A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry: The Tensor Contraction Engine*

Gerald Baumgartner[1] David E. Bernholdt[2] Venkatesh Choppella[2]
J. Ramanujam[3] P. Sadayappan[1]
†‡§

Abstract

This paper provides an overview of a program synthesis system for a class of quantum chemistry computations. These computations are expressible as a set of tensor contractions and arise in electronic structure modeling. The input to the system is a a high-level specification of the computation, from which the system can synthesize high-performance parallel code tailored to the characteristics of the target architecture. Several components of the synthesis system are described, focusing on performance optimization issues that they address.

1 Introduction

Computers have made dramatic strides in speed over the last few decades, but unfortunately the ease of programming parallel computers has not made much progress. As computers have increased in achievable performance, making it feasible to accurately model more complex phenomena, the time and effort required to develop the software has become the bottleneck in many areas of science and engineering. The difficulty of developing high-performance software using the available languages and tools is being recognized as one of the most significant challenges today in the effective use of high-performance computers.

Over the last few years, we have engaged in a collaborative project with quantum chemists to develop a program transformation system to automatically transform a high-level specification of computations (expressed as tensor-contraction expressions) into optimized parallel programs. This paper provides an overview of the TCE system, and discusses the performance-model driven search-based approach to program transformation [5, 10, 26, 27] that are pursuing.

The paper is organized as follows. In the next section, we provide an overview of the components of the TCE system. Section 3 discusses the approach to storage and data locality optimization using a model-driven search-based approach. Section 4 discusses related work and we conclude in Section 5.

*Supported in part by the U.S. National Science Foundation and the U.S. Department of Energy

†[1] Dept. of Computer Science and Engineering, Ohio State University, {gb,saday}@cis.ohio-state.edu

‡[2] Oak Ridge National Laboratory, {bernholdtde,choppellav}@ornl.gov

§[3] Dept. of Electrical and Computer Engineering, Louisiana State University, jxr@ece.lsu.edu

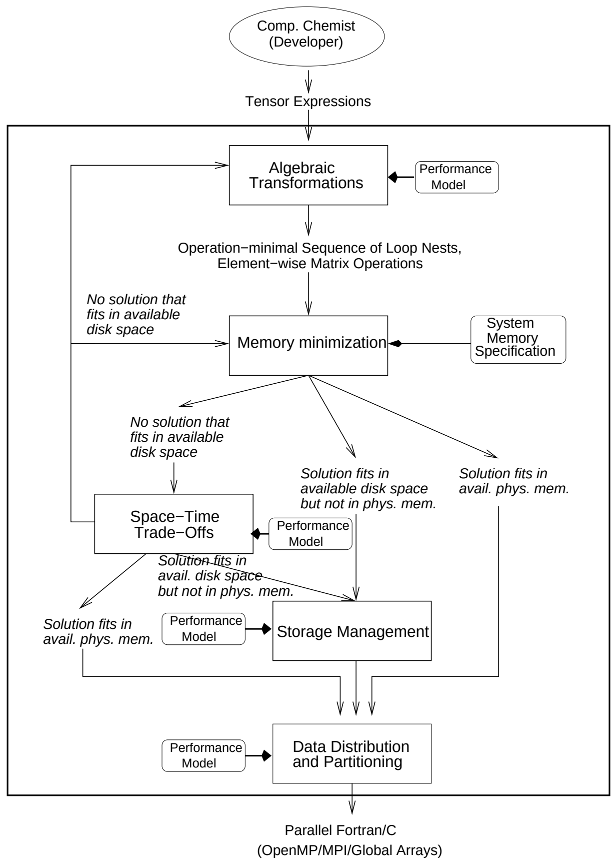
Comp. Chemist (Developer)
Tensor Expressions
Algebraic Transformations
Performance Model
Operation−minimal Sequence of Loop Nests, Element−wise Matrix Operations
No solution that fits in available disk space
Memory minimization
System Memory Specification
No solution that fits in available disk space
Solution fits in available disk space but not in phys. mem.
Solution fits in avail. phys. mem.
Space−Time Trade−Offs
Performance Model
Solution fits in avail. disk space but not in phys. mem.
Solution fits in avail. phys. mem.
Performance Model
Storage Management
Performance Model
Data Distribution and Partitioning
Parallel Fortran/C (OpenMP/MPI/Global Arrays)

