses T to produce the result $B(1:V, 1:V)$.

In Fig. 1(b), each loop nest is abbreviated into a single loop with a sequence of indices. Fig. 1(c) illustrates the result of loop fusion. Note that all loops in each of the two loop nests in Fig. 1(a) are fully permutable and there are no fusion-preventing dependences between the loops. Hence, the common loops i and n, shown bold-faced, can be fused. After loop fusion, the storage requirements for T can be reduced because there is no longer a need for an explicit dimension of T corresponding to any loop indices that are fused between the producer of T and the consumer of T — storage elements can be reused over sequential iterations of fused loops. In this example, T can be contracted to a scalar as shown in Fig. 1(c). Although the total number of arithmetic operations remains unchanged, the significant reduction in size of the intermediate array T implies that it may be completely stored in memory, without the need for any disk I/O for it. In contrast, if $V \times N$ is larger than the available memory, the unfused version will require that T be written out to disk after it is produced in the first loop, and then read in from disk for the second loop.

3 Related Work

The issue arising in the synthesis context described earlier has been previously addressed by us, focusing primarily on minimizing memory-to-cache data movement [9, 10]. In [9], an integrated approach to fusion and tiling transformations was developed for a restricted class of loops arising in the context of our program synthesis system. The algorithm developed in [10] removed the restrictions assumed in [9]. A tile size search procedure was developed to estimate the total capacity miss cost for each of a large number of combinations of tile sizes for the various loops of an imperfectly nested loop set. After finding the best combination of tile sizes, adjustments were made to address spatial locality considerations done — by adjusting the tile sizes for any loop indexing the fastest varying dimension of any array to be larger than the cache line size.

This approach was extended to the disk-memory hierarchy in [38], where a greedy approach to disk read/write placement was taken. For each set of tile sizes, the algorithm places read/write statements immediately inside those loops at which the memory limit is exceeded. In [39], a set of candidate fusion structures with disk I/O placements was taken as input and the tile size search space was explored. The search space was divided into feasible and infeasible solution spaces and their boundary was shown to contain the optimal solution. An algorithm was devloped to locate the boundary efficiently and a steepest ascent hill-climbing used to determine an efficient solution for the tile sizes.

There has been some work in the area of software techniques for optimizing disk I/O. These include parallel file systems, compile time [33, 32, 31, 26, 25, 24, 42, 27] and runtime libraries and optimizations [34, 47]. Bordawekar et al. [33, 31] discuss several compiler methods for optimizing out-of-core programs in High Performance Fortran. Bordawekar et al. [32] develop a scheduling strategy to eliminate additional I/O arising from communication among processors. Solutions for choreographing disk I/O with computation are presented by Paleczny et al. [27]. They organize computations into groups that operate efficiently on data accessed in chunks. Compiler-directed pre-fetching is discussed by Mowry et al. [42]. This is orthogonal to compiler transformations discussed in this paper. ViC* (Virtual C*) [40] is a preprocessor that transforms out-of-core C* programs into in-core programs with appropriate calls to the ViC* I/O library. Kandemir et al. [26, 25, 24] develop file layout and loop transformations for reducing I/O. None of these techniques address model-driven automatic tile size selection for optimizing I/O and all of them address only perfectly nested loops.

Considerable research on loop transformations for locality in nested loops has been reported in the literature [35, 45, 46, 21, 22]. Nevertheless, a performance-model driven approach to the integrated use of loop fusion and loop tiling for enhancing locality in imperfectly nested loops has not been addressed in these works. Loop tiling for enhancing data locality has been studied extensively [17, 35, 16, 15, 14, 22, 23, 48], and analytical models of the impact of tiling on locality in perfectly nested loops have been developed [36, 28, 30]. Ahmed et. al. [29] have developed a framework that embeds an arbitrary collection of loops into an equivalent perfectly nested loop that can be tiled; this allows a cleaner treatment of imperfectly nested loops. Lim et al. [2] developed a framework based on affine partitioning and blocking to reduce synchronization and improve data locality. Specific issues of locality enhancement, I/O optimization and automatic tile size selection have not been addressed in the works that can handle imperfectly nested loops [29, 2, 48].

