A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry*

Gerald Baumgartner[1] David E. Bernholdt[2] Daniel Cociorva[1] Robert Harrison[3] So Hirata[3]
Chi-Chung Lam[1] Marcel Nooijen[4] Russell Pitzer[5] J. Ramanujam[6] P. Sadayappan[1]

[1] CIS Dept., Ohio State University, {gb,cociorva,clam,saday}@cis.ohio-state.edu
[2] Oak Ridge National Laboratory, bernholdtde@ornl.gov
[3] Pacific Northwest National Laboratory, Robert.Harrison@pnl.gov
[4] Chemistry Department, Princeton University, Nooijen@Princeton.edu
[5] Chemistry Department, Ohio State University, Pitzer.3@osu.edu
[6] ECE Department, Louisiana State University, jxr@ece.lsu.edu

Abstract

This paper discusses an approach to the synthesis of high-performance parallel programs for a class of computations encountered in quantum chemistry and physics. These computations are expressible as a set of tensor contractions and arise in electronic structure modeling. An overview is provided of the synthesis system, that transforms a high-level specification of the computation into high-performance parallel code, tailored to the characteristics of the target architecture. An example from computational chemistry is used to illustrate how different code structures are generated under different assumptions of available memory on the target computer.

1 Introduction

The complexity of developing high-performance parallel software has led to many efforts aimed at raising the level of abstraction above message-passing using MPI. The approaches range from a) new parallel languages like ZPL [32] and parallel extensions to general-purpose sequential languages like C [5] Java [36] and Fortran [30], to b) parallel libraries and problem solving environments like SCALAPACK [6], PLAPACK [1], UHFFT [27], Global Arrays [29], OVERTURE [4], Cactus [31], PETSc [2], Broadway [13] etc., to c) domain-specific synthesis from high-level specification, such as SPIRAL for the signal processing domain [28].

In this paper, we provide an overview of a project that is developing a program synthesis system to facilitate the rapid development of high-performance parallel programs for a class of scientific computations encountered in chemistry and physics — electronic structure calculations, where many computationally intensive components are expressible as a set of tensor contractions. Currently, manual development of accurate quantum chemistry models in this domain is very tedious and takes an expert several months to years to develop and debug. The synthesis tool aims to reduce the development time to hours/days, by having the chemist specify the computation in a high-level form, from which an efficient parallel program is automatically synthesized. This should enable the rapid synthesis of high-performance implementations of sophisticated ab-initio quantum chemistry models, including models that are too tedious for manual development by quantum chemists.

The computational domain that we consider is extremely compute-intensive and consumes significant computer resources at national supercomputer centers. Many of these codes are limited in the size of the problem that they can currently solve because of memory and performance limitations. The computational structures that we address are present in some computational physics codes modeling electronic properties of semiconductors and metals, and in computational chemistry codes

*Supported in part by the National Science Foundation through the ITR program (CHE-0121676, CHE-0121706 and CHE-0121383), and the U. S. Department of Energy through award DE-AC05-00OR22725.

such as ACES II, GAMESS, Gaussian, NWChem[14], 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 [22, 23]. 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 crucial to the understanding of chemical processes in real-world systems. Examples of applications include combustion and atmospheric chemistry, chemical vapor deposition, protein structure and enzymatic chemistry, and industrial chemical processing. Computational chemistry and materials science account for significant fractions of supercomputer usage at national centers (for example, approximately 85% of total usage at Pacific Northwest National Laboratories, 30% at NERSC, and about 50% of one of SDSC's systems).

The synthesis of efficient parallel code from a high-level specification as a set of tensor contractions requires many optimization issues to be addressed. We use an example to illustrate the issues and solution approach, and provide pointers to other publicatiosn that provide about the compiler transformations. Section 2 provides some background about the computational context addressed. Section 3 uses a motivating example that abstracts a computation implemented in many quantum chemistry packages. Section 4 provides an overview of the components of the synthesis system. Section 5 provides pseudocode output by the synthesis system for the motivating example, under different assumptions of available memory. This illustrates the power of the system to generate tailored code optimized for the problem and system parameters. Section 6 provides a discussion along with pointers to related work.

2 The computational context

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

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

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 $4 \times N^{10}$ if the range of each index $a - l$ is N. Instead, the same expression can be rewritten by use of associative and distributive laws:

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

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

The operation minimization problem encountered here is a generalization of the well known matrix-chain multiplication problem, where a linear chain of matrices to be multiplied is given, e.g., ABCD, and the optimal order of pair-wise multiplications is sought, i.e., ((AB)C)D versus (AB)(CD), etc. In contrast, for computations expressed as sets of matrix contractions, there is additional freedom in choosing the pair-wise products. For the above example, instead of forcing a single chain order, e.g., ABCD, other orders are possible, such as the BDCA order shown for the operation-reduced form above.

