Synthesis of High-Performance Parallel Programs for a Class of *Ab Initio* Quantum Chemistry Models

GERALD BAUMGARTNER, ALEXANDER AUER, DAVID E. BERNHOLDT, ALINA BIBIREATA,
VENKATESH CHOPPELLA, DANIEL COCIORVA, XIAOYANG GAO, ROBERT J. HARRISON,
SO HIRATA, SRIRAM KRISHNAMOORTHY, SANDHYA KRISHNAN, CHI-CHUNG LAM,
QINGDA LU, MARCEL NOOIJEN, RUSSELL M. PITZER, J. RAMANUJAM, P. SADAYAPPAN,
AND ALEXANDER SIBIRYAKOV

Invited Paper

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.

Keywords—*Communication minimization, compiler optimizations, data locality optimization, domain-specific languages, high-level programming languages, memory-constrained optimization, tensor contraction expressions.*

G. Baumgartner is with the Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803 USA (e-mail: gb@csc.lsu.edu).

A. Auer is with the Institut für Chemie, Technische Universität Chemnitz, Chemnitz D-09111, Germany (e-mail: alexander.auer@chemie.tu-chemnitz.de).

D. E. Bernholdt, V. Choppella, and R. J. Harrison are with Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA (e-mail: bernholdtde@ornl.gov; choppellav@ornl.gov; harrisonrj@ornl.gov).

A. Bibireata, D. Cociorva, X. Gao, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, and P. Sadayappan are with the Department of Computer Science and Engineering, Ohio State University, Columbus, OH 43210 USA (e-mail: bibireat@cse.ohio-state.edu; cociorva@cse.ohio-state.edu; gaox@cse.ohio-state.edu; krishnsr@cse.ohio-state.edu; krishnas@cse.ohio-state.edu; clam@cse.ohio-state.edu; luq@cse.ohio-state.edu; saday@cse.ohio-state.edu; sibiryak@cse.ohio-state.edu).

S. Hirata is with the Quantum Theory Project, University of Florida, Gainesville, FL 32611 USA (e-mail: hirata@qtp.ufl.edu).

M. Nooijen is with the Department of Chemistry, University of Waterloo, Waterloo, ON N2L 6B4, Canada (e-mail: nooijen@uwaterloo.ca).

R. M. Pitzer is with the Department of Chemistry, Ohio State University, Columbus, OH 43210 USA (e-mail: pitzer.3@osu.edu).

J. Ramanujam is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803 USA (e-mail: jxr@ece.lsu.edu).

A. Sibiryakov is with Microsoft, Redmond, WA 98052 USA.

Digital Object Identifier 10.1109/JPROC.2004.840311

I. INTRODUCTION

This paper provides 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. Fig. 1 shows a tensor contraction expression for one of the terms in the coupled cluster model [43], [47] for *ab initio* electronic structure modeling. An optimized parallel Message Passing Interface (MPI) program to implement such an expression containing a large number of tensor products typically requires several thousands of lines of code.

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]

Fig. 1. CCSD doubles expression from quantum chemistry.

The computational domain that we consider is also 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 such as ACES II [69],[1] GAMESS [63], Gaussian [15], NWChem [22], PSI [24], and MOLPRO [71].[2] These structures comprise the bulk of the computation with the coupled cluster approach to the accurate description of the electronic structure of atoms and molecules [43], [47]. 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.

II. 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 multidimensional

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:

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

where typical index ranges are on the order of tens to a few thousands. If this expression is directly translated to code (with ten nested loops, for indexes $a - l$), 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. 2(a) and can be directly translated into code as shown in Fig. 2(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 indexes $a - d$ are the largest, while the extents along indexes $i - l$ are the smallest. Therefore, the size of temporary array $T1$ would dominate the total memory requirement.

Thus, 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

$$T1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel}$$

$$T2_{bcjk} = \sum_{df} T1_{bcdf} \times C_{dfjk}$$

$$S_{abij} = \sum_{ck} T2_{bcjk} \times A_{acik}$$

(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)

Fig. 2. Example illustrating use of loop fusion for memory reduction.

"operation-minimal" form may be too large to even fit on disk.

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

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. Enumerating all fusion choices to find the loop structure that minimizes the memory requirements is beyond the scope of existing compiler techniques, as discussed in Section X.

III. EXAMPLE

In this section, an example from quantum chemistry is used to illustrate issues pertinent to the synthesis system. We discuss a component of the so-called CCSD(T) calculation [58], one of the most computationally intensive components of many quantum chemistry packages. It is a coupled cluster approximation that includes single and double excitations from the Hartree–Fock wave function 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 5 cm^{-1}. This level of accuracy is adequate to answer many of the questions that arise in studies of chemical systems.

The following representative equation arises in the Laplace factorized expression for linear triples perturbation correction:

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

```
for a, e, c, f
  [ for i, j
  [   [ Xaecf += Tijae Tijcf
for c, e, b, k
  [ T1cebk = f1(c, e, b, k)
for a, f, b, k
  [ T2afbk = f2(a, f, b, k)
for c, e, a, f
  [ for b, k
  [   [ Yceaf += T1cebk T2afbk
for c, e, a, f
  [ E += Xaecf Yceaf
```

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

Fig. 3. Unfused operation-minimal form.

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, where indexes that occur twice in a term are implicitly summed over.

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 $\langle \mu\nu \mid \omega\lambda \rangle = \int \int dr^3 ds^3 \phi_\mu(\underline{r})\phi_\nu(\underline{s})|\underline{r} - \underline{s}|^{-1}\phi_\omega(\underline{r})\phi_\lambda(\underline{s})$ 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 indexes i, j, k, l, m, n refer to occupied orbitals, of number O between 30 and 100. The indexes 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 (MO) basis, but must be computed in the underlying atom-centered Gaussian basis, and transformed to the MO basis. We omit these details in our 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. 3 shows pseudocode 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.[3] The intermediate quantities X_{aecf} are computed by contracting over (i.e., summing over products of) input array T, while

---

[3]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 tradeoff problem addressed by the synthesis system
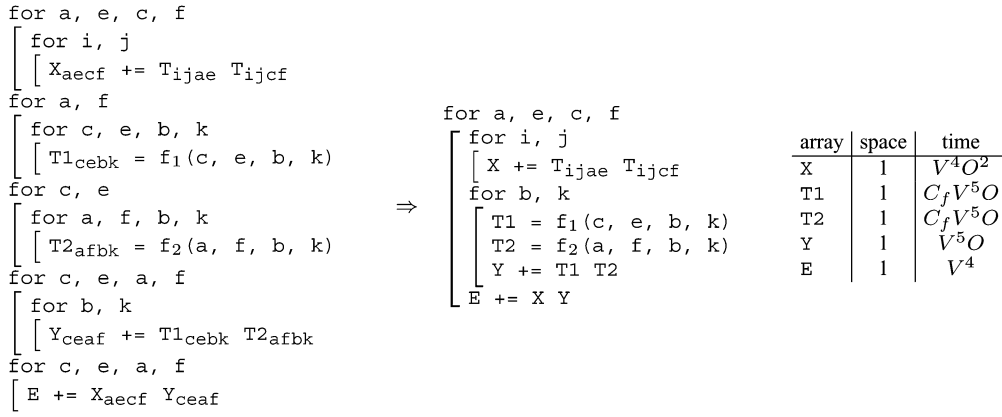
```
for a, e, c, f
   for i, j
      Xaecf += Tijae Tijcf
for a, f
   for c, e, b, k
      T1cebk = f1(c, e, b, k)
   for c, e
      for a, f, b, k
         T2afbk = f2(a, f, b, k)
   for c, e, a, f
      for b, k
         Yceaf += T1cebk T2afbk
   for c, e, a, f
      E += Xaecf Yceaf
```