4 Proposed Approach

This section gives a detailed explanation of the approach proposed for out-of-core code generation. We use the Discrete Constrained Search (DCS) [43, 12] package, a non-linear programming solver, developed at the University of Illinois at Urbana Champaign. The DCS package takes as input a set of unknown variables to be determined, the objective function to be minimized, and a set of constraints to be satisfied. DCS uses AMPL, *A Modeling Language for Mathematical Programming* [1], as the problem input format.

The out-of-core code generation process uses loop tiling and placement of disk I/O statements to convert an abstract code to concrete code. The search space of disk I/O placements and tile sizes is fully explored here. The search is formulated as a non-linear constrained minimization problem where the objective function is the disk I/O cost. The solution to be determined is constrained by the memory limit and minimum I/O block size for efficient disk I/O. The non-linear problem that is formulated is input to the DCS system, which determines the optimal combination of placement of disk I/O statements and tile sizes.

We continue with the two-index transform example introduced in Sec. 2. Fig. 2(a) shows an abstract code for the two-index transform. We assume that the arrays involved are too large to fit in the physical memory of the machine. The arrays involved in the loop structure fall into the following three categories: input arrays that initially reside on disk (A, $C1$ and $C2$), intermediate arrays produced and consumed within the computation and not required on completion (T), and output arrays that must be finally written to disk (B). Fig. 2(b) shows the parse tree corresponding to the abstract code in Fig. 2(a).

The input to the out-of-core code generation algorithm consists of the abstract code, ranges of the loops and memory limit of the machine. The algorithm comprises of the following three steps:

1. **Loop Tiling:** Each loop is split into a tiling loop and an intra tile loop, and the intra tile loops are propagated down to the leaves. For example, as shown in Fig. 3, loop i is split into tiling loop i_T and intra tile loop i_I. Fig. 3(b) shows the tiled parse tree for the tiled abstract code in Fig. 3(a).

2. **Candidate Placements:** For each array, all the possible placements of disk read/write statements are enumerated.

3. **DCS Input Construction:** Given the enumeration from step 2, non-linear equations for the objective function and constraints are constructed and provided as input to the DCS solver. The DCS solver outputs the disk read/write placement for each array and the tile sizes that minimize the disk I/O cost and satisfy the memory limit constraint.

4.1 Candidate Placements

Given a tiled imperfectly nested loop structure (Fig. 3), we consider various possible placements of reads for input arrays, reads and writes for intermediates, and writes (and reads, if required) for output arrays. In enumerating the candidate placements, there are some constraints that must be satisfied.

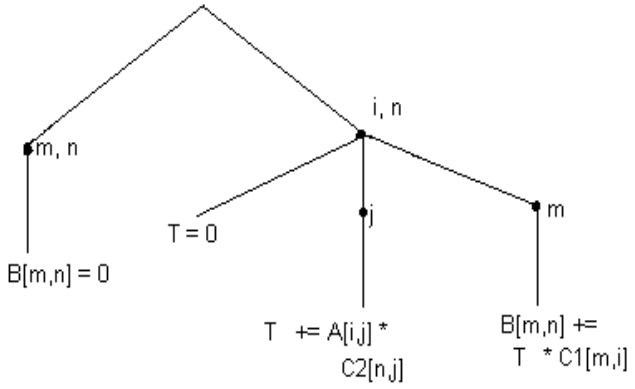
1. **Input Array Constraints:** The read statement for an input array may only be placed to be executed before the statement where it is consumed. For example, in Fig. 3(a), the read for input array A can be placed anywhere before line 7.

2. **Output Array Constraints:** The write for an output array may only be placed after the statement where it is produced. For example, in Fig. 3(a), the write for output array B can be placed anywhere after line 9.