We have shown that the problem of determining the operator tree with minimal operation count is NP-complete and have developed a pruning search procedure [20, 21] that is very efficient in practice. For the above example, although the latter form is far more economical in terms of the number of operations, its implementation will require the use of temporary intermediate arrays to hold the partial results of the parenthesized array subexpressions. 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. 1(c). This way, $T1$ can be reduced to a scalar and $T2$ to a 2-dimensional array, without changing the number of operations.

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

(a) Formula sequence

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

(b) Direct implementation (unfused code)

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

(c) Memory-reduced implementation (fused)

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

For a computation comprising a number of nested loops, there are often many 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 addressed the problem of finding the choice of fusions for a given operator tree that minimized the total space required for all arrays [17, 18, 19].

3 A motivating example from CCSD(T)

One of the most computationally intensive components of many quantum chemistry packages is the CCSD(T) scheme. It is a coupled cluster approximation that includes single and double excitations from the Hartree-Fock wavefunction plus a perturbative estimate for the *connected* triple excitations. For molecules well described by a Hartree-Fock wave function, this method predicts bond energies, ionization potentials, and electron affinities to an accuracy of ± 0.5 kcal/mol, bond lengths accurate to ± 0.0005 Å, and vibrational frequencies accurate to $\pm 5cm^{-1}$. This level of accuracy is adequate to answer many of the questions that arise in studies of chemical systems.

As a motivating example to illustrate some of the pertinent optimization issues addressed, we discuss a component of the CCSD(T) calculation. The following representative equation arises in the Laplace factorized expression for linear triples perturbation correction:

$$\begin{aligned} A3A &= X_{ce,af}Y_{ae,cf} + X_{a\bar{e},c\bar{f}}Y_{c\bar{e},a\bar{f}} + X_{a\bar{e},\bar{c}f}Y_{\bar{c}\bar{e},af} \\ &+ X_{\bar{a}e,c\bar{f}}Y_{ce,\bar{a}\bar{f}} + X_{\bar{a}e,\bar{c}f}Y_{\bar{c}e,\bar{a}f} + X_{\bar{a}\bar{e},\bar{c}\bar{f}}Y_{\bar{c}\bar{e},\bar{a}\bar{f}}, \end{aligned}$$

where X and Y are of the form $X_{ae,cf} = t_{ij}^{ae}t_{ij}^{cf}$ and $Y_{ce,af} = \langle cb \parallel ek\rangle\langle ab \parallel fk\rangle$, respectively.

Integrals with two vertical bars have been antisymmetrized and may be expressed as: $(\langle pq \parallel rs\rangle = \langle pq \mid rs\rangle - \langle pq \mid sr\rangle)$, where integrals with one vertical bar are of the form $\langle \mu\nu \mid \omega\lambda\rangle = \int\int dr^3 ds^3 \phi_\mu(\underline{\mathrm{r}})\phi_\nu(\underline{\mathrm{s}})|\underline{\mathrm{r}} - \underline{\mathrm{s}}|^{-1}\phi_\omega(\underline{\mathrm{r}})\phi_\lambda(\underline{\mathrm{s}})$ and are quite expensive to compute (requiring on the order of 1000 arithmetic operations). Electrons may have either up or down (or alpha/beta) spin. Down spin is denoted here with an over-bar. The indices i, j, k, l, m, n refer to occupied orbitals, of number O between 30 and 100. The indices a, b, c, d, e, f refer to unoccupied orbitals of number V between 1000 and 3000. The integrals are written in the molecular orbital basis, but must be computed in the underlying atom-centered Gaussian basis, and transformed to the molecular orbital basis. We omit these details in our initial discussion here.

A3A is one of many contributions to the energy, and among the most expensive, scaling as $O(OV^5)$. Here, we assume that we have already computed the amplitudes t_{ij}^{ae}, and they must be read as necessary, and contracted to form a block of X. The integrals $\langle cb \parallel ek\rangle$ must be recomputed as necessary, contracted to form a block of Y corresponding to X, and the two contracted to form the scalar contribution to the energy.