Figure 1: The Synthesis System

2 Overview of the Tensor Contraction Engine

Fig. 1 shows a flowchart for the TCE system. A “front-end” (not shown) generates a set of working equations for the computational model, in the form of tensor contraction expressions, which are essentially comprised of a large number of generalized matrix products involving multi-dimensional arrays. The tensor contraction expressions, which may have hundreds of contraction terms, are transformed by the TCE into parallel Fortran programs that can be interfaced with quantum chemistry suites such [20]. The program transformation steps in the TCE include:

Operation minimization: The properties of commutativity and associativity of addition and multiplication and the distributivity of multiplication over addition are used to search for various possible ways of applying these properties to an input sum-of-products expression. A combination that results in an equivalent form of the computation with minimal operation cost is generated. The problem of determining an equivalent operation-minimal form of the expression is NP-complete, but efficient pruning-search procedures have been developed that are very effective in practice [30].

Memory minimization: The operation-minimal computation sequence synthesized by applying algebraic transformation often requires the use of large temporary intermediate arrays. The Memory Minimization step seeks to perform loop fusion transformations to reduce the memory requirements. An abstraction called the fusion-graph has been developed and has served as the basis for a search process used to evaluate alternate the loop fusion choices in the context of the TCE [29].

Storage and data locality optimization: If the space requirement exceeds physical memory capacity, portions of the arrays must be moved between disk and main memory as needed, in a way that maximizes reuse of elements in memory. Loop blocking transformations are used to minimize disk-to-memory transfer overhead [9, 8, 26, 27].

Space-time trade-off: When the memory-minimal solution is too large to fit in physical memory, an optimization framework is used to determine the optimal trade-off between redundant recomputation and I/O for large intermediates stored on disk [10].

Data distribution and partitioning: This step determines how best to partition the arrays among the processors of a parallel system. The data and work distribution that minimizes the total interprocessor communication in executing a sequence of tensor contractions is determined [12].

Due to space limitations, we restrict ourselves in this paper to providing more details about only the approach to storage and locality optimization.

3 Storage and Locality Optimization

A fundamental issue in transformation of a tensor contraction expression into an efficient program is that of minimizing the cost of I/O and memory copies. If one or more of the arrays are too large to fit within physical memory, disk I/O will be unavoidable. However, by suitable blocking of the data and its access, the cost of disk I/O can be minimized. In addition, if some arrays are intermediate results that are produced and then consumed, but not required upon exit, there may be opportunities to produce and consume them "by-parts", so that it may be possible to avoid the need to write them out to disk at all. This possibility can be modeled in a loop framework in terms of loop fusion and array contraction.

An Example: AO-to-MO Transformation

As an example, we consider a transformation often used in quantum chemistry codes to transform a set of two-electron integrals from an atomic orbital (AO) basis to a molecular orbital (MO) basis:

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

Here, $A(p,q,r,s)$ is an input four-dimensional array (assumed to be initially stored on disk), and $B(a,b,c,d)$ is the output transformed array, which needs to be placed on disk at the end of the calculation. The arrays $C1$ through $C4$ are called transformation matrices. In reality, these four arrays are identical; we identify them by different names in our example 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, and equal to $O + V$, where O is the number of occupied orbitals in the chemistry problem, V is 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:

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

This results in the creation of temporary intermediate arrays $T1$, $T2$, and $T3$:

$$T1(a,q,r,s) = \sum_{p} C4(p,a) \times A(p,q,r,s)$$

$$T2(a,b,r,s) = \sum_{q} C3(q,b) \times T1(a,q,r,s)$$

$$T3(a,b,c,s) = \sum_{r} C2(r,c) \times T2(a,b,r,s)$$

Assuming that the available memory limit on the machine running this calculation is less than V^4 (which is about $3TB$ for $V = 800$ and double precision arrays), any of the logical arrays A, $T1$, $T2$, $T3$, and B is too large to entirely fit in memory. Therefore, if the computation is implemented as a succession of four independent steps, the intermediates $T1$, $T2$, and $T3$ have to be written to disk once they are produced, and read from disk before they are used in the next step.