$\Rightarrow$

```
for a, e, c, f
   for i, j
      X += Tijae Tijcf
   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	$C_f V^5 O$
T2	1	$C_f V^5 O$
Y	1	$V^5 O$
E	1	V^4

Fig. 4. Use of redundant computation to allow full fusion.

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 $O(OV^3)$ pairwise 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 pseudocode form shown in Fig. 3 is computationally very efficient in minimizing the 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. 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. Fusing together pairs of producer–consumer loops in the computation may result in a reduction of the array sizes: given a pair of loops, storage may be eliminated along the dimension over which the common indexes iterate. It can be seen that the loop that produces X (with indexes a, e, c, f), the loop that produces Y (with indexes c, e, a, f) and the loop that consumes X and Y to produce E (with indexes c, e, a, f) can all be fully fused together (after some loop permutation to make the nesting order match between the producer loop and consumer loop), permitting the elimination of all explicit indexes in X and Y to reduce them to scalars. Thus, the largest intermediates, of size $O(V^4)$, can be reduced to scalars with loop fusion. However, the loops producing $T1$ (with indexes c, e, b, k) and $T2$ (with indexes a, f, b, k) cannot both also be directly fused with the other three loops because their indexes do not match.

Fig. 4 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 indexes a, f for $T1$ and c, e for $T2$. Now all five loops have common indexes a, e, c, f that can be fused, permitting elimination of those indexes from all temporaries. Further, by fusing the producer loops for $T1$ and $T2$ with their consumer loop, which produces Y, the b, k indexes 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
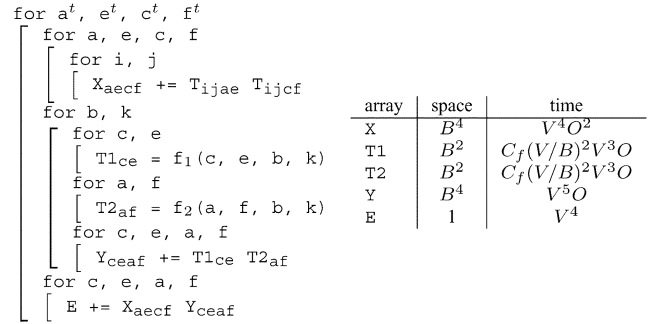
```
for at, et, ct, ft
   for a, e, c, f
      for i, j
         Xaecf += Tijae Tijcf
   for b, k
      for c, e
         T1ce = f1(c, e, b, k)
      for a, f
         T2af = f2(a, f, b, k)
      for c, e, a, f
         Yceaf += T1ce T2af
   for c, e, a, f
      E += Xaecf Yceaf
```

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

Fig. 5. Use of tiling and partial fusion to reduce recomputation cost.

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. 3.

A desirable solution would be somewhere in between the unfused structure of Fig. 3 (with maximal memory requirement and maximal reuse) and the fully fused structure of Fig. 4 (with minimal memory requirement and minimal reuse). We show such a solution in Fig. 5, where tiling and partial fusion of the loops is employed. The loops with indexes a, e, c, f are tiled by splitting each of those indexes into a pair of indexes. The indexes with a superscript t represent the tiling loops and the unsuperscripted indexes now stand for intratile 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 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). When B^4 becomes larger than the size of physical memory, expensive paging in and out of disk will be required for Y. Further, there are diminishing returns on reuse of $T1$
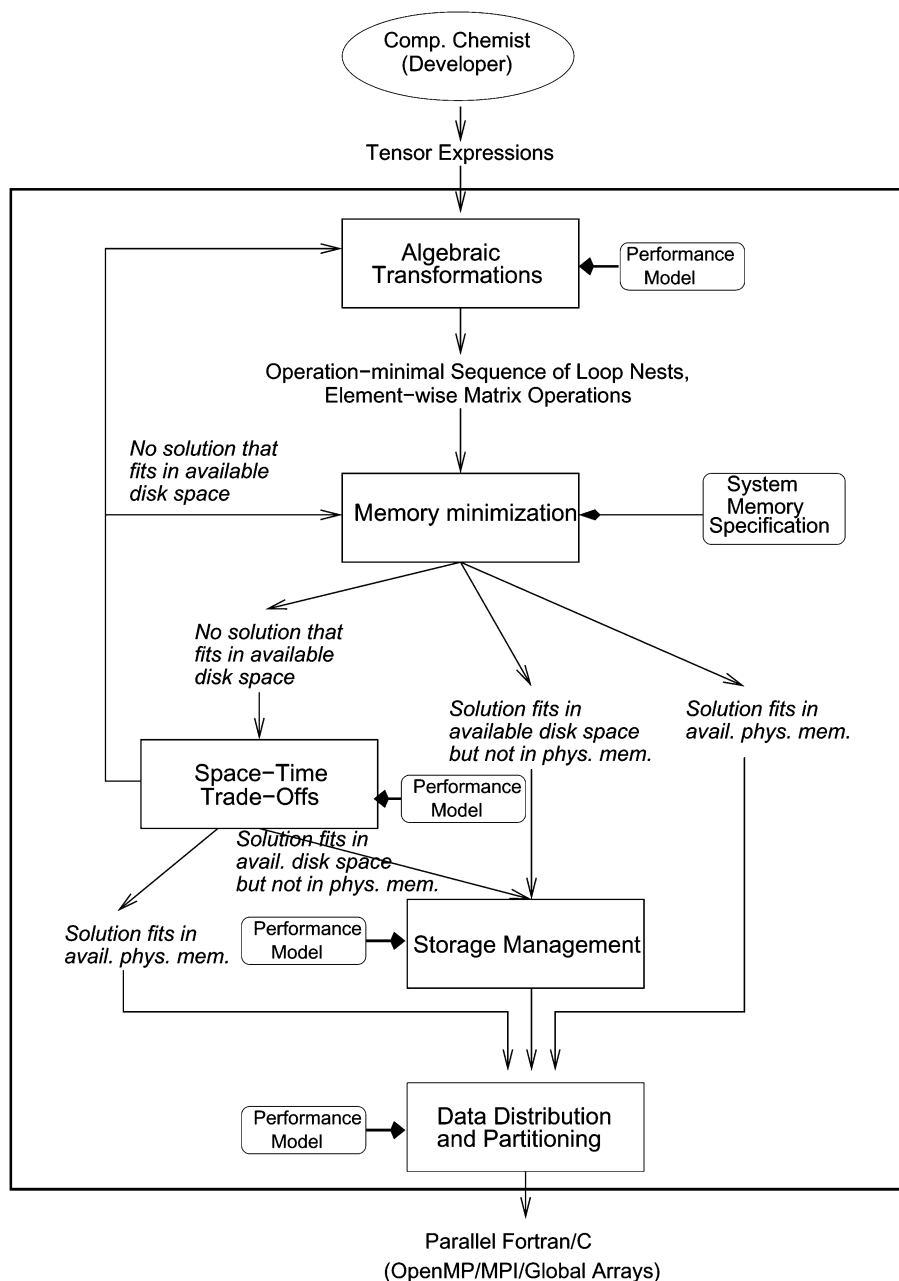
Fig. 6. Tensor Contraction Engine.

and $T2$ after B^2 becomes comparable to C_f, since the loop producing Y now becomes the dominant one. So we can expect that as B is increased, performance will improve and then level off and then deteriorate. The optimum value of B will clearly depend on the cost of access at the various levels of the memory hierarchy.

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 expressions can get quite complex—such as the one shown earlier in Fig. 1.

IV. OVERVIEW OF THE SYNTHESIS SYSTEM

Fig. 6 shows the components of the Tensor Contraction Engine (TCE), the synthesis system being developed. We present in this section a brief description of the basic components. 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 (essen-

tially sum-of-products array expressions) together with declarations of index ranges and symmetry and sparsity of matrices. 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: Input from the user in the form of tensor expressions is transformed into a computation sequence. 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.

Memory minimization: The operation-minimal computation sequence synthesized by applying algebraic transformation might require an excessive amount of memory due to the need to use large temporary intermediate arrays. The memory minimization step seeks to perform loop fusion transformations to reduce the memory requirements. This is done without incurring any increase on the number of arithmetic operations.

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.

Space–time transformation: If the memory minimization step is unable to reduce memory requirements of the computation sequence below the available disk capacity on the system, the computation is infeasible unless a space–time tradeoff is performed. If no satisfactory transformation is found, feedback is provided to the memory minimization module, causing it to seek a different solution. 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 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. The same considerations are involved in minimizing cache misses—blocks of data are moved between physical memory and the space available in the cache.

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 MPI or Global Arrays (GA) [54], [55]. Depending on the circumstances, the synthesized 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.

In Sections V–XI, we provide some details about the optimizations implemented in some of these modules.

V. OPERATION MINIMIZATION

The operation minimization problem encountered in this context 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 pairwise 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 pairwise products. For the example of Fig. 2, 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.

The problem of determining the operator tree with minimal operation count is NP-complete, and an efficient pruning search procedure has been developed [40], [41]. Given a sum-of-products term, the following procedure can be used to exhaustively enumerate all valid sequences of matrix summation or matrix product (as defined below; different from the standard notion of matrix-matrix product in linear algebra) operations to compute it.

1) Let X_a denote the ath product term in the given sum-of-products expression and X_a.dimens the set of index variables in $X_a[\ldots]$. Set r to zero.
2) Increment r. Then, perform either action:
   a) Write a product formula $f_r[\ldots] = X_a[\ldots] \times X_b[\ldots]$ where $X_a[\ldots]$ and $X_b[\ldots]$ are any two terms in the pool. The indexes for f_r are f_r.dimens $= X_a$.dimens $\cup X_b$.dimens. Replace $X_a[\ldots]$ and $X_b[\ldots]$ from the pool by $f_r[\ldots]$.
   b) If there exists an summation index (say i) that appears in exactly one term (say $X_a[\ldots]$) in the list, increment r and create a summation formula $f_r[\ldots] = \sum_i X_a[\ldots]$ where f_r.dimens $= X_a$.dimens $- \{i\}$. Replace $X_a[\ldots]$ in the pool by $f_r[\ldots]$.
3) When step 2 cannot be performed any more, a valid formula sequence is obtained. To obtain all valid sequences, exhaust all alternatives in step 2 using depth-first search.

The enumeration procedure above is inefficient in that a particular formula sequence may be generated more than once in the search process. This can be avoided by creating an ordering among the product terms and the intermediate generated functions (which can be treated as new terms, numbered in increasing order as they are generated).

A further reduction in the cost of the search procedure can be achieved by pruning the search space by use of the following two rules.

1) When a summation index appears in only one term, perform the summation over that index immediately, without considering any other possibilities at that step.
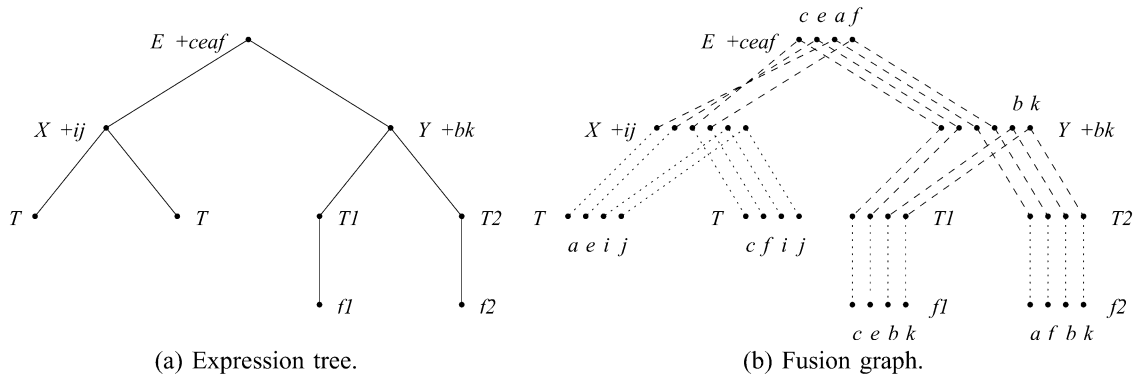