Fig. 2 shows pseudo-code for the computation of one of the energy components E for $A3A$. Temporary arrays $T1$ and $T2$ are used to store the integrals of form $\langle ab \parallel ek\rangle$, where the functions f_1 and f_2 represent the integral calculations (in reality, f_1 and f_2 represent the same array/function, but it is more convenient to treat them as distinct initially, to simplify our explanation about the space-time trade-off problem addressed by the synthesis system). The intermediate quantities X_{aecf} are computed by contracting over (i.e., summing over products of) input array T, while the intermediate quantities Y_{ceaf} are obtained by contracting over $T1$ and $T2$. The final result is a single scalar quantity E, that is obtained by adding together the $O(OV^3)$ pair-wise products $X_{aecf}Y_{ceaf}$.

The cost of computing each integral f_1, f_2 is represented by C_f, and in practice is of the order of hundreds or a few thousand arithmetic operations. The pseudo-code form shown in Fig. 2 is computationally very efficient in minimizing the

```
for a, e, c, f
[ for i, j
  [ X_aecf += T_ijae T_ijcf
for c, e, b, k
[ T1_cebk = f1(c, e, b, k)
for a, f, b, k
[ T2_afbk = f2(a, f, b, k)
for c, e, a, f
[ for b, k
  [ Y_ceaf += T1_cebk T2_afbk
for c, e, a, f
[ E += X_aecf Y_ceaf
```

array	space	time
X	V^4	V^4O^2
T1	V^3O	C_fV^3O
T2	V^3O	C_fV^3O
Y	V^4	V^5O
E	1	V^4

Figure 2. Unfused operation-minimal form.

```
for a, e, c, f
[ for i, j
  [ X_aecf += T_ijae T_ijcf
for a, f
[ for c, e, b, k
  [ T1_cebk = f1(c, e, b, k)
for c, e
[ for a, f, b, k
  [ T2_afbk = f2(a, f, b, k)
for c, e, a, f
[ for b, k
  [ Y_ceaf += T1_cebk T2_afbk
for c, e, a, f
[ E += X_aecf Y_ceaf
```

⇒

```
for a, e, c, f
[ for i, j
  [ X += T_ijae T_ijcf
  for b, k
  [ T1 = f1(c, e, b, k)
  [ T2 = f2(a, f, b, k)
  [ Y += T1 T2
[ E += X Y
```

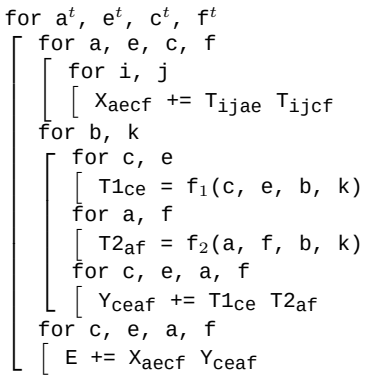
array	space	time
X	1	V^4O^2
T1	1	C_fV^5O
T2	1	C_fV^5O
Y	1	V^5O
E	1	V^4

Figure 3. Use of redundant computation to allow full fusion.

number of expensive integral function evaluations f_1 and f_2, and maximizing the reuse of the stored integrals in $T1$ and $T2$ (each element of $T1$ and $T2$ is used $O(V^2)$ times). However, it is impractical due to the huge memory requirement. For example, with $O = 100$ and $V = 5000$, the size of $T1, T2$ is $O(10^{14})$ bytes and the size of X, Y is $O(10^{15})$ bytes. By fusing pairs of producer-consumer loops, reductions in the array sizes may be obtained, since the array dimension corresponding to the fused loop can be eliminated from the intermediate array. It can be seen that the loop that produces X (with indices a, e, c, f), the loop that produces Y (with indices c, e, a, f) and the loop that consumes X and Y to produce E (with indices c, e, a, f) can all be fully fused together, permitting the elimination of all explicit indices in X and Y to reduce them to scalars. However, the loops producing $T1$ (with indices c, e, b, k) and $T2$ (with indices a, f, b, k) cannot also be directly fused with the other three loops because their indices do not match.

Fig. 3 shows how a reduction of space for $T1$ and $T2$ can be achieved by introducing redundant loops around their producer loops — add loops with the missing indices a, f for $T1$ and c, e for $T2$. Now all five loops have common indices a, e, c, f that can be fused, permitting elimination of those indices from all temporaries. Further, by fusing the producer loops for $T1$ and $T2$ with their consumer loop, which produces Y, the b, k indices can also be eliminated from $T1$ and $T2$. A dramatic reduction of memory space is achieved, reducing all temporaries $T1, T2, X$ and Y to scalars, but the space savings come at the price of a significant increase in computation. No reuse is achieved of the quantities derived from the expensive integral calculations f_1 and f_2. Since C_f is of the order of 1000 in practice, the integral calculations now dominate the total compute time, increasing the operation count by three orders of magnitude over the unfused form in Fig. 2.