We use loop fusion and loop tiling to reduce memory requirements. To illustrate the benefit of loop fusion, consider the first two steps in the AO-to-MO transformation: $T1(a,q,r,s) = \sum_p C4(p,a) \times A(p,q,r,s)$; $T2(a,b,r,s) = \sum_q C3(q,b) \times T1(a,q,r,s)$. Fig. 2(a) shows the loop structure for the direct computation as a two-step sequence: first produce the intermediate $T1(1:Na, 1:Nq, 1:Nr, 1:Ns)$ and then use $T1$ to produce $T2(1:Na, 1:Nb, 1:Nr, 1:Ns)$. This is as an *abstract* form of a specification of the computation, because it cannot be executed in this form if the sizes of arrays are larger than limits due to the physical memory size.

Since all loops in either of the loop nests are fully permutable, and since there are no fusion-preventing dependences, the common loops a, q, r, and s can be fused. Once fused, the storage requirements for $T1$ can be reduced by contracting it to a scalar as shown in Fig. 2(b). Although the total number of arithmetic operations remains unchanged, the dramatic reduction in size of the intermediate array $T1$ implies that it can be completely stored in memory, without the need for any disk I/O for it. In contrast, if $Na \times Nq \times Nr \times Ns$ is larger than available memory, the unfused version will require that $T1$ be written out to disk after it is produced in the first loop, and then read in from disk for the second loop.

Performance Modeling

Fig. 3(a) shows one abstract code specification for AO-to-MO transform, where loop fusion has already been performed. Even after array contraction through loop fusion, some of the arrays may be larger than available physical memory. If so, blocks of such arrays must be copied into memory buffers for computation. In general, a block of a disk-resident array may have to be copied into memory multiple times. For a given abstract code such as shown in Fig. 3(a), there are an explosively large number of possible ways of generating concrete code with I/O statements for movement of blocks of disk-resident arrays into in-memory buffers. Each alternative code structure imposes a physical memory requirement and involves a data movement cost that can be quantified.

A convenient way of structuring the alternatives is by viewing the transformation of an abstract form of the code into a concrete form in multiple steps: 1) Tile each loop by splitting it into a pair of loops: tiling loop and intra-tile loop; 2) Move intra-tile loops inside nesting structures, subject to dependence constraints; and 3) Place disk I/O statements to copy blocks of disk-resident arrays from/to in-memory buffers.

```
double T1(Na,Nq,Nr,Ns)
double T2(Na,Nb,Nr,Ns)
T1(*,*,*,*) = 0
T2(*,*,*,*) = 0
FOR a = 1, Na
 FOR q = 1, Nq
  FOR r = 1, Nr
   FOR s = 1, Ns
    FOR p = 1, Np
      T1(a,q,r,s)
        += C4(p,a) * A(p,q,r,s)
END FOR p,s,r,q,a
FOR a = 1, Na
 FOR b = 1, Nb
  FOR r = 1, Nr
   FOR s = 1, Ns
    FOR q = 1, Nq
      T2(a,b,r,s)
        += C3(q,b) * T1(a,q,r,s)
END FOR q,s,r,b,a
```

(a) Unfused code

```
double T1(1,1,1,1)
double T2(Na,Nb,Nr,Ns)
T1(1,1,1,1) = 0
T2(*,*,*,*) = 0
FOR a = 1, Na
 FOR q = 1, Nq
  FOR r = 1, Nr
   FOR s = 1, Ns
    FOR p = 1, Np
      T1(1,1,1,1)
        += C4(p,a) * A(p,q,r,s)
    END FOR p
    FOR b = 1, Nb
      T2(a,b,r,s)
        += C3(q,b) * T1(1,1,1,1)
    END FOR b
END FOR s,r,q,a
```