(a) Expression tree.　　　　(b) Fusion graph.

Fig. 7. Expression tree and fusion graph for unfused operation-minimal form of loop in Fig. 3.

2) If two or more terms have exactly the same set of indexes, first multiply them together before considering any other possibilities.

Although the problem of determining minimal operation count for a sum-of-products expression is NP-complete, the pruning search procedure above works very effectively in practice, since the number of nested loops and number of product terms is typically less than ten.

VI. Memory Minimization and Space–Time Tradeoffs

As discussed in Section II, the operation minimization procedure often results in the creation of intermediate temporary arrays. For typical computations in computational chemistry, the space required for storing these temporary arrays can be several terabytes, which makes the computation impractical. As shown in Fig. 2(c), the problem with memory requirements of large intermediate arrays can be mitigated through loop fusion. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array that 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, lowers the memory requirement. The use of loop fusion can be seen to result in significant potential reduction to the total memory requirement. For a computation composed of a number of nested loops, there will generally be a number of fusion choices that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost.

A. Fusion Graph

For facilitating the enumeration of all possible loop fusion configurations for a given expression tree, we define a data structure we call a *fusion graph* [37], [39]. The fusion graph makes the indexes of nodes in the expression tree explicit and indicates the scopes of fused loops.

Let T be an expression tree. For any given node $v \in T$, let subtree(v) be the set of nodes in the subtree rooted at v, parent(v) be the parent node of v, and indexes(v) be the set of loop indexes for v (including the summation indexes

sumindexes(v) if v is a summation node). A *fusion graph* for T is constructed as follows.

1) Corresponding to each node v in T a fusion graph contains a set of vertices, one for each index $i \in$ indexes(v).
2) For each Array or Const node v in T and for each index $i \in$ indexes(parent(v)) − indexes(v) that is fused between v and its parent, an i-index is added to the set of nodes corresponding to v.
3) For each loop of index i that is fused between a node and its parent, the i-vertices for the two nodes are connected with a *fusion edge*.
4) For each index i that is shared between a node and its parent, for which the corresponding loops are not fused, the i-vertices for the two nodes are connected with a *potential fusion edge*.

Fig. 7(a) shows the expression tree corresponding to the computation in Fig. 3. Fig. 7(b) shows the fusion graph for the unfused form of this computation. Corresponding to each node in a expression tree, the fusion graph has a set of vertices corresponding to the loop indexes of the node of the expression tree. The potential for fusion of a common loop among a producer–consumer pair of loop nests is indicated in the fusion graph through a dashed *potential fusion* edge connecting the corresponding vertices. Leaf nodes in the fusion graph correspond to input arrays or primitive function evaluations and do not represent a loop nest. The edges from the leaves to their parents are shown as dotted edges and do not affect the fusion possibilities.

If a pair of loop nests is fused using one or more common loops, it is captured in the fusion graph by changing the dashed potential-fusion edges to continuous fusion edges. If more than two loop nests are fused together, a chain of fusion edges results, called a *fusion chain*. The *scope of a fusion chain* is the set of nodes it spans. The fusion graph allows us to characterize the condition for feasibility of a particular combination of fusions: the scope of any two fusion chains in a fusion graph must either be disjoint or a subset/superset of each other. Scopes of fusion chains do not partially overlap because loops do not (i.e., loops must be either separate or nested).

The fusion graph in Fig. 7(b) can be used to determine the fusion possibilities. On the left side of the graph, the edges
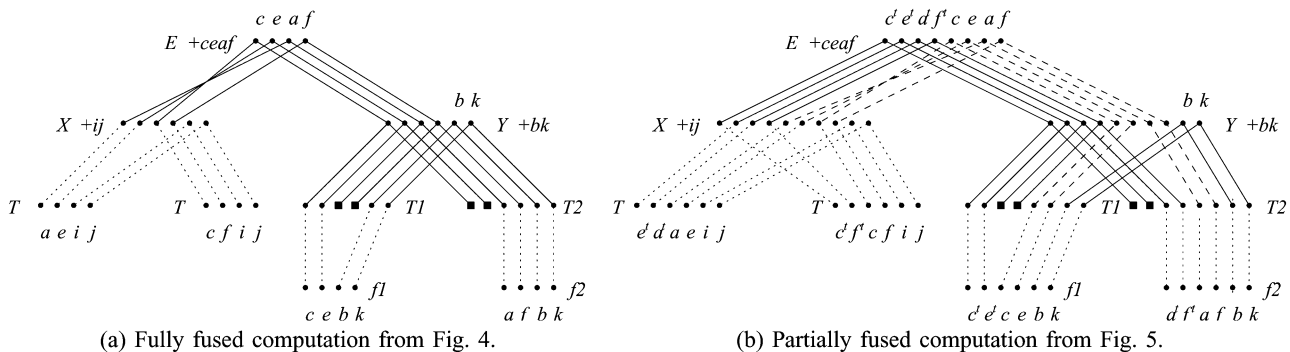
(a) Fully fused computation from Fig. 4.   (b) Partially fused computation from Fig. 5.

Fig. 8. Fusion graphs showing redundant computation and tiling.

corresponding to (a, e, c, f) can all be made fusion edges, suggesting that complete fusion is possible for the loop nests producing and consuming X, reducing it to a scalar. Similarly, on the right side of the graph, the edges corresponding to (c, e, a, f) can also be made fusion edges, reducing Y to a scalar. Further, by creating fusion edges for indexes (c, e), the producer loop for $T1$ can be fully fused with the Y loop that consumes it. However, now the producer loop for $T2$ cannot be fused, since the addition of any fusion edge (say, for index a) will result in partially overlapping fusion chains for a and (c, e).

The fully fused version from Fig. 4 can be represented graphically as shown in Fig. 8(a). Additional vertices have been added for indexes (c, e) and (a, f), respectively, at the nodes corresponding to the producer loops for $T1$ and $T2$. Now, complete fusion chains can be created without any partial overlap in the scopes of the fusion chains. From the figure, it can be seen that in fact the redundant computation need only be added to one of $T1$ or $T2$ to achieve complete fusion. For example, removing the additional vertices for (a, f) at $T2$ does not violate the nonpartial-overlap condition for fusion.

The fusion graph was used to develop an algorithm [37], [39] to determine the combination of fusions that minimizes the total storage required for all the temporary intermediate arrays. A bottom-up dynamic programming approach was used that maintains a set of pareto-optimal fusion configurations at each node, merging solutions from children nodes to generate the optimal configurations at a parent. The two metrics used are the total memory required under the subtree rooted at the node, and the constraints imposed by a configuration on fusion further up the tree. A configuration is inferior to another if it is *more or equally constraining* with respect to further fusions than the other, and uses no less memory. At the root of the tree, the configuration with the lowest memory requirement is chosen.

The complexity of the algorithm is exponential in the number of index variables and the number of solutions could in theory grow exponentially with the size of the expression tree. However, since in practical applications the number of dimensions of tensors and the number of tensors in a term are in the single digits, the number of index variables remain small enough and the pruning is effective in keeping the size of the solution set at each node manageable. If needed,

heuristics can be used to further reduce the size of solution sets.

If the storage requirements still exceed the disk capacity after memory minimization, we can choose to recompute some (parts of) temporary arrays in order to further reduce the space requirements. We have developed a space–time tradeoff algorithm [9] that employs a combination of fusion and tiling to achieve a good balance between recomputation and memory usage. The first step of the space–time tradeoff algorithm uses a dynamic programming approach similar to the memory minimization algorithm that maintains a set of pareto-optimal fusion/recomputation configurations, in which the recomputation cost is used as a third metric. Solutions exceeding the memory limit are pruned out. The result of the search is a set of loop structures with different combinations of space requirements and recomputation cost.