3. **Intermediate Array Constraints:** For intermediate arrays, we have two cases to consider: the array is either kept in memory or written to disk. If the array is kept in memory, there will be no disk I/O statements inserted for the array. On the other hand, if it is written to disk, there is a constraint imposed on its disk read/write placement. For example, in Fig. 3(a), intermediate T is produced in statement 7 and consumed in statement 9. If we consider these statements in the parse tree in Fig. 3(b), the lowest common ancestor for

```
FOR m, n
  B[m,n] = 0
FOR i, n
  T = 0
  FOR j
    T += A[i,j] * C2[n,j]
  FOR m
    B[m,n] += T * C1[m,i]
```

(a) Abstract code for 2-index transform example

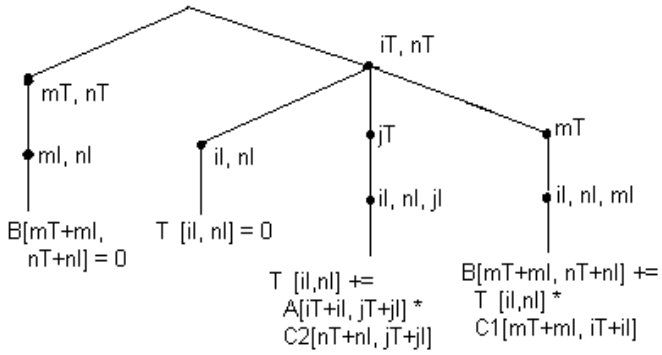


(b) Parse tree for 2-index transform example

Figure 2. Example of abstract code and corresponding parse tree for 2-index transform.

```
1.  FOR mT, nT, mI, nI
2.    B[mT+mI,nT+nI] = 0
3.  FOR iT, nT
4.    FOR iI, nI
5.      T[iI,nI] = 0
6.    FOR jT, iI, nI, jI
7.      T[iI,nI] += A[iT+iI,jT+jI] * C2[nT+nI,jT+jI]
8.    FOR mT, iI, nI, mI
9.      B[mT+mI,nT+nI] += T[iI,nI] * C1[mT+mI,iT+iI]
```

(a) Abstract tiled code for 2-index transform example



(b) Tiled parse tree for 2-index transform example

Figure 3. Example of abstract tiled code and corresponding parse tree for 2-index transform.

```
Input Arrays: (Read Placements)
A:  iI, nT
C2: iI, jT
C1: iI, nT

Output Arrays: (Write Placements)
B:
  Write Placement: iI,  mT
  Read Required  : Yes, Yes

Intermediates: (Write and Read Placements)
T: In Memory
```

(a) Candidate I/O placements

```
FOR mT, nT
  FOR mI, nI
    B[mI,nI] = 0
  Write BDisk
FOR jT, nT
  FOR jI, nI
    T[jI,nI] = 0
  FOR iT
    C2[1..Tn,1..Ti] = Read C2Disk
    A[1..Tj,1..Ti] = Read ADisk
    FOR jI, nI, iI
      T[jI,nI] += C2[nI,iI] * A[jI,iI]
  FOR mT
    B[1..Tm,1..Tn] = Read BDisk
    C1[1..Tm,1..Tj] = Read C1Disk
    FOR jI, nI, mI
      B[mI,nI] += T[jI,nI] * C1[mI,jI]
    Write BDisk
```

(b) Final concrete code for 2-index transform

Figure 4. Candidate I/O placements and final concrete code. $N_m = N_n = 35000, N_i = N_j = 40000$**, memory limit =** 1**GB, double precision arrays.**

both the statements is loop n_T. The write statement for the production and read statement for the consumption must be inside this n_T loop.

The approach to enumerating the placements for input, output and intermediate arrays is sketched below; details may be found in [38].

1. **Input Arrays:** Each loop index surrounding the consumption of an input array is considered as a candidate position for placing the read. At any candidate position, if there exists a redundant loop immediately surrounding it, then we ignore that position and move further up. A redundant loop for a read statement is one that does not index the array being read. We also ignore those read placements that cause the in-memory version of the input array to be a scalar or a vector. This is because, the resulting concrete code will involve in-memory matrix-matrix products using BLAS kernels, and scalar and vector operands will result in poor performance. Consider the abstract tiled code in Fig. 3. All loops surrounding statement 7 are candidate positions for placing the read for array A. Loops j_I and n_I are ignored so that the in-memory version of array A is at least two-dimensional. Loop j_T is not considered because the surrounding loop n_T is redundant for array A. Another important check that needs to be made is that the in-memory version of the array fits in memory. For every candidate position, we compute the memory cost of the corresponding *local buffer* assuming a tile size of one. If the buffer does not fit in memory, we do not move further up.