(b) Fused code

Figure 2: Example of the use of loop fusion to reduce memory requirements

Fig. 3(b) shows one possible placement of intra-tile loops for this code. Given a particular tiled loop structure, such as in Fig. 3(b), there are still many possible choices for the placement of I/O statements, and choice of tile sizes for the various tiled loops. The fundamental objectives in making the choice are to minimize the total disk I/O cost, subject to the physical memory constraints. The placement of disk I/O statements and tile sizes both affect the disk I/O cost as well as the total physical memory requirement. At one extreme, if disk I/O statements are placed at the innermost possible positions, just before/after computational statements inside loop nests, only a single memory location will be needed for each array, but the disk I/O cost would be prohibitively large. At the other extreme, if unlimited physical memory were available, the disk I/O statements could be moved all the way out, with all reads of input arrays happening before any of the loops of the computation, and the writes being done at the end of all the loops. In practice, for computations that cannot fit within physical memory, some intermediate placements of disk I/O statements, along with suitable choice of tile sizes will be best.

Quantifying Data Movement Costs

We next develop cost expressions for memory usage and disk I/O cost, as a function of tile sizes and placements of disk I/O statements. Let the tile sizes be $T_a, T_b, ...$ for loops $a, b, ...$ 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 to encode placement choices, as explained below. These variables are used to determine where the I/O statement for an array will be placed. Each of these λ values are constrained to take only values 0 or 1 by using the following equality constraint:

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

Consider the possible read placements for input array $C4$. Irrespective of the read placement, the disk access cost will include the total size of the disk version of the array. If any redundant loops (i.e. those that do not explicitly occur in any of the index expressions) surround the read statement,

```
FOR a,b,r,s
  T2[b,a,r,s]=0
FOR r,s
  FOR a,q
    T1[a,q]=0
  FOR a,p,q
    T1[a,q]+=C4[p,a]*A[p,q,r,s]
  FOR a,b,q
    T2[b,a,r,s]+=C3[q,b]*t_1[a,q]
FOR a,b,c,d
  B[a,b,c,d]=0
FOR a,b,c
  FOR s
    T3[s]=0
  FOR r,s
    T3[s]+=C2[r,c]*t_2[b,a,r,s]
  FOR d,s
    B[a,b,c,d]+=C1[s,d]*T3[s]
```

(a) Abstract Code for AO-to-MO transform

```
FOR aT,bT,rT,sT,aI,bI,rI,sI
  T2[bT+bI,aT+aI,rT+rI,sT+sI]=0
FOR rT,sT,rI,sI
  FOR aT,qT,aI,qI
    T1[aT+aI,qT+qI]=0
  FOR aT,pT,qT,aI,pI,qI
    T1[aT+aI,qT+qI]+=C4[pT+pI,aT+aI]
                *A[pT+pI,qT+qI,rT+rI,sT+sI]
  FOR aT,bT,qT,aI,bI,qI
    T2[bT+bI,aT+aI,rT+rI,sT+sI]+=
            C3[qT+qI,bT+bI]*T1[aT+aI,qT+qI]
FOR aT,bT,cT,dT,aI,bI,cI,dI
  B[aT+aI,bT+bI,cT+cI,dT+dI]=0
FOR aT,bT,cT,aI,bI
  FOR sT,cI,sI
    T3[sT+sI,cI]=0
  FOR rT,sT,cI,rI,sI
    T3[sT+sI,cI]+=C2[rT+rI,cT+cI]
              *T2[bT+bI,aT+aI,rT+rI,sT+sI]
  FOR dT,sT,cI,dI,sI
    B[aT+aI,bT+bI,cT+cI,dT+dI]+=
              C1[sT+sI,dT+dI]*T3[sT+sI,cI]
```