In the second step of the algorithm, recomputation indexes are split into tiling and intratile loop pairs, as shown in Fig. 8(b). By making intratile loops the innermost loops, any recomputation only needs to be performed once per iteration of the tiling loop in exchange for increasing the storage requirements for temporaries in which the dimension corresponding to the tiled loop had been eliminated. For each solution from the first step of the algorithm, we then search for tile sizes that minimize the recomputation cost, and take the solution that results in the lowest recomputation cost.

VII. Data Locality Optimization

Once a solution is found that fits onto disk, we optimize the data locality to reduce memory and disk access times. We developed algorithms [7], [8] that, given a memory-reduced (fused) version of the code, find the appropriate blocking of the loops in order to maximize data reuse. These algorithms can be applied at different levels of the memory hierarchy—for example, to minimize data transfer between main memory and disk (disk access minimization) or to minimize data transfer between main memory and the cache (cache optimization). In this section, we briefly describe the main points of our algorithm [8], focusing mostly on the cache management problem. For the disk access minimization problem, the same approach is used, replacing the cache size by the physical memory size.

We introduce a memory access cost model (*Cost*), an estimate on the number of cache misses, as a function of tile

sizes and loop bounds. In a bottom-up traversal of the abstract syntax tree, we count for each loop the number (*Accesses*) of distinct array elements accessed in its scope. If this number is smaller than the number of elements that fit into the cache, then *Cost = Accesses*. Otherwise, it means that the elements in the cache are not reused from one loop iteration to the next, and the cost is obtained by multiplying the loop range by the cost of its inner loop(s).

Using this cost model, we can compute the total memory access cost for given tile sizes. The procedure is repeated for different sets of tile sizes, and new costs are computed. In the end the lowest possible cost is chosen, thus determining the optimal tile sizes. We define our tile size search space in the following way: if N_i is a loop range, we use a tile size starting from $T_i = 1$ (no tiling), and successively increasing T_i by doubling it until it reaches N_i. This approach ensures a slow (logarithmic) growth of the search space with increasing array dimension for large N_i. If N_i is small enough, an exhaustive search is performed instead.

VIII. PARALLELISM: DATA PARTITIONING AND COMMUNICATION MINIMIZATION

Given a sequence of formulas, we need to find an effective partitioning of arrays and operations among the processors and a choice of loop fusions in order to minimize interprocessor communication, while staying within the available memory in implementing the computation on a message-passing parallel computer.

Since primitive tensor contractions are essentially generalized multidimensional matrix multiplications, we use a generalized form of the memory efficient Cannon algorithm [4], [36]. A logical view of the P processors as a 2-D $\sqrt{P} \times \sqrt{P}$ grid is used, and each array is fully distributed along the two processor dimensions. We use a pair of indexes to denote the partitioning or *distribution* of the elements of a data array on a 2-D processor array. The dth position in a pair α, denoted $\alpha[d]$, where d can be either one or two, corresponds to the dth processor dimension. Each position is an index variable distributed along that processor dimension. As an example, suppose 16 processors form a 2-D 4×4 logical array. For the array $B(b, e, f, l)$ in Fig. 2(a), the pair $\alpha = \langle b, f \rangle$ specifies that the first (b) and the third (f) dimensions of B are distributed along the first and second processor dimensions respectively, and that the second (e) and fourth (l) dimensions of B are not distributed. Thus, a processor whose id is P_{z_1, z_2}, with z_1 and z_2 between one and four, will be assigned a portion of B specified by $B(\text{myrange}(z_1, N_b, 4), 1 : N_e, \text{myrange}(z_2, N_e, 4), 1 : N_l)$, where $\text{myrange}(z, N, p)$ is the range $(z - 1) \times N/p + 1$ to $z \times N/p$.

A tensor contraction formula can be expressed as a generalized matrix multiplication $C(I, J) + = A(I, K) * B(K, J)$, where I, J, and K represent index collections, or index sets. This observation follows from a special property of tensor contractions: all the indexes appearing on the left-hand side must appear on the right-hand side only *once* (index sets I and J, for A and B, respectively), and all summation indexes must appear on both right-hand side arrays (index set K). For example, the tensor contraction $T1(b, c, d, f) = \sum_{e,l} B(b, e, f, l) \times D(c, d, e, l)$ is characterized by the index sets $I = \{b, f\}$, $J = \{c, d\}$, and $K = \{e, l\}$.

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

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

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

The partitioning of data arrays among the processors and the fusions of loops both affect the total interprocessor communication cost. Fusion generally results in an increase of communication cost, but can significantly reduce the per-processor memory requirement. We use a dynamic programming algorithm to search among all combinations of loop fusions and array distributions to find the one with minimal total communication cost, that also fits within the available memory. We omit details here and refer the reader to [10] and [11].

IX. EXPERIMENTAL RESULTS

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 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 indexes p, q, r, and s have the same range N, denoting the total number of orbitals. $N = O + V$, where 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 $O(V^4 N^4)$ in the initial formula (eight nested loops, for p, q, r, s, a, b, c, and d) to $O(V N^4)$

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

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 $3 TB$), 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 disk I/O volume being much larger than the total amount of data on disk.

Three different combinations of optimizations were used to generate final concrete code, with explicit disk I/O statements.

1) **Fusion + Optimized Tiling**: The TCE loop fusion and tiling optimizations were enabled [2], [35].

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 one-third of the fourth root of the memory size.

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 2 GB. The TCE source code is shown in Fig. 9; the generated code was compiled with the Intel Itanium Fortran Compiler for Linux. Figs. 10(a), 10(b), and 11 show the generated codes for the above three combinations of optimizations. The disk I/O statements in bold face have redundant loop indexes surrounding them. A loop index is redundant for a disk I/O statement if that loop does not index the array being read or written.

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.

X. 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 some attention in research on compiler optimizations.

Reduction of arithmetic operations has been traditionally done by compilers using the technique of common subexpression elimination. Much work has been done on improving locality and parallelism by loop fusion [31], [46], [64]. However, the synthesis system presented in this paper considers a different use of loop fusion, which is to reduce array sizes and memory usage of automatically synthesized code containing nested loop structures. The contraction of arrays into scalars through loop fusion is studied in [19] but is motivated by data locality enhancement and not memory reduction. Loop fusion in the context of delayed evaluation of array expressions in APL programs is discussed in [21], but their work is also not aimed at minimizing array sizes; in addition, they consider loop fusion without considering any loop reordering.

Some recent work has explored the use of loop fusion for memory reduction for sequential execution. Strout *et al.* [70] present a technique for determining the minimum amount of