2. **Output Arrays:** The algorithm for enumerating write placements for an output array is exactly the same as that for input arrays, except that if any redundant loop surrounds the write statement, we need to insert a corresponding read for the array before the production. This is required as we will be re-accessing the disk array for every iteration of the redundant loop. For example, consider statement 9 in Fig. 3(a), where the output array B is produced. If the write for array B is placed just after loop m_T, an extra read will be required as the write will be surrounded by the redundant loop i_T.

3. **Intermediate Arrays:** If an intermediate array is written to disk, the algorithm for enumerating the disk read/write statements is exactly the same as for input/output arrays, except that the constraint specified earlier for intermediate arrays must be satisfied.

Fig. 4(a) shows the candidate read and write placements computed for each array in the code shown in Fig. 3(a).

4.2 DCS Input Construction

If all possible combinations of disk I/O placements, shown in Fig. 4(a), are considered for all the arrays, a very large number of cases will have to be evaluated. Our approach to avoid explicit evaluation for each combination of I/O placements is to encode the placement into the formulation of a nonlinear optimization problem that is input to the DCS system, as explained below. DCS attempts to minimize an objective function subject to equality and inequality constraints. The input to DCS comprises of *input parameters*, *variables*, *objective function*, and a set of *constraints*.

Parameters

The input parameters for our problem are the memory limit of the machine and the ranges $N_i, N_j, ...$ of the loop indices $i, j, ...$

Variables

The variables in our case include tile sizes $T_i, T_j, ...$ for loops $i, j, ...$ where each tile size variable has a lower bound of 1 and an upper bound of the full loop range. In addition to tile size variables, placement variables, $\lambda_i, i = 0, 1, 2, ...,$ are introduced for those arrays that have more that one candidate placement. These variables are used to determine the I/O placement for an array. Each of these λ variables is constrained to take value of either 0 or 1.

Objective Function

The objective function for our problem is the disk I/O cost. The disk I/O cost for an I/O statement is the product of the size of the array being read/written and the ranges of any redundant loops surrounding the statement. Consider the two possible read placements for input array A shown in Fig. 4(a). For the first read placement above loop i_I, the disk I/O cost will be:

$$D1_A = (N_n/T_n) \times \mathit{Size}_A$$

where the total size of array A is multiplied by the range of the redundant loop n_T. The disk I/O cost for the second read placement (above loop n_T), is $D2_A = Size_A$. Since there are two possible placements for A, $\lceil log_2(2) \rceil = 1$ placement variable λ_0 is introduced as follows, to express the disk I/O cost:

$$(\lambda_0 \times D1_A) + ((1 - \lambda_0) \times D2_A)$$

If $\lambda_0 = 1$, the first placement is selected, else if $\lambda_0 = 0$, the second one is selected. The placement encoding variables are constrained to have a value of 0 or 1.

Constraints

The first constraint is the memory limit. A static memory cost model is used, in which all the in-memory buffers are allocated memory at compile time. The total memory cost is the sum of the memory usage for all the individual in-memory buffers. The memory cost for an in-memory buffer is the product of the ranges of its indices. The memory cost expression for array A can be constructed along the same lines as the disk cost expression as follows. For the read placement above loop i_I, the in-memory buffer for input array A will be $A[i_I, j_I]$, which makes the memory cost $M1_A = T_i \times T_j$. On the other hand, for the read placement above loop n_T, the in-memory buffer is $A[i_I, j]$, thus making the memory cost $M2_A = T_i \times N_j$. The memory limit constraint using placement variable λ_0 is:

$$(\lambda_0 \times M1_A) + ((1 - \lambda_0) \times M2_A) \leq \text{MemoryLimit}$$