(b) One possible structure for tiling of loops

```
C4[pT+pI,aT+aI]=Read(C4Disk,1,5600.0)
C3[qT+qI,bT+bI]=Read(C3Disk,1,5600.0)
FOR rT,sT,rI,sI
  FOR aT,qT,aI,qI
    T1[aT+aI,qT+qI]=0
  A[pT+pI,qT+qI]=Read(ADisk,sT/80*1+rT/80+1,6400.0)
  FOR aT,pT,qT,aI,pI,qI
    T1[aT+aI,qT+qI]+=C4[pT+pI,aT+aI]*A[pT+pI,qT+qI]
  FOR aT,bT
    FOR aI,bI
      T2[bI,aI]=0
    FOR qT,aI,bI,qI
      T2[bI,aI]+=C3[qT+qI,bT+bI]*T1[aT+aI,qT+qI]
    Write(T2Disk,sT/80*1*2*2+rT/80*2*2+
            bT/35*2+aT/35+1,1225.0)
C2[rT+rI,cT+cI]=Read(C2Disk,1,5600.0)
C1[sT+sI,dT+dI]=Read(C1Disk,1,5600.0)
FOR aT,bT
  T2[bI,aI,rT+rI,sT+sI]=
        Read(T2Disk,bT/35*2+aT/35+1,7840000.0)
  FOR cT,aI,bI
    FOR sT,cI,sI
      T3[sT+sI,cI]=0
    FOR rT,sT,cI,rI,sI
      T3[sT+sI,cI]+=
         C2[rT+rI,cT+cI]*T2[bI,aI,rT+rI,sT+sI]
    FOR dT
      FOR cI,dI
        B[cI,dI]=0
      FOR sT,cI,dI,sI
        B[cI,dI]+=C1[sT+sI,dT+dI]*T3[sT+sI,cI]
      Write(BDisk,dT/70*1*2*2+cT/70*2*2
               +bT/35*2+aT/35+1,4900.0)
```

(c) Concrete Code for AO-to-MO transform with Disk I/O

Figure 3: Example of abstract, tiled and concrete code with I/O for AO-to-MO transform $N_a = N_b = N_c = N_d = 70, N_p = N_q = N_r = N_s = 80$, Memory Limit = 128MB, Double Precision Arrays.

they add a multiplicative cost to the accesses of the array. For the possible placement above loop q_T, the disk access volume will be:

$$N_r \times N_s \times \mathit{Size}_{C4}$$

where the total size of array $C4$ is multiplied by the ranges of the redundant loops s_T, r_T, s_I, r_I. Although this only counts the number of elements accessed, the cost model can be refined to account for seek time etc. The disk access cost for read placement above loop p_T is again $N_r \times N_s \times Size_{C4}$. However, the cost if the read is placed above loop r_T is $Size_{C4}$. It can be reasoned in this case that only these three possible placements need be considered for I/O for $C4$ and that other potential placements are either inferior or equivalent in terms of memory usage cost and disk access cost. So for $C4$, $\lceil log_2(3) \rceil = 2$ placement variables λ_0 and λ_1 are introduced as follows:

$$(\lambda_0 \times \lambda_1 \times N_r \times N_s \times \mathit{Size}_{C4}) + (\lambda_0 \times (1 - \lambda_1) \times N_r \times N_s \times \mathit{Size}_{C4})$$
$$+((1 - \lambda_0) \times \lambda_1 \times \mathit{Size}_{C4})$$

If $\lambda_0 = 1, \lambda_1 = 1$, the first placement is selected, if $\lambda_0 = 1, \lambda_1 = 0$, the second one is selected, and so on. Along the same lines, the memory cost expression for $C4$ using the placement variables can be written as follows:

$$(\lambda_0 \times \lambda_1 \times T_p \times T_a) + (\lambda_0 \times (1 - \lambda_1) \times N_p \times T_a) + ((1 - \lambda_0) \times \lambda_1 \times N_p \times N_a) +$$
$$((1 - \lambda_0) \times (1 - \lambda_1) \times 2 \times memoryLimit)$$

For $(\lambda_0, \lambda_1) = (1, 1), (1, 0), (0, 1)$, there are valid disk cost and memory cost terms. However, for $(\lambda_0, \lambda_1) = (0, 0)$, the following memory cost term is introduced:

$$((1 - \lambda_0) \times (1 - \lambda_1) \times 2 \times memoryLimit)$$

so that the memory constraint is never satisfied for this combination of values for the placement variables. No disk cost term needs to be introduced for this combination.