```
mlimit = 2GB;

range V = 140;
range N = 150;

procedure Transform(in disk A[N,N,N,N], in disk C1[N,V], in disk C2[N,V],
                    in disk C3[N,V], in disk C4[N,V], out disk B[V,V,V,V])=
  index a,b,c,d : V;
  index p,q,r,s : N;
begin
  B[a,b,c,d] = sum[C1[s,d]*C2[r,c]*C3[q,b]*C4[p,a]*A[p,q,r,s],{p,q,r,s}];
end
```

Fig. 9. TCE source code for four-index transform.

Read $C4, C3, C2, C1$

  FOR s_T

   FOR r_T

    FOR q_T

     FOR p_T

      Read A

      FOR a, s_I, r_I, q_I, p_I

       $T1[a, s_I, r_I, q_I] + =$

        $C4[p, a] * A[p_I, q_I, r_I, s_I]$

      FOR a, s_I, r_I, q_I, b

       $T2[a, b, s_I, r_I] + =$

        $T1[a, s_I, r_I, q_I] * C3[q, b]$

      FOR a, s_I, r_I, b, c

       $T3[a, b, c, s_I] + =$

        $T2[a, b, s_I, r_I] * C2[r, c]$

     Write $T3$

  FOR a_T

   FOR s_T

    Read $T3$

    FOR a_I, b, c, s_I, d

     $B[a_I, b, c, d] + =$

      $T3[a_I, b, c, s_I] * C1[s, d]$

   Write B

(a) Fusion + Optimized Tiling

FOR a_T

  Read $C4$

  FOR r_T, s_T

   Read A

   FOR a_I, p, q, r_I, s_I

    $T1[a_I, q, r_I, s_I] + =$

     $C4[p, a_I] * A[p, q, r_I, s_I]$

   Write $T1$

FOR a_T, b_T

  Read C3

  FOR r_T

   Read T1

   FOR s, a_I, b_I, q, r_I

    $T2[a_I, r_I, s, b_I] + =$

     $T1[a_I, q, r_I, s] * C3[q, b_I]$

  Write $T2$

Read $C2, C1$

FOR a_T, b_T

  Read $T2$

  FOR c, r, s, a_I, b_I

   $T3[a_I, s, b_I, c] + =$

    $T2[a_I, r, s, b_I] * C2[r, c]$

  Write $T3$

FOR a_T, b_T

  Read $T3$

  FOR c, d, s, a_I, b_I

   $B[a_I, b_I, c, d] + =$

    $T3[a_I, s, b_I, c] * C1[s, d]$

  Write B

(b) No Fusion, Optimized Tiling

Fig. 10. Codes generated for cases 1 and 2.

memory required for executing a perfectly nested loop with a set of constant-distance dependence vectors. 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 [66] and Song *et al.* [67], [68] 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. Pike and Hilfinger [56] apply tiling and fusion to a set of consecutive perfectly nested loops (each containing one statement) of the same nesting depth. Their work does not apply to the class of loops with complex nestings that are

considered here. There has been some work in the area of design automation to estimate storage needed for a single perfectly nested loop or a sequence of such loops [5], [32], [60], [61] and references therein. These techniques do not consider tiling and they incur additional runtime memory management overhead.

Considerable research on loop transformations for locality in nested loops has been reported in the literature [12], [44], [49], [73]. 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.* [74] consider the integrated treatment of fusion and tiling only from the point