A desirable solution would be somewhere in between the unfused structure of Fig. 2 (with maximal memory requirement and maximal reuse) and the fully fused structure of Fig. 3 (with minimal memory requirement and minimal reuse). This is shown in Fig. 4, where tiling and partial fusion of the loops is employed. The loops with indices a, e, c, f are tiled by splitting each of those indices into a pair of indices. The indices with a superscript t represent the tiling loops and the unsuperscripted indices now stand for intra-tile loops with a range of B, the block size used for tiling. For each tile (a^t, e^t, c^t, f^t), blocks of $T1$ and $T2$ of size B^2 are computed and used to form B^4 product contributions to the components of Y, which are stored in



array	space	time
X	B^4	V^4O^2
T1	B^2	$C_f(V/B)^2V^3O$
T2	B^2	$C_f(V/B)^2V^3O$
Y	B^4	V^5O
E	1	V^4

Figure 4. Use of tiling and partial fusion to reduce recomputation cost.

an array of size B^4.

As the tile size B is increased, the cost of function computation for f_1, f_2 decreases by a factor of B^2, due to the reuse enabled. However, the size of the needed temporary array for Y increases as B^4 (the space needed for X can be reduced back to a scalar by fusing its producer loop with the loop producing E, but Y's space requirement cannot be decreased). The maximum allowable size of B will be determined by memory capacity.

The above shows just one of many possible ways of introducing redundancy to trade-off recomputation cost and memory requirements. In [8] we provide a formalization and solution to the space-time trade-off problem encountered in this context. Besides the space-time trade-off optimization, a number of other compile-time optimization issues are addressed in the synthesis system, as outlined in the next section.

The computation considered here is just one component of the $A3A$ term, which in turn is only one of very many terms that must be computed. Although developers of quantum chemistry codes naturally recognize and perform some of these optimizations, a collective analysis of all these computations to determine their optimal implementation is beyond the scope of manual effort. Further, the time required to develop codes to implement such computational models is quite large, especially since the tensor contraction expressions can get quite complex — Fig. 5 shows an example of the kind of tensor expressions encountered when developing accurate computational models.

In the next section, we provide an overview of a transformation system that we are developing to aid quantum chemists in rapidly developing high-performance parallel codes for computations that they specify as a set of high-level tensor contractions.

4 Overview of the synthesis system

In this section, we briefly describe the basic structure of the synthesis system being developed. Some of these components are tightly coupled (for example, memory minimization and data distribution), and they are treated together as one combined module in the synthesis system.

High-level language: The input to the synthesis system is a sequence of tensor contraction expressions (essentially sum-of-products array expressions) together with declarations of index ranges of arrays, as shown in Fig. 6. This high-level notation provides essential information to the optimization components that would be difficult or impossible to extract out of low-level code.

Algebraic transformations: It takes input from the user in the form of tensor expressions and synthesizes an output computation sequence. The Algebraic Transformations module uses the properties of commutativity and associativity of addition and multiplication and the distributivity of multiplication over addition [20, 21]. It searches for all possible ways of applying these properties to an input sum-of-products expression, and determines a combination that results in an equivalent form of the computation with minimal operation cost.