Quantifying Memory Usage Costs

Next, we enumerate constraints on the size of the in-memory buffer for the arrays. Consider different possible placements for array A. If the read for array A is placed just above loop a_I, its in-memory buffer size will be $T_p \times T_q$, and the following constraint is imposed on the buffer size:

$$T_p \times T_q \times \mathit{sizeof(double)} \geq 2\text{MB}$$

There turn out to be six possible placements to be considered for array A, and so three placement variables $\lambda_i, i = 2, 3, 4$, are used in the disk cost and memory cost expressions for A, to construct the entire constraint expression. For example, the first term in the expression will be

$$\lambda_2 \times \lambda_3 \times \lambda_4 \times T_p \times T_q \times \mathit{sizeof(double)}$$

while the second term will be

$$\lambda_2 \times \lambda_3 \times (1 - \lambda_4) \times T_p \times N_q \times \mathit{sizeof(double)}$$

and so on.

In this manner, we construct disk cost, memory cost and other constraint expressions for all arrays. The generation of optimized concrete code can be done by solving a constrained optimization problem: minimize the disk access cost expression, subject to the constraints that requires the memory usage expression to be no greater than available physical memory. This can be done by use of a specialized search procedure, or a general-purpose solver. For the particular example in Fig. 3(b), use of the DCS solver [1] resulted in the optimized values shown in the concrete form of the code shown in Fig. 3(c).

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.

Optimizations included and omitted	Total Disk I/O I/O time (secs)	Total execution time (secs)
Fusion + Optimizing Tiling	248.43	954.87
No Fusion, Optimizing Tiling	747.83	1261.95
No Fusion, Tile size = 4^{th} root of memorySize/3	1240.85	1957.18

Table 2: Total disk I/O and execution times for codes generated for all three cases.

Experimental Results

In this section, we present experimental data on the performance of code generated by the TCE for the four-index transform calculation discussed earlier. Three different combinations of optimizations were used to generate code with explicit disk I/O statements.

1. **Fusion + Optimized Tiling:** The TCE loop fusion and tiling optimizations were enabled [5, 26].
2. **No Fusion, Optimized Tiling:** Loop fusion was disabled, but the TCE tiling optimization was enabled.
3. **No Fusion, Standard Tiling:** Loop Fusion was disabled; the tile sizes of all loops were standardized to $1/3$ of the 4^{th} root of the memory size. This is the approach used in current quantum chemistry codes.

The sizes of the tensors used for the experiments were $N_a = N_b = N_c = N_d = V = 140$ and $N_p = N_q = N_r = N_s = N = 150$. The performance of the generated concrete code was measured on the Itanium 2 Cluster at The Ohio Supercomputer Center. Each node in the cluster has the configuration shown in Table 1. Since not all of the physical memory can be used for data, the memory limit for the optimizations was set to 2GB.

As can be seen, the code with standard tiling has the most redundant disk I/O. This is the state of the art for the code generators currently used by chemists. Table 2 shows the disk I/O times and total execution times of the generated code for all three cases. Our combined fusion and tiling optimizations result in code that has 80% less disk I/O than the code with standard tiling.

4 Related Work

Aspects of some of the important problems addressed in the synthesis system such as operation minimization, memory reduction and locality optimization have also received attention in research on compiler optimizations.

Some recent work has explored the use of loop fusion for memory reduction for sequential execution. Fraboulet et al. [16] use loop alignment to reduce memory requirement between adjacent loops by formulating the one-dimensional version of the problem as a network flow problem. Song et. al. [43] present a different network flow formulation of the memory reduction problem and they include a simple model of cache misses as well. However, they do not consider the issue of trading off memory for recomputation.