```
FOR $a_T, p_T$
    Read $C4$
    FOR $q_T, r_T, s_T$
        Read T1
        Read A
        FOR $a_I, p_I, q_I, r_I, s_I$
            $T1[a_I, q_I, r_I, s_I]+=$
                $C4[p_I, a_I] * A[p_I, q_I, r_I, s_I]$
        Write T1
FOR $a_T, b_T, q_T$
    Read C3
    FOR $r_T$
        Read T2
        Read T1
        FOR $s, a_I, b_I, q_I, r_I$
            $T2[a_I, r_I, s, b_I]+=$
                $T1[a_I, q_I, r_I, s] * C3[q_I, b_I]$
        Write T2
Read $C2, C1$
FOR $a_T, b_T$
    Read $T2$
    FOR $c, r, s, a_I, b_I$
        $T3[a_I, s, b_I, c]+=$
            $T2[a_I, r, s, b_I] * C2[r, c]$
    Write $T3$
FOR $a_T, b_T$
    Read $T3$
    FOR $c, d, s, a_I, b_I$
        $B[a_I, b_I, c, d]+=$
            $T3[a_I, s, b_I, c] * C1[s, d]$
    Write $B$
```

Fig. 11. Code generated for case 3: no fusion, standard tiling.

Table 2
Total Disk I/O and Execution Times for Codes Generated for all Three Cases

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 = 4th root of memorySize/3	1240.85	1957.18

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 this paper. Loop tiling for enhancing data locality has been studied extensively [12], [33], [34], [59], [62], [65], [73], [74], and analytic models of the impact of tiling on locality in perfectly nested loops have been developed [20], [42], [52]. Frameworks for handling imperfectly nested loops have been presented in [1], [45], and [65]. Ahmed *et al.* [1] 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.* [45] 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 [1], [45], [65].

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], [53], [57], [75]. 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], [18], the telescoping languages project [29], [30], ATLAS [13], [72] for deriving efficient implementation of BLAS routines, and the PHIPAC [3] project. All these efforts use search-based approaches for performance tuning of codes. A comparison of model-based and search-based approaches for matrix-matrix multiplication is reported in [77] and [78]. 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 [6], [14], [48], [50], [51].

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 tradeoffs, 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). Also, some of the multidimensional arrays involved in the computation have certain domain-specific symmetry properties that can be exploited in order to lower the number of arithmetic operations, and, thus, total execution time.

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 *ab initio* 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 many months of tedious effort 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.

Table 3
Current Capabilities of the Prototype and Optimizing Tensor Contraction Engines

Capability	Prototype TCE	Optimizing TCE
Sequential code generation for CC-based methods	Yes	Yes
QC Packages Interfaced:		
· File-based	NWChem, UTCHEM	NWChem
· General (file, memory, direct)		Under development
Symmetry Support:		
· Spin	Spin Orbitals	General, in progress
· Spatial	Abelian	Abelian, in progress
· Permutational	Fermions	General, in progress
Optimizations:		
· Operation Minimization	Partial	Yes
· Memory Minimization	Partial	Yes
· Space-Time Transformation	No	Yes
· Data Locality	Partial	Yes
Parallel code generation	Limited general	General, in progress

XI. DOMAIN-SPECIFIC ISSUES IN OPTIMIZATION

One of the most challenging issues in this work has been the integration of optimizations which arise from the chemistry and physics of the particular class of problems we are targeting with the more general optimizations typical of optimizing compilers, which has been the primary focus of the paper to this point. Clearly presence of domain-specific optimizations reduces the generality of the code synthesis environment, and limits its extensibility to other problems and domains.

Such domain-specific issues arise in a number of forms. In some cases, the search space for a given optimization is too large to examine exhaustively in a reasonable time, so heuristics which encapsulate the experience of the chemist can be valuable to help restrict the search space. One example of this approach, which we are currently investigating, is the task of common subexpression elimination across multiple tensor contraction expressions in order to find intermediates that can be evaluated and reused in numerous places. An experienced quantum chemist knows from experience that certain pairings of tensors are more likely than others to yield useful factorizations. This is a case where chemistry-based heuristics can be isolated into a particular optimization module, and so do not limit the extensibility of the environment (in so far as the module can be replaced with another appropriate to a new problem or domain).

On the other hand, there are domain-specific issues that cut across many modules. For example, the prior discussion of tensors and tensor contractions did not include the fact that the tensors in this particular class of problems have a number of symmetry properties which must be utilized in order to generate the most efficient possible code.

- Permutational symmetry: tensors may be symmetric or antisymmetric on the interchange of certain indexes. As the multidimensional generalization of (anti-)symmetric matrices, this defines equivalences (within a sign) for certain portions of the tensor. Permutational symmetries are associated with the particular type of tensor, and the formulation of the quantum chemical method.

- Spatial symmetry: tensors also reflect geometric symmetry properties of the molecule on which the calculation is being performed (for example, in benzene (C_6H_6), all six carbon atoms are equivalent, as are all six hydrogen atoms). Spatial symmetry gives tensors a block structure in each index and allows the problem to be reduced to only the symmetry-unique blocks. The specifics of the number and sizes of such symmetry blocks are clearly specific to the molecule being studied, and so are only known at run time.

- Spin symmetry: is associated with the quantum mechanical spin of the electrons. Electronic structure calculations are typically formulated so that electron spin is among the slowest changing variables in the nested loop structure that drives these the calculations. So spin symmetries typically lead to very large blocks of tensors being hard zeros; there are few enough nonzero blocks that each combination of spins is often implemented in a separate tensor object.

All of these symmetries affect the detailed structure of the tensors, their natural block structure, and number of unique elements. Clearly this has a significant impact on code generation, but it can also factor into the optimizations themselves. For example, permutational symmetries affect where loop fusions can be effectively applied, and spatial symmetry constrains tile sizes for space–time tradeoffs and data locality optimization.

Another domain-specific factor that enters into the problem pertains to the formulation of the quantum chemical method. Historically, most methods in quantum chemistry have been expressed in the MO basis, which has certain implications on the structure and sparsity of the tensors. More recently, there is increasing use of an alternative "local/AO" based formulation, in an attempt to produce implementations with better computational scaling properties (the two approaches are often mixed in different parts of the quantum chemical method). For example, in the traditional MO formulation, most of the tensors are relatively dense, while in the AO formulation, a rank-4 tensor of total size $O(N^4)$ might have just $O(N^2)$ nonzero elements for large molecules, with a rather different blocking pattern than in the MO approach. Though it might have been different 10 or

15 years ago, today it is not reasonable to develop a general purpose code synthesis capability in this domain without the ability to handle both MO and AO-based approaches with good efficiency.

Our approach to dealing with domain-specific optimization issues has been twofold. A "prototype" TCE has been developed [25], [26], which focuses on systematizing and automating the approach a chemist might take in writing the code by hand. Concurrently, we are working on implementing an "optimizing" TCE which starts from a relatively domain-independent computer science base. In addition to being extremely functional in its own right, the prototype TCE augments discussions between the chemists and computer scientists on the team by allowing us to look at concrete implementations of code generation tools from the chemists' viewpoint. Once the domain-specific optimizations are understood *in situ*, we can more easily move to integrate them into the optimizing TCE, while retaining as much generality as possible. We have worked through this process for the tensor symmetries mentioned above, and appropriate modifications are now being incorporated into the optimizing TCE. The prototype TCE is now capable of generating code for AO-based approaches, and this will be the next domain-specific optimization to be integrated into the optimizing TCE.

XII. CURRENT STATUS

Both the prototype and optimizing TCE tools are capable of generating both sequential and parallel code for a wide range of electronic structure methods in the target domain. Table 3 summarizes the current capabilities of both tools.

The prototype TCE has been particularly valuable, having been used already to implement more than 20 different methods in the coupled cluster family, many of which received their first ever parallel implementation in this way. These new capabilities have been integrated with, and are distributed as part of the NWChem version 4.5 [23] and UTCHEM 2003 [76] computational chemistry packages and have enabled benchmark quantum chemical applications that were not possible before the development of the TCE [27].

REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali, "Synthesizing transformations for locality enhancement of imperfectly-nested loops," presented at the ACM Int. Conf. Supercomputing, Santa Fe, NM, 2000.

[2] A. Bibireata, S. Krishnan, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, and V. Choppella *et al.*, "Memory-constrained data locality optimization for tensor contractions," in *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*, L. Rauchwerger *et al.*, Eds. Heidelberg, Germany: Springer-Verlag, 2004, vol. 2958, pp. 93–108.

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

[4] L. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State Univ., Bozeman, 1969.

[5] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*. Norwell, MA: Kluwer, 1998.

[6] R. Choy and A. Edelman, "Parallel MATLAB: Doing it right," *Proc. IEEE*, vol. 93, no. 2, pp. 331–341, Feb. 2005.

[7] D. Cociorva, J. Wilkins, C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam, "Loop optimizations for a class of memory-constrained computations," in *Proc. 15th ACM Int. Conf. Supercomputing*, 2001, pp. 103–113.

[8] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison, "Toward automatic synthesis of high-performance codes for electronic structure calculations: Data locality optimization," in *Lecture Notes on Computer Science, High Performance Computing—HiPC 2001*, 2001, vol. 2228, pp. 237–248.

[9] ——, "Space–time trade-off optimization for a class of electronic structure calculations," in *Proc. 2002 Conf. Programming Language Design and Implementation (PLDI)*, pp. 177–186.

[10] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam, "Memory-constrained communication minimization for a class of array computations," presented at the Workshop Languages and Compilers for Parallel Computing, West Lafayette, IN, 2004.