The placement variables are constrained to take values 0 or 1 as follows:

$$\lambda_i \times (1 - \lambda_i) = 0, i = 0, 1, 2 \ldots$$

We also introduce constraints on the minimum size of the in-memory version of an array. The arrays are stored in a blocked fashion on disk. The block sizes of the arrays are equal to the size of their in-memory versions, determined by the out-of-core code generation algorithm. A block is the basic unit of I/O and is chosen to be large enough to make the disk seek time negligible compared to the block transfer time. In [37], the incremental improvement obtained in the ratio of transfer time to seek time was observed to become negligible and approach the performance of sequential I/O beyond a block size, which depends on the system under consideration. The in-memory version of the array, and hence the block size, is constrained to be larger than this block size. For the system whose configuration is shown in Table 1, block size for reads must be at least 2MB, while that for writes must be at least 1MB.

In this manner, we can construct disk cost, memory cost and other constraint expressions for all arrays. Using these expressions, we build the input to DCS using the AMPL format [1]. DCS minimizes the objective function, that is, the disk I/O cost expression, while satisfying the memory limit, boundary, placement variable and buffer size constraints. DCS outputs values for the placement variables and tile sizes, thus giving the parameters for the concrete code.

The code generated for a multi-processor system uses the Global Arrays (GA) and Disk Resident Arrays (DRA) library [20, 19]. GA provides a shared-memory programming model while encouraging locality of access. DRA extends the shared-memory model to the secondary storage. GA/DRA provide an array abstraction in which the portion of data to be accessed is specified as a section of the array. In the generated code, the reads and writes from the disk are performed by the read and write routines in DRA. The in-memory computation is performed using kernel matrix multiplication libraries in GA. The I/O operations and the in-memory computations are collective operations.

5 Experimental Results

The performance of the generated concrete code was measured on the Itanium 2 Cluster at The Ohio Supercomputer Center [44]. Each node in the cluster has the configuration shown in Table 1. The generated code was compiled with the Intel Itanium Fortran Compiler for Linux. The imperfectly nested loop structure shown in Fig. 5 was used for the four-index transform example discussed in Sec. 2. The loop structure was given as input to two out-of-core code synthesis algorithms:

1. **Uniform Sampling Approach**, developed by extending the memory-cache algorithm from [10] to disk-memory hierarchy in [38]. Here, a greedy approach to disk I/O placement is used, where for each set of tile sizes, the algorithm places read/write statements immediately inside those loops at which the memory limit is exceeded. The tile size search space is sampled uniformly in a logarithmic fashion along each dimension. This sampled search space is then explored using a brute force approach.

2. **DCS-Based Approach**, described in this paper.

The sizes of the tensors (double precision) used for the experiment were: $N_p = N_q = N_r = N_s = (140, 190)$ and $N_a = N_b = N_c = N_d = (120, 180)$. The generated code was run sequentially and in parallel using 2 and 4 processors. Although the total physical memory size of each machine used for the experiments is 4GB, the codes were generated using a memory limit of 2GB per node. This is because the operating system kernel, write buffers, and other utilities occupy almost half of the physical memory on the machine. If the in-memory versions of the disk resident arrays in the generated code occupy more than half the memory, significant paging caused degradation of performance.

Table 2 shows the code generation times for the uniform sampling approach and for the DCS approach for the abstract code in Fig. 5. The DCS code generation process is orders of magnitude faster than the uniform sampling approach (which takes two hours to generate the solution compared to a couple of minutes for the DCS approach). For more complex computations such as energy calculations with higher order coupled cluster methods, the computational complexity of the uniform sampling approach makes it impractical. The DCS-based approach took no more than

Processor	OS	Compiler	Memory
Dual Itanium-2 (900 MHz)	Linux 2.4.18	efc version 7.1	4GB

Table 1. Configuration of the system whose I/O characteristics were studied.

```
T1[*,*,*,*]=0
FOR a,p,q,r,s
  T1[a,q,r,s]+=C4[p,a]*A[p,q,r,s]
B[*,*,*,*]=0
FOR a,b
  T3[*,*]=0
  FOR r,s
    T2=0
    FOR q
      T2+=C3[q,b]*T1[a,q,r,s]
    FOR c
      T3[c,s]+=C2[r,c]*T2
  FOR c,d,s
    B[a,b,c,d]+=C1[s,d]*T3[c,s]
```

Figure 5. Abstract code for AO-to-MO transform

a few minutes of execution time even for the most complex computations wo which we have applied it so far.