Memory minimization: The operation-minimal computation sequence synthesized by the Algebraic Transformation module might require an excessive amount of memory due to the large temporary intermediate arrays involved. The Memory Minimization module attempts to perform loop fusion transformations to reduce the memory requirements [19]. This is done without any change to the number of arithmetic operations.

```
hbar[a,b,i,j] == sum[f[b,c] * t[i,j,a,c], c] - sum[f[k,c] * t[k,b] * t[i,j,a,c], k,c] + sum[f[a,c] * t[i,j,c,b], c] - sum[f[k,c] * t[k,a] * t[i,j,c,b], k,c] - sum[f[k,j] * t[i,k,a,b], k] - sum[f[k,c] * t[j,c] * t[i,k,a,b], k,c] - sum[f[k,i] * t[j,k,b,a], k] - sum[f[k,c] * t[i,c] * t[j,k,b,a], k,c] + sum[t[i,c] * t[j,d] * v[a,b,c,d], c,d] + sum[t[i,j,c,d] * v[a,b,c,d], c,d] + sum[t[j,c] * v[a,b,i,c], c] - sum[t[k,b] * v[a,k,i,j], k] + sum[t[i,c] * v[b,a,j,c], c] - sum[t[k,a] * v[b,k,j,i], k] - sum[t[k,d] * t[i,j,c,b] * v[k,a,c,d], k,c,d] - sum[t[i,c] * t[j,k,b,d] * v[k,a,c,d], k,c,d] - sum[t[j,c] * t[k,b] * v[k,a,c,i], k,c] + 2 * sum[t[j,k,b,c] * v[k,a,c,i], k,c] - sum[t[j,k,c,b] * v[k,a,c,i], k,c] - sum[t[i,c] * t[j,d] * t[k,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[k,d] * t[i,j,c,b] * v[k,a,d,c], k,c,d] - sum[t[k,b] * t[i,j,c,d] * v[k,a,d,c], k,c,d] - sum[t[j,d] * t[i,k,c,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[i,c] * t[j,k,b,d] * v[k,a,d,c], k,c,d] - sum[t[i,c] * t[j,k,d,b] * v[k,a,d,c], k,c,d] - sum[t[j,k,b,c] * v[k,a,i,c], k,c] - sum[t[i,c] * t[k,b] * v[k,a,j,c], k,c] - sum[t[i,k,c,b] * v[k,a,j,c], k,c] - sum[t[i,c] * t[j,d] * t[k,a] * v[k,b,c,d], k,c,d] - sum[t[k,d] * t[i,j,a,c] * v[k,b,c,d], k,c,d] - sum[t[k,a] * t[i,j,c,d] * v[k,b,c,d], k,c,d] + 2 * sum[t[j,d] * t[i,k,a,c] * v[k,b,c,d], k,c,d] - sum[t[j,d] * t[i,k,c,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[j,k,d,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[k,a] * v[k,b,c,j], k,c] + 2 * sum[t[i,k,a,c] * v[k,b,c,j], k,c] - sum[t[i,k,c,a] * v[k,b,c,j], k,c] + 2 * sum[t[k,d] * t[i,j,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,d] * t[i,k,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,c] * t[k,a] * v[k,b,i,c], k,c] - sum[t[j,k,c,a] * v[k,b,i,c], k,c] - sum[t[i,k,a,c] * v[k,b,j,c], k,c] + sum[t[i,c] * t[j,d] * t[k,a] * t[l,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[k,l,c,d], k,l,c,d] + sum[t[k,a] * t[l,b] * t[i,j,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,c,d], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,c,a] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,b] * t[j,k,d,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,c,d], k,l,c,d] + 4 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,a] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[j,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,j,c,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,c,b] * t[k,l,a,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,a,c] * t[k,l,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[j,c] * t[k,b] * t[l,a] * v[k,l,c,i], k,l,c] + sum[t[l,c] * t[j,k,b,a] * v[k,l,c,i], k,l,c] - 2 * sum[t[l,a] * t[j,k,b,c] * v[k,l,c,i], k,l,c] + sum[t[l,a] * t[j,k,c,b] * v[k,l,c,i], k,l,c] - 2 * sum[t[k,c] * t[j,l,b,a] * v[k,l,c,i], k,l,c] + sum[t[k,a] * t[j,l,b,c] * v[k,l,c,i], k,l,c] + sum[t[k,b] * t[j,l,c,a] * v[k,l,c,i], k,l,c] + sum[t[j,c] * t[l,k,a,b] * v[k,l,c,i], k,l,c] + sum[t[i,c] * t[k,a] * t[l,b] * v[k,l,c,j], k,l,c] + sum[t[l,c] * t[i,k,a,b] * v[k,l,c,j], k,l,c] - 2 * sum[t[l,b] * t[i,k,a,c] * v[k,l,c,j], k,l,c] + sum[t[l,b] * t[i,k,c,a] * v[k,l,c,j], k,l,c] + sum[t[i,c] * t[k,l,a,b] * v[k,l,c,j], k,l,c] + sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,a] * t[i,k,c,b] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,b] * t[j,l,d,a] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,d,c], k,l,c,d] + sum[t[k,a] * t[l,b] * v[k,l,i,j], k,l] + sum[t[k,l,a,b] * v[k,l,i,j], k,l] + sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[l,k,c,d], k,l,c,d] + sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,d,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,c,b] * t[k,l,a,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,a,c] * t[k,l,b,d] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[l,c] * t[i,k,a,b] * v[l,k,c,j], k,l,c] + sum[t[l,b] * t[i,k,a,c] * v[l,k,c,j], k,l,c] + sum[t[l,a] * t[i,k,c,b] * v[l,k,c,j], k,l,c] + v[a,b,i,j]
```

Figure 5. A tensor contraction expression from quantum chemistry.