[11] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam, "Global communication optimization for tensor contraction expressions under memory constraints," presented at the 17th Int. Parallel and Distributed Processing Symp. (IPDPS), Nice, France, 2003.

[12] S. Coleman and K. McKinley, "Tile size selection using cache organization and data layout," presented at the ACM SIGPLAN Conf. Programming Language Design and Implementation, La Jolla, CA, 1995.

[13] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick, "Self adapting linear algebra algorithms and software," *Proc. IEEE*, vol. 93, no. 2, pp. 293–312, Feb. 2005.

[14] L. De Rose and D. Padua, "A MATLAB to fortran 90 translator and its effectiveness," in *Proc. 10th ACM Int. Conf. Supercomputing*, 1996, pp. 309–316.

[15] J. Foresman and A. Frisch, *Exploring Chemistry With Electronic Structure Methods: A Guide to Using Gaussian*, Second ed. Pittsburgh, PA: Gaussian, Inc., 1996.

[16] A. Fraboulet, G. Huard, and A. Mignotte, "Loop alignment for memory access optimization," in *Proc. 12th Int. Symp. System Synthesis*, 1999, pp. 71–77.

[17] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, vol. 3, 1998, pp. 1381–1384.

[18] ——, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[19] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath, "Collective loop fusion for array contraction," presented at the Workshop Languages and Compilers for Parallel Computing, New Haven, CT, 1992.

[20] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," presented at the 8th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1998.

[21] L. Guibas and D. Wyatt, "Compilation and delayed evaluation in APL," in *Proc. 5th Annu. ACM Symp. Principles of Programming Languages*, 1978, pp. 1–8.

[22] *NWChem, A Computational Chemistry Package for Parallel Computers, Version 3.3*, High Perform. Comput. Chem. Group, Pacific Northwest Nat. Lab., Richland, WA, 1999.

[23] *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.5*, High Perform. Comput. Chem. Group, Pacific Northwest Nat. Lab., Richland, WA, 2003.

[24] C. L. Janssen, E. T. Seidl, G. E. Scuseria, T. P. Hamilton, Y. Yamaguchi, R. B. Remington, Y. Xie, G. Vacek, C. D. Sherrill, T. D. Crawford, J. T. Fermann, W. D. Allen, B. R. Brooks, G. B. Fitzgerald, D. J. Fox, J. F. Gaw, N. C. Handy, W. D. Laidig, T. J. Lee, R. M. Pitzer, J. E. Rice, P. Saxe, A. C. Scheiner, and H. F. Schaefer, "PSI 2.0.8," PSITECH, Inc., Watkinsville, GA, psi@ccqc.uga.edu, 1995.

[25] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *J. Phys. Chem. A*, vol. 107, pp. 9887–9897, 2003.

[26] S. Hirata, A. Auer, and M. Nooijen, *Tensor Contraction Engine*. Richland, WA: Pacific Northwest Nat. Lab., 2003.

[27] S. Hirata, T. Yanai, W. A. de Jong, T. Nakajima, and K. Hirao, "Third-order Douglas.Kroll relativistic coupled-cluster theory through connected single, double, triple, and quadruple substitutions: Applications to diatomic and triatomic hydrides," *J. Chem. Phys.*, vol. 120, pp. 3297–3310, 2004.

[28] J. Johnson, R. Johnson, D. Padua, and J. Xiong, "Searching for the best FFT formulas with the SPL compiler," in *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*. Heidelberg, Germany: Springer-Verlag, 2001, pp. 112–126.

[29] K. Kennedy, B. Broom, 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 Distrib. Comput.*, vol. 61, no. 12, pp. 1803–1826, Dec. 2001.

[30] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey, "Telescoping languages: A system for automatic generation of domain languages," *Proc. IEEE*, vol. 93, no. 2, pp. 387–408, Feb. 2005.

[31] K. Kennedy and K. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*. Heidelberg, Germany: Springer-Verlag, 1993, vol. 768, pp. 301–320.

[32] P. Kjeldsberg, F. Catthoor, and E. Aas, "Data dependency size estimation for use in memory optimization," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 22, no. 7, pp. 908–921, July 2003.

[33] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric multi-level blocking," in *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, 1997, pp. 346–357.

[34] I. Kodukula, K. Pingali, R. Cox, and D. Maydan, "An experimental evaluation of tiling and shackling for memory hierarchy management," presented at the ACM Int. Conf. Supercomputing (ICS '99), Rhodes, Greece.

[35] 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 algoritms," in *Lecture Notes in Computer Science, High Performance Computing—HiPC 2003*, 2003, vol. 2913, pp. 406–417.

[36] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*. Redwood City, CA: Benjamin Cummings, 1994, pp. 171–171.

[37] C. Lam, "Performance optimization of a class of loops implementing multi-dimensional integrals," Ph.D. dissertation, Ohio State Univ., Columbus, 1999.

[38] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan, "Memory-optimal evaluation of expression trees involving large objects," in *Lecture Notes in Computer Science, High Performance Computing*. Heidelberg, Germany: Springer-Verlag, 1999, vol. 1745, pp. 103–110.

[39] ——, "Optimization of memory usage for a class of loops implementing multi-dimensional integrals," in *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*. Heidelberg, Germany: Springer-Verlag, 1999, vol. 1863, pp. 350–364.

[40] C. Lam, P. Sadayappan, and R. Wenger, "On optimizing a class of multi-dimensional loops with reductions for parallel execution," *Parallel Process. Lett.*, vol. 7, no. 2, pp. 157–168, 1997.

[41] ——, "Optimization of a class of multi-dimensional integrals on parallel machines," presented at the 8th SIAM Conf. Parallel Processing for Scientific Computing, Minneapolis, MN, 1997.

[42] M. Lam, E. Rothberg, and M. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 63–74.

[43] T. Lee and G. Scuseria, "Achieving chemical accuracy with coupled cluster theory," in *Quantum Mechanical Electronic Structure Calculations With Chemical Accuracy*, S. R. Langhoff, Ed. Norwell, MA: Kluwer, 1997, pp. 47–109.

[44] W. Li, "Compiling for NUMA parallel machines," Ph.D. dissertation, Cornell University, Ithaca, NY, 1993.

[45] A. Lim, S. Liao, and M. Lam, "Blocking and array contraction across arbitrarily nested loops using ane partitioning," in *Proc. 8th ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, 2001, pp. 103–112.

[46] N. Manjikian and T. Abdelrahman, "Fusion of loops for parallelism and locality," in *Proc. Int. Conf. Parallel Processing*, 1995, pp. II:19–II:28.

[47] J. M. L. Martin, "Benchmark studies on small molecules," in *Encyclopedia of Computational Chemistry*, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, and H. F. Schaefer III, Eds. Berne, Switzerland: Wiley, 1998, vol. 1, pp. 115–128.

[48] The Match Project: A MATLAB compilation environment for distributed heterogeneous adaptive computing systems. [Online]. Available: http://www.ece.nwu.edu/cpdc/Match/Match.html

[49] K. McKinley, S. Carr, and C. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, Jul. 1996.

[50] V. Menon and K. Pingali, "High-level semantic optimization of numerical codes," in *Proc. 13th ACM Int. Conf. Supercomputing*, 1999, pp. 434–443.

[51] ——, "A case for source-level transformations in MATLAB," in *Proc. 2nd Conf. Domain-Specific Languages*, 1999, pp. 53–65.

[52] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante, "Quantifying the multi-level nature of tiling interactions," *Int. J. Parallel Program.*, vol. 26, no. 6, pp. 641–670, Jun. 1998.

[53] J. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Puschel, and M. Veloso. (1998) SPIRAL: Portable library of optimized signal processing algorithms. [Online]. Available: http://www.ece.cmu.edu/~piral

[54] J. Nieplocha and R. Harrison, "Shared-memory programming in metacomputing environments: The global array approach," *J. Supercomput.*, vol. 11, pp. 119–136, 1997.

[55] J. Nieplocha, R. Harrison, and R. Littlefield, "Global arrays: A portable shared memory model for distributed memory computers," in *Proc. Supercomputing '94*, 1994, pp. 340–349.

[56] G. Pike and P. Hilfinger, "Better tiling and array contraction for compiling scientific programs," in *Proc. ACM/IEEE Conf. Supercomputing '02: High Performance Networking and Computing*, pp. 1–12.

[57] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005.

[58] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon, "A fifth-order perturbation comparison of electron correlation theories," *Chem. Phys. Lett.*, vol. 157, pp. 479–483, 1989.

[59] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for multicomputers," *J. Parallel Distrib. Comput.*, vol. 16, no. 2, pp. 108–120, Oct. 1992.

[60] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan, "Reducing memory requirements of nested loops for embedded systems," in *Proc. 38th ACM/IEEE Design Automation Conf.*, 2001, pp. 359–364.

[61] ——, "Estimating and reducing the memory requirements of signal processing codes for embedded processor systems," *IEEE Trans. Signal Process.*, to be published.

[62] G. Rivera and C. Tseng, "A comparison of compiler tiling algorithms," presented at the 8th Int. Conf. Compiler Construction, Armsterdam, The Netherlands, 1999.

[63] M. Schmidt, K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis, and J. Montgomery, "General atomic and molecular electronic structure system (GAMESS)," *J. Comput. Chem.*, vol. 14, pp. 1347–1363, 1993.

[64] S. Singhai and K. McKinley, "Loop fusion for data locality and parallelism," presented at the Mid-Atlantic Student Workshop on Programming Languages and Systems, New Paltz, NY, 1996.

[65] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," presented at the 1999 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'99), Atlanta, GA.

[66] Y. Song, "Compiler algorithms for efficient use of memory systems," Ph.D. dissertation, Purdue Univ., West Lafayette, IN, 2000.

[67] Y. Song, C. Wang, and Z. Li, "Locality enhancement by array contraction," in *Proc. 14th Int. Workshop Languages and Compilers for Parallel Computing*, 2001, pp. 132–146.

[68] Y. Song, R. Xu, C. Wang, and Z. Li, "Data locality enhancement by memory reduction," in *Proc. 15th ACM Int. Conf. Supercomputing*, 2001, pp. 50–64.

[69] J. F. Stanton, J. Gauss, J. D. Watts, M. Nooijen, N. Oliphant, S. A. Perera, P. G. Szalay, W. J. Lauderdale, S. A. Kucharski, S. R. Gwaltney, S. Beck, A. B. D. E. Bernholdt, K. K. Baeck, P. Rozyczko, H. Sekino, C. Hober, and R. J. Bartlett, "ACES II. Quantum Theory Project," Univ. Florida, Gainesville.

[70] M. Strout, L. Carter, J. Ferrante, and B. Simon, "Schedule-independent storage mapping for loops," in *Proc. 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 24–33.

[71] H.-J. Werner and P. J. Knowles, "MOLPRO,", http://www.tc.bham.ac.uk/.

[72] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software (ATLAS)," presented at the Supercomputing Conf. '98, Reno, NV.

[73] M. Wolf and M. Lam, "A data locality optimization algorithm," in *Proc. SIGPLAN'91 Conf. Programming Language Design and Implementation*, pp. 30–44.

[74] M. Wolf, D. Maydan, and D. Chen, "Combining loop transformations considering caches and scheduling," in *Proc. 29th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO-29)*, 1996, pp. 274–286.

[75] J. Xiong, D. Padua, and J. Johnson, "Spl: A language and compiler for DSP algorithms," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2001, pp. 298–308.

[76] T. Yanai, M. Kamiya, Y. Kawashima, T. Nakajima, H. Y. N. Nakano, H. Sekino, J. Paulovic, T. Tsuneda, S. Yanagisawa, and K. Hirao, "UTCHEM 2003," Univ. Tokyo, Tokyo, Japan.

[77] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization," in *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation*, pp. 63–76.

[78] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "A comparison of empirical and model-driven optimization," *Proc. IEEE*, vol. 93, no. 2, pp. 358–386, Feb. 2005.