Table 3 shows the measured and predicted disk access times for the generated sequential concrete code. The measured times match quite well with the predicted times. The table presents results for the two approaches to out-of-core code generation. The code generated by the DCS approach has superior performance compared to the uniform sampling approach.

Table 4 shows the measured disk I/O times for the parallel code generated for two and four processors. It can be seen that the scaling of I/O time with four processors as compared to two processors is non-linear. This is because, as the number of processors is doubled, the total amount of aggregate physical memory available is also doubled, thus providing a decrease in the total amount of disk I/O, which is further shared and performed concurrently from the multiple local disks of the processors.

6 Conclusion

We have described an approach to the synthesis of out-of-core algorithms for a class of imperfectly nested loops. The approach was developed for the implementation in a component of a program synthesis system targeted at the quantum chemistry domain. The determination of optimal placements of disk I/O statements and choice of tile sizes requires search in a very large search space. By formulating it as a non-linear constrained optimization problem, and use of a general-purpose constrained optimization solver, dramatic reduction was achieved in the time taken to generate good solutions. Experimental results were provided that showed a good match between the predicted and measured performance for the synthesized code.

Acknowledgments

We would like to sincerely thank Prof. Benjamin Wah and Yi Xin Chen of the University of Illinois for their significant help with using the Discrete Constrained Search (DCS) Solver. We wish to express our appreciation to the reviewers of the paper for their suggestions. Special thanks go to anonymous “Reviewer #1”, whose thorough critique helped us rewrite significant portions of Section 4 and make other changes throughout the paper in an attempt to improve the presentation. We are grateful to the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

References

[1] A Modeling Language for Mathematical Programming (AMPL).

[2] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using ane partitioning. In *Proc. of the Eigth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 103–112. ACM Press, 2001.

[3] C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, August 1999.

[4] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *Proc. of Intl. Conf. on High Perf. Comp.*, 1999.

[5] C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. In *Proc. of Twelfth LCPC Workshop*, 1999.

[6] C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Parallel Processing Letters*, 7(2):157–168, 1997.

Memory Limit = 2GB		Uniform Sampling Approach	DCS Approach
Ranges (p, q, r, s)	Ranges (a, b, c, d)	Code generation time (secs)	Code generation time(secs)
140	120	7920	65
190	180	9000	118

Table 2. Code generation times for the two approaches to out-of-core code generation

Memory Limit = 2GB		Uniform Sampling Approach		DCS Approach	
Ranges (p, q, r, s)	Ranges (a, b, c, d)	Measured time (secs)	Predicted time (secs)	Measured time (secs)	Predicted time (secs)
140	120	426	430	227	296
190	180	2461	2630	1545	1537

Table 3. Measured and predicted sequential disk I/O times for the two approaches

[7] C. Lam, P. Sadayappan and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. In *Proc. of Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[8] 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. In *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, 2002.

[9] D. Cociorva, J. Wilkins, C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *Proc. of the Fifteenth ACM International Conference on Supercomputing (ICS'01)*, pages 500–509, 2001.

[10] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. E. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. In *Proc. of the Intl. Conf. on High Performance Computing*, volume 2228, pages 237–248. Springer-Verlag, 2001.

[11] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. In *Proc. of Seventeenth International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[12] Dr. Benjamin W. Wah and Yi Xin Chen. Web Interface for Discrete Constrained Search Solver.

[13] G. Baumgartner and D.E. Bernholdt and D. Cociorva and R. Harrison and S. Hirata and C. Lam and M. Nooijen and R. Pitzer and J. Ramanujam and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc. of Supercomputing 2002*, November 2002.

[14] G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proc. of Intl. Conf. on Supercomputing*, 1998.

[15] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proc. of ACM International Conference on Supercomputing (ICS 99)*, 1999.