Data distribution and partitioning: This component determines how best to partition the arrays among the processors of a parallel system. We assume a data-parallel model, where each operation in the operation sequence is distributed across the entire parallel machine. The arrays are to be disjointly partitioned between the physical memories of the processors. Since the data distribution pattern affects the memory usage on the parallel machine, this component is closely coupled with the memory minimization component [7].

Space-time transformation: If the Memory Minimization module is unable to reduce memory requirements of the computation sequence below the available disk capacity on the system, the computation is infeasible unless some space-time trade-off is utilized. If no satisfactory transformation is found, feedback is provided to the Memory Minimization module, causing it to seek a different solution [8]. If the Space-Time Transformation module is successful in bringing down the memory requirement below the disk capacity, the Data Locality Optimization module is invoked.

Data locality optimization: If the space requirements exceed physical memory capacity, portions of the arrays must be moved between disk and main memory as needed to maximize the reuse of elements in memory. The same considerations are involved in minimizing cache misses — blocks of data are moved between physical memory and the space available in the cache [9, 10].

Code generation: The back end of the synthesis system provides the output as pseudocode, FORTRAN or C code. The generated code can be either serial or parallel, using either MPI or the Global Arrays library. Depending on the circumstances, the synthesised code could also call highly-tuned, machine-specific Basic Linear Algebra Subprograms (BLAS) libraries, or optimized low-level functions from the existing quantum chemistry packages.

5 Code synthesized for motivating CCSD(T) example

For demonstrating the range of different loop structures for a given tensor contraction expression, consider the A3A term with explicit conversion from atomic to molecular orbitals; a high-level representation of this calculation is presented in the example input file in Fig. 6. Here the numbers of occupied and virtual orbitals are `O = 100` and `V = 3000`, respectively, and the number of atomic orbitals is assumed to be `N = O + V`. The atomic orbital indices are `la`, `mu`, `om`, and `nu`, and range from `1` to `N`; the virtual orbital indices, ranging from `1` to `V`, are denoted by `a`, `b`, `c`, `d`, `e`, and `f`, and the occupied orbital

```
range N = 3100;
range V = 3000;
range O = 100;

index la,mu,nu,om : N;
index a,b,c,d,e,f : V;
index i,j,k : O;

mlimit = 100GB;

function F(N,N,N,N);

procedure P(in Co[N,O], in Cv[N,V], in T[O,O,V,V], out E)=
begin
  E==sum[
       sum[
         sum[sum[sum[sum[F(mu,nu,om,la)*Co[la,k],{la}]*Cv[om,b],{om}]*Cv[nu,f],{nu}]*Cv[mu,a],{mu}]
         *
         sum[sum[sum[sum[F(mu,nu,om,la)*Co[la,k],{la}]*Cv[om,b],{om}]*Cv[nu,e],{nu}]*Cv[mu,c],{mu}],
         {b,k}]
       *
       sum[T[i,j,a,e]*T[i,j,c,f],{i,j}],
       {a,e,c,f}];
end
```

Figure 6. Input file for the A3A term with conversion from atomic to molecular orbitals.

indices, ranging from `1` to `O`, are `i`, `j`, and `k`.

The `mlimit` declaration specifies the space limit allowed for the arrays involved in the calculation. The `function` declaration sets parameters of functions evaluations; here, `F` denotes evaluation of the N^4 atomic integrals. The `procedure` declaration specifies the input and output arrays in the calculation, and the computation to be performed. In Fig. 6, the input arrays are the t amplitudes (represented by `T`), and the atomic to molecular transformation matrix `C(N,N)`, which is broken up, for convenience, in two sub-matrices: `Co(N,O)` and `Cv(N,V)`. The output of the computation is the scalar `E`, and it is calculated according to the given formula.

The code output of the synthesis tool depends on the parameters — number of orbitals, space limit — specified in the input file. The same high-level representation results in different code structures, tailored to specific problem sizes and machine capabilities. For example, Fig. 7 shows pseudocode generated for the case when the molecular integrals $\langle cb \parallel ek \rangle$ and $\langle ab \parallel fk \rangle$ fit on disk. Here, the `readF` statements represent the disk read operations for $\langle cb \parallel ek \rangle$ and $\langle ab \parallel fk \rangle$, and `tmp`$\_n$ arrays denote intermediate arrays. The transformation of the integrals from the atomic to the molecular orbital basis is decoupled from the actual CCSD(T) computation shown in Fig. 7. The four-step transformation through the arrays `C` is performed once, and its results stored on disk.

```
E == 0.0;
for aT, cT, eT, fT
  for fI, eI, cI, aI
    tmp_3[aI,cI,eI,fI] == 0.0;
  for b, k
    for fI, aI
      tmp_1[aI,fI] == readF((aT + aI),b,(fT + fI),k);
    for eI, cI
      tmp_2[eI,cI] == readF((cT + cI),(eT + eI),b,k);
    for fI, eI, cI, aI
      tmp_3[aI,cI,eI,fI] += tmp_2[eI,cI] * tmp_1[aI,fI];
  for fI, eI, cI, aI
    tmp_4 == 0.0;
    for i, j
      tmp_4 += T[i,j,(cT + cI),f] * T[i,j,a,e];
    E += tmp_4 * tmp_3[aI,cI,eI,fI];
return E;
```