Gerald Baumgartner received the Dipl.Ing. degree in computer science from the University of Linz, Linz, Austria, in 1988 and the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, IN, in 1992 and 1996, respectively.

He began his academic career at Ohio State University, Columbus, in 1997 and is currently visiting the Department of Computer Science at Louisiana State University, Baton Rouge. His research interest includes compiler optimizations, the design and implementation of domain-specific and object-oriented languages, desktop grids, and development and testing tools for object-oriented and embedded systems programming.

Alexander Auer received the Diploma degree in chemistry from the University of Köln, Köln, Germany, in 1998 and the Ph.D. degree in chemistry from the University of Mainz, Mainz, Germany, in 2002.

He was a Visiting Student at the University of Oslo from August 1998 to February 1999. From July 2002 to April 2004, he was a Postdoctoral Research Fellow at Princeton University, Princeton, NJ, and the University of Waterloo, Waterloo, ON Canada. Since April 2004, he has been a "Juniorprofessor Theoretische Chemie" at the Technische Universität Chemnitz, Chemnitz, Germany.

David E. Bernholdt received the B.S. degree in chemistry from the University of Illinois, Urbana, in 1986 and the Ph.D. degree in chemistry with minors in physics and mathematics from the University of Florida, Gainesville, in 1993.

He held positions at Pacific Northwest National Laboratory and Syracuse University before taking his current position at Oak Ridge National Laboratory, Oak Ridge, TN, in 2000. His research experience includes significant experience in the development and implementation of new methods and algorithms for computational quantum chemistry on parallel computers. Since then, his research interests have shifted to computer science, with a particular focus on improving performance and productivity of scientific software through the development of technologies such as component environments, parallel programming models, and domain-specific high-level programming languages.

Alina Bibireata received the B.S. degree in computer science from the University of Bucharest, Bucharest, Romania, in 1997 and the M.S. degree in computer science from Ohio State University, Columbus, in 2004.

Her research interests are in programming languages, software engineering, and design patterns for enterprise applications.

Venkatesh Choppella received the B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur, in 1985 and the Ph.D. degree in computer science from Indiana University, Bloomington, in 2002.

He has held research and engineering positions at Xerox Corporation, Hewlett-Packard Company, Indiana University, and Oak Ridge National Laboratory. He is currently on the faculty at the Indian Institute of Information Technology and Management, Kerala. His research interests are in programming languages, compilers for domain specific languages, automated deduction, and software engineering.

Daniel Cociorva received the B.S. and M.S. degrees in theoretical physics from the University of Lyon, Lyon, France, in 1995 and 1996, respectively, and the Ph.D. degree from Ohio State University, Columbus, in 2001.

He started working in computational physics as a graduate student at Ohio State University, Columbus. As a Postdoctoral Associate, he worked on the Tensor Contraction Engine project for automatic code generation and optimization in quantum chemistry. He is currently employed as a Bioinformatics Analyst in mass spectrometry proteomics at the Scripps Research Institute, La Jolla, CA. In his Ph.D. dissertation, completed in 2001, he used advanced numerical methods to study properties of interfaces and defects in semiconductor structures.

Xiaoyang Gao received the B.S. degree in computer science from Peking University, Beijing, China, in 1997. She is currently working toward the Ph.D. degree in the Department of Computer Science and Engineering, Ohio State University, Columbus.

Her research interests are in distributed systems, compilers for high-performance computer systems, and software optimizations.

Robert J. Harrison received the B.A. degree in natural sciences and the Ph.D. degree in theoretical chemistry from Cambridge University, Cambridge, U.K., in 1981 and 1984, respectively.

He worked as a Postdoctoral Research Fellow at the Quantum Theory Project and the Daresebury Laboratory before joining the staff of the theoretical chemistry group at Argonne National Laboratory in 1988. In 1992, he moved to the Environmental Molecular Sciences Laboratory of Pacific Northwest National Laboratory, conducting research in theoretical chemistry and leading the development of NWChem, a computational chemistry code for massively parallel computers. Since 2002, he has held a joint appointment between Oak Ridge National Laboratory (ORNL), Oak Ridge, TN, and the Chemistry Department of the University of Tennessee, Knoxville. AT ORNL, he is Leader of the Computational Chemical Sciences Group in the Computer Science and Mathematics Division. He has over 75 publications in peer-reviewed journals in the areas of theoretical and computational chemistry, and high-performance computing. In addition to his SciDAC research into efficient and accurate calculations on large systems, he has been pursuing applications in molecular electronics and chemistry at the nanoscale. His interests are in theoretical and computational chemistry, high-performance computing, electron correlation, electron transport, relativistic chemistry, and response theory.

Dr. Harrison and the NWChem team received an *R&D Magazine* R&D100 award in 1999. In 2002, he received the IEEE Computer Society Sydney Fernbach award.

So Hirata received the B.S. and M.S. degrees in chemistry from the University of Tokyo, Tokyo, Japan, in 1994 and 1996, respectively, and the Ph.D. degree in theoretical chemistry from the Graduate University for Advanced Studies, Okazaki, Japan, in 1998.

He was a Senior Research Scientist at Pacific Northwest National Laboratory (2001–2004); a Postdoctoral Research Fellow at the University of Florida (1999–2001); a Visiting Scholar at University of California, Berkeley (1998–1999); and a Japan Society for the Promotion of Science Young Scientist (1996–1999). He currently serves as Assistant Professor in Chemistry at the University of Florida, Gainesville, and an Adjunct Associate Professor in Chemistry at Hiroshima University, Hiroshima, Japan. He has authored or coauthored over 40 peer-reviewed journal articles, one book chapter, and two high-performance quantum chemistry software suites. His research interests include *ab initio* MO theory and density functional theory for electronic structure calculations of atoms, molecules, and crystalline solids.

Sriram Krishnamoorthy was born in Chennai, India, in 1981. He received the B.E. degree from the College of Engineering, Guindy, Anna University, Chennai, India, in 2002. He is currently working toward the M.S. degree at Ohio State University, Columbus, under the supervision of Prof. P. Sadayappan.

His research interests include high-performance computing, out-of-core algorithms, and optimizations for scientific computing.

Sandhya Krishnan received the B.E. degree in computer engineering from Bombay University, India in March 2001, and graduated with a Masters in Computer and Information Science from The Ohio State University, Columbus, Ohio in September 2003. She continued to work as a Systems Developer on the same project until May 2004. Her research interests include high performance computing and development of optimization algorithms.

Chi-Chung Lam received the Ph.D. degree in computer and information science from Ohio State University, Columbus, in 1999. The title of his dissertation was "Performance optimization of a class of loops implementing multi-dimensional integrals."

Qingda Lu received the B.E. degree in computer science from Beijing Institute of Technology, Beijing, China, in 1999 and the M.S. degree in computer science from Peking University, Beijing, in 2002. He is currently working toward the Ph.D. degree in the Department of Computer Science and Engineering, Ohio State University, Columbus.

His research interests include optimizing compilers and performance modeling/monitoring.

Marcel Nooijen was born in the Netherlands in 1963. He received the Ph.D. degree from the Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, in 1992, working with E.-J. Baerends and the late J. Snijders.

He continued his research in many-body and coupled cluster theory for electronically excited states with R. Bartlett at the University of Florida. In 1997, he joined Princeton University as an Assistant Professor and started work on automated program generation and the Tensor Contraction Engine. In 2003, he relocated to the University of Waterloo, Waterloo, ON, Canada. His other research interests concern the coupling of electronic and nuclear motion and their effect on the description of molecular spectra.

Dr. Nooijen is the recipient of the 2003 Medal of the Academy of Quantum Molecular Science.

Russell M. Pitzer received the B.S. degree in chemistry from the California Institute of Technology, Pasadena, in 1959 and the Ph.D. degree from Harvard University, Cambridge, MA, in 1963.

He has been a member of the faculty of the Department of Chemistry, Ohio State University, Columbus, since 1968. He is a Cofounder of the Ohio Supercomputer Center and its attendant statewide academic computer network. He is one of the authors of the Columbus suite of molecular programs. His research interests include developing and applying methods and software for molecular electronic structure and spectroscopy. His current research is on relativistic methods for molecules containing heavy atoms.

J. Ramanujam received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, in 1983 and the M.S. and Ph.D. degrees in computer science from Ohio State University, Columbus, in 1987 and 1990, respectively.

He is currently a Professor in the Department of Electrical and Computer Engineering at Louisiana State University, Baton Rouge. He has published over nearly 120 papers in refereed journals and conferences in addition to several book chapters and a book. His research interests are in compilers for high-performance computer systems, embedded systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures and algorithms.

Dr. Ramanujam received the National Science Foundation's Young Investigator Award in 1994. In addition, he has received the best paper awards at the 2003 International Conference on High Performance Computing (HiPC 2003) and the 2004 International Parallel and Distributed Processing Symposium (IPDPS 2004) for his work with others on compiler optimizations for quantum chemistry computations.

P. Sadayappan received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, in 1977 and the M.S. and Ph.D. degrees in electrical engineering from the State University of New York, Stony Brook, in 1978 and 1983, respectively.

He is currently a Professor in the Department of Computer Science and Engineering at Ohio State University, Columbus. His research interests include compile/runtime optimization and scheduling and resource management for parallel/distributed systems.

Alexander Sibiryakov received the B.S. degree in physics from Tomsk State University, Tomsk, Russia, in 1993 and the M.S. degree in physics and the M.S. degree in computer science and engineering from Ohio State University, Columbus, in 2002 and 2004, respectively.

He is currently with Microsoft, Redmond, WA. His research interests include operation minimization and tensor symmetries.