Considerable research on loop transformations for locality in nested loops has been reported in the literature [13, 32, 36, 46]. Over the fifteen years, there has also been considerable progress towards developing powerful frameworks based on the polyhedral model of loop computations [3, 4, 7, 14, 19]. 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. Wolf et al. [47] consider the integrated treatment of fusion and tiling only from the point of view of enhancing locality and do not consider the impact of the amount of required memory; the memory requirement is a key issue for the problems considered in the context of the TCE. Loop tiling for enhancing data locality has been studied extensively [13, 24, 25, 41, 46, 47, 42], and analytic models of the impact of tiling on locality in perfectly nested loops have been developed [18, 31, 39]. Frameworks for handling imperfectly nested loops have been presented in [2, 33, 42]. Ahmed et. al. [2] 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. [33] develop a framework based on affine partitioning and blocking to reduce synchronization and improve data locality. Specific issues of locality enhancement, I/O placement and optimization, and automatic tile size selection have not been addressed in the works that can handle imperfectly nested loops [2, 33, 42].

The approach undertaken in this project bears similarities to some projects in other domains, such as the SPIRAL project which is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms [21, 48]. SPIRAL generates efficient implementations of algorithms expressed in a domain-specific language called SPL by a systematic search through the space of possible implementations. Other efforts in automatically generating efficient implementations of programs include FFTW [17], the telescoping languages project [22], ATLAS [45] for deriving efficient implementation of BLAS routines, the PHIPAC [6] project, and the TUNE project [44]. In addition, motivated by the difficulty of detecting and optimizing matrix operations hidden in array subscript expressions within loop nests, several projects have worked on efficient code generation from high-level languages such as MATLAB and Maple [15, 35, 37, 38].

While our effort shares some common goals with several of the projects mentioned above, there are also significant differences. Some of the optimizations we consider, such as the algebraic optimizations, memory minimization, and space-time trade-offs, do not appear to have been previously explored, to the best of our knowledge. We also take advantage of certain domain-specific properties of the computations; for example, since all expressions considered in this framework are tensor contractions, the loops of the resulting code are fully permutable, and there are no dependencies preventing fusion. This observation is crucial for the optimization algorithms of several components (memory minimization, space-time transformation, data locality).

5 Current Status

We have taken a two-pronged approach to concurrently develop two versions of the TCE: one developed by our collaborators at Pacific Northwest National Laboratory PNNL, that incorporates many domain-specific features such as spin and symmetry, but does not implement many of the optimizations and transformations shown in Fig. 1; and another being developed at OSU, that is based on a framework for program transformation and optimization.

The prototype TCE system has automatically derived and implemented optimized, parallel programs for various, high-order, single-reference configuration interaction (CI), coupled cluster (CC), and many-body perturbation theory (MBPT) models for the description of electron correlation in the ground state. These are the initial parallel implementations for many of these models, and have proved competitive to equivalent hand-written programs in sequential performance. These implementations are being distributed with the latest release of NWChem 4.5 [20].

Theory	# Terms	# Lines	Year
CCD	11	3209	1978
CCSD	48	13213	1982
CCSDT	102	33932	1988
CCSDTQ	183	79901	1992

Table 1: History of development of implementations of Coupled Cluster methods

It is estimated that already several person-years of effort have been saved in the implementation of these methods using the TCE. Table 1 gives an idea of the potential benefits from using the automated synthesis capability of the TCE. It lists members of the Coupled-Cluster family in increasing order of the method, the number of terms involved in the tensor contraction expression, the number of lines of code for the synthesized parallel Fortran code, and the year of first manual implementation of the method. A fourteen year span separated the availability of a CCD code and a CCSDTQ code. Starting from scratch, the TCE can synthesize codes for all these methods, representing tens of thousands of lines of code, in a matter of minutes to hours.

References

[1] A Modeling Language for Mathematical Programming(AMPL). *http://www.ampl.com/*

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

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 1991.