Figure 7. Example of loop structure generated by the synthesis tool for the disk-resident array case.

```
E = 0.0;
for aT, fT
  for kT
    for b, aI, fI, kI
      tmp_5[b,kI,aI,fI] == 0.0;
    for mu
      for b, fI, kI
        tmp_4[b,fI,kI] == 0.0;
      for nu
        for b, kI
          tmp_3[b,kI] == 0.0;
        for om
          for kI
            tmp_2[kI] == 0.0;
          for la
            tmp_1 == F(mu,nu,om,la);
            for kI
              tmp_2[kI] += Co[la,(kT + kI)] * tmp_1;
          for b, kI
            tmp_3[b,kI] += Cv[om,b] * tmp_2[kI];
        for b, fI, kI
          tmp_4[b,fI,kI] += Cv[nu,(fT + fI)] * tmp_3[b,kI];
      for b, aI, fI, kI
        tmp_5[b,kI,aI,fI] += Cv[mu,(aT + aI)] * tmp_4[b,fI,kI];
  for eT
    for c, aI, eI, fI
      tmp_11[c,aI,eI,fI] == 0.0;
    for kT
      for b, c, eI, kI
        tmp_10[b,c,eI,kI] == 0.0;
      for mu
        for b, eI, kI
          tmp_9[b,eI,kI] == 0.0;
        for nu
          for b, kI
            tmp_8[b,kI] == 0.0;
          for om
            for kI
              tmp_7[kI] == 0.0;
            for la
              tmp_6 == F(mu,nu,om,la);
              for kI
                tmp_7[kI] += Co[la,(kT + kI)] * tmp_6;
            for b, kI
              tmp_8[b,kI] += Cv[om,b] * tmp_7[kI];
          for b, eI, kI
            tmp_9[b,eI,kI] += Cv[nu,(eT + eI)] * tmp_8[b,kI];
        for b, c, eI, kI
          tmp_10[b,c,eI,kI] += Cv[mu,c] * tmp_9[b,eI,kI];
      for b, c, aI, eI, fI, kI
        tmp_11[c,aI,eI,fI] += tmp_10[b,c,eI,kI] * tmp_5[b,kI,aI,fI];
    for c, fI, eI, aI
      tmp_12 == 0.0;
      for i, j
        tmp_12 += T[i,j,c,(fT + fI)] * T[i,j,(aT + aI),(eT + eI)];
      E += tmp_12 * tmp_11[c,aI,eI,fI];
return E;
```

Figure 8. Example of loop structure generated by the synthesis tool for the case when molecular orbital integrals cannot be stored on disk.

Figure 8 shows the more interesting case when the input parameters are such that the molecular integrals are too large to fit on disk. Then repeated calculation of these integrals becomes necessary, resulting in repeated calculation of the atomic integrals `F(mu,nu,om,la)`. The synthesis tool analyzes several hundred different code structures that allow the evaluation on-the-fly of the molecular integrals $\langle cb \parallel ek \rangle$ and $\langle ab \parallel fk \rangle$. For each code structure, the ranges of the tiling and intra-tile loops (*i.e.* the "block sizes") are optimized so that the recomputation cost is minimized, without exceeding the memory space limit.

Typically, the code structure and the block sizes cannot be easily determined by the human developer. Evaluating hundreds of different possibilities, whose recomputation costs may vary by a few orders of magnitude, is tedious at best. Moreover, the entire analysis depends on the problem and machine parameters; different parameters result in different final code structures for the same high-level input expression. The example input expression in Fig. 6, which should be computed in different ways depending on the size of the molecular integrals relative to the disk size, illustrates the power of an automated synthesis tool approach.

6 Related work and discussion

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 [28, 15, 35]. 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 [12], the telescoping languages project [16], ATLAS [34] for deriving efficient implementation of BLAS routines, the PHIPAC [3] project, and the TUNE project [33]. 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 [11, 24, 25, 26].