[16] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. of SIGPLAN Conf. Programming Language Design and Implementation*, 1997.

[17] J. M. Anderson and S. P. Amarasinghe and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, July 1995.

[18] J. M. L. Martin. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. F. Schaefer III (Eds.). . *Encyclopedia of Computational Chemistry*, 1:115–128, 1998.

[19] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[20] J. Nieplocha, I. J. Harrison and R. J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[21] K. S. McKinley, S. Carr and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM TOPLAS*, 18(4):424–453, July 1996.

[22] M. E. Wolf and M. S. Lam. A Data Locality Algorithm. In *Proc. of ACM SIGPLAN PLDI*, 1991.

[23] M. E. Wolf, D. E. Maydan, and D. J. Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the Twenty Ninth Annual International Symposium on Microarchitecture*, pages 274–286, 1996.

[24] M. Kandemir, A. Choudhary, and J. Ramanujam. An I/O conscious tiling strategy for disk-resident data sets. *The Journal of Supercomputing*, 21(3):257–284, 2002.

[25] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy. A unified framework for optimizing locality, parallelism, and communication in out-of-core computations. *IEEE Transactions of Parallel and Distributed Systems*, 11(7):648–668, July 2000.

[26] M. Kandemir, A. Choudhary, J. Ramanujam and R. Bordawekar. Compilation techniques for out-of-core parallel computations. *Parallel Computing*, 24(3-4):597–628, June 1998.

[27] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines. Technical

		Measured time (secs)	
Number of processors	Total Memory limit	Uniform Sampling Approach	DCS Approach
2	4GB	997	778
4	8GB	491.6	368.4

Table 4. Measured parallel disk I/O times, for the two approaches to out-of-core code generation. $N_p = N_q = N_r = N_s = 140, N_a = N_b = N_c = N_d = 120$

Report 94509-S, Rice University, Houston, TX, December 1994.

[28] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. of Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[29] N. Ahmed and N. Mateev and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. of ACM Intl. Conf. on Supercomputing*, 2000.

[30] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *Intl. Journal of Parallel Programming*, 26(6):641–670, June 1998.

[31] R. Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs*. PhD thesis, Dept. of Electrical and Computer Eng., Syracuse University, April 1996.

[32] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Out-of-Core Stencil Codes. In *Proc. of Tenth ACM International Conference on Supercomputing*, pages 366–373, 1996.

[33] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A Model and Compilation Strategy for Out-of-Core Data-Parallel Programs. In *Proc. of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, 1995.

[34] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proc. of Scalable Parallel Libraries Conference*, pages 119–128, 1994.

[35] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. of the SIGPLAN '95 Conference on Programming Languages Design and Implementation*, 1995.

[36] S. Ghosh, M. Martonosi and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proc. of the Eighth ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[37] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam and P. Sadayappan. On Efficient Out-of-core Matrix Transposition. Technical Report OSU-CIRSC-9/03-T52, The Ohio State University, Columbus, OH, September 2003.

[38] S. Krishnan. DataLocality Optimization for Synthesis of Out-of-Core Programs. Master's thesis, The Ohio State University, Columbus, OH, September 2003.

[39] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. E. Bernholdt, and V. Choppella. Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms. In *Proc. of the Intl. Conf. on High Performance Computing*, 2003.

[40] T. Cormen and A. Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Technical Report PCS-TR94-243, Dartmouth College, November 1994.

[41] T. J. Lee and G. E. 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.

[42] T. Mowry, A. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proc. of Second Symposium on Operating Systems Design and Implementations*, pages 3–17, 1996.

[43] T. Wang. *Global Optimization of Constrained Nonlinear Programming*. PhD thesis, University of Illinois at Urbana-Champaign, IL, December 2000.

[44] The Ohio Supercomputer Center.

[45] W. Li. *Compiling for NUMA Parallel Machines*. PhD thesis, Cornell University, August 1993.

[46] W. Li. Compiler cache optimizations for banded matrix problems. In *Proc. of International Conference on Supercomputing*, 1995.

[47] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization. In *Proc. of Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[48] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. of ACM SIGPLAN PLDI*, 1999.