[4] C. Bastoul. Efficient code generation for automatic parallelization and optimization. *Proceedings of Second International Symposium on Parallel and Distributed Computing*, Oct. 2003.

[5] A. Bibireata, S. Krishnan, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, V. Choppella, Memory-Constrained Data Locality Optimization for Tensor Contractions. In *Proc. 16th LCPC Workshop*, College Station, TX, Oct. 2003.

[6] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. ACM Intl. Conf. on Supercomputing,* pages 340–347, 1997.

[7] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing* Vol. 24, pages 421-444, 1998.

[8] 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.

[9] 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.

[10] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. *Proc. of the 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[11] D. Cociorva, G. Baumgartner, C.-C Lam, P. Sadayappan, and J. Ramanujam. Memory-Constrained Communication Minimization for a Class of Array Computations. *Languages and Compilers for Parallel Computing*, 2002.

[12] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. *Proc. of 17th International Parallel & Distributed Processing Symposium (IPDPS)*, Apr. 2003.

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

[14] J.-F. Collard, T. Risset, and P. Feautrier. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, 1995.

[15] L. De Rose and D. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proc. 10th ACM Intl. Conf. on Supercomputing,* 1996.

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

[17] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. ICASSP 98,* Volume 3, pages 1381–1384, 1998, `http://www.fftw.org`.

[18] S. Ghosh, M. Martonosi and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. *8th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[19] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. *Proc. International Conference on Parallel Architectures and Compilation Techniques*, pages 106–111, 1998.

[20] High Performance Computational Chemistry Group, *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.5* (2003), Pacific Northwest National Laboratory, Richland, Washington 99352, USA.

[21] J. Johnson, R. Johnson, D. Padua, and J. Xiong. Searching for the best FFT formulas with the SPL compiler. In *Languages and Compilers for High-Performance Computing*, Springer-Verlag, 2001.

[22] K. Kennedy, B. Broo, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *J. Parallel and Distributed Computing,* 2001.

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

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

[25] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proc. ACM Intl. Conf. on Supercomputing (ICS 99),* Rhodes, Greece, June 1999.

[26] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, V. Choppella, Data Locality Optimization for Synthesis of Efficient Out-of-Core Algoritms. In *Proc. Intl. Conf. on High Perf. Comp.*, Dec. 2003.

[27] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, V. Choppella and P. Sadayappan. Efficient Synthesis of Out-of-core Algorithms Using a Nonlinear Optimization Solver. *Proc. of 18th International Parallel & Distributed Processing Symposium (IPDPS)*, Apr. 2004.

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

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

[30] C.-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.

[31] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Palo Alto, CA, April 1991.

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

[33] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using ane partitioning. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming,* pages 103-112, 2001.

[34] 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.* Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115–128, 1998.

[35] The Match Project. A MATLAB compilation environment for distributed heterogeneous adaptive computing systems. `www.ece.nwu.edu/cpdc/Match/Match.html.`

[36] K. S. 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.

[37] V. Menon, K. Pingali. High-Level Semantic Optimization of Numerical Codes. In *Proc. ACM Intl. Conf. on Supercomputing.* 1999.

[38] V. Menon, K. Pingali. A Case for Source-Level Transformations in MATLAB. In *Proc. 2nd Conf. on Domain-Specific Languages.* 1999.

[39] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *Intl. Journal on Parallel Programming,* June 1998.

[40] F. Quillere, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469 - 498, October 2000.

[41] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. *Proc. of the 1998 Intl. Conf. on Supercomputing,* Melbourne, Australia, July 1998.

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

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

[44] M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's matrix multiplication for memory hierarchies. In *Proc. SuperComputing '98*.

[45] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. SC '98* (Electronic Publication), IEEE Publication, 1998.

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

[47] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proc. MICRO-29,* pages 274–286, Paris, France, December 1996.

[48] J. Xiong, D. Padua, and J. Johnson. SPL: A language and compiler for DSP algorithms. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2001.