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. While optimization of performance is a significant goal, more important in our context is the potential for dramatically reducing the developmental effort required of a quantum chemist to develop a new computational model. Currently, the manual development and testing of a reasonably efficient parallel code for a computational model such as the coupled cluster model typically takes months to years for a computational chemist. We aim to reduce the time to prototype a new model to under a day, through use of the synthesis system being developed. We are at present in the testing stages of the synthesis tool, and we expect to soon have a first release available for the computational chemistry community.

References

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, Yuan-Jye J. Wu. PLAPACK: Parallel Linear Algebra Package. In *Proc. of the SIAM Parallel Processing Conference*, 1997.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. *PETSc Users Manual*. Technical Report ANL-95/11 — Revision 2.1.2, Argonne National Laboratory, 2002.

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

[4] D. L. Brown, William D. Henshaw and Daniel J. Quinlan. Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids. In *Proc. SIAM conference on Object Oriented Methods for Scientfic Computing*, UCRL-JC-132017, 1999.

[5] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification*. CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[6] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers — Design Issues and Performance*. Technical Report CS-95-283, University of Tennessee, Knoxville, Mar. 1995.

[7] D. Cociorva, G. Baumgartner, C.-C. Lam, P. Sadayappan, J. Ramanujam. Memory-Constrained Communication Minimization for a Class of Array Computations. In *Proc. of the 15th International Workshop on Languages and Compilers for Parallel Computing*, Jul. 2002.

[8] D. Cociorva, G. Baumgartner, C.-C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Jun. 2002, pp. 177–186.

[9] 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. *Proc. of the Intl. Conf. on High Performance Computing*, Dec. 2001, Lecture Notes in Computer Science, Vol. 2228, pp. 237–248, Springer-Verlag, 2001.

[10] D. Cociorva, J. Wilkins, C.-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*, Jun. 2001, pp. 103–113.

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

[12] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. ICASSP 98*, Vol. 3, pp. 1381–1384, 1998, `http://www.fftw.org`.

[13] Samuel Guyer and Calvin Lin. Broadway: A Software Architecture for Scientific Computing. In *The Architecture of Scientific Software*, R.F. Boisvert and P.T.P. Tang (eds.), Kluwer Academic Press, 2000, pp. 175–192.

[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] 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.

[16] 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.

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

[18] C.-C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-Optimal Evaluation of Expression Trees Involving Large Objects. In *Intl. Conf. on High Performance Computing*, Dec. 1999, Lecture Notes in Computer Science, Vol. 1745, Springer-Verlag, 1999.

[19] 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*, Aug. 1999, Lecture Notes in Computer Science, Vol. 1863, Springer-Verlag, 1999.

[20] C.-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.

[21] C.-C. Lam, P. Sadayappan, and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. In *8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[22] 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*, pp. 47–109, Kluwer Academic, 1997.

[23] 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.

[24] The Match Project. *A MATLAB Compilation Environment for Distributed Heterogeneous Adaptive Computing Systems*. `www.ece.nwu.edu/cpdc/Match/Match.html`.

[25] Vijay Menon, Keshav Pingali. High-Level Semantic Optimization of Numerical Codes. In *Proc. ACM Intl. Conf. on Supercomputing*, 1999.

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

[27] D. Mirkovic and L. Johnsson. Automatic Performance Tuning in the UHFFT Library. In *Proc. International Conference on Computational Science*, Lecture Notes in Computer Science, Vol. 2073, pp. 71–80, Springer-Verlag, 2001.

[28] J. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Puschel, and M. Veloso. *SPIRAL: Portable Library of Optimized Signal Processing Algorithms*, 1998. `http://www.ece.cmu.edu/~spiral`.

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

[30] R.W. Numrich and J.K. Reid. Co-Array Fortran for Parallel Programming. *Fortran Forum*, Vol. 17, No. 2, 1998.

[31] Matei Ripeanu and Adriana Iamnitchi. Cactus application: Performance predictions in grid environments. In *Proc. EuroPar 2001*, Lecture Notes in Computer Science, Springer-Verlag, 2001.

[32] L. Snyder. *A Programmer's Guide to ZPL*. The MIT Press, Mar. 1999.

[33] M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's Matrix Multiplication for Memory Hierarchies. In *Proc. Supercomputing '98*, 1998

[34] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing '98*, 1998.

[35] 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.

[36] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect, *Concurrency: Practice and Experience*, Vol. 10, No. 11–13, Sept.-Nov. 1998.