Molecular Physics, Vol. 104, No. 2, 20 January 2006, 211–228

Taylor & Francis
Taylor & Francis Group

Automatic code generation for many-body electronic structure methods: the tensor contraction engine‡‡

ALEXANDER A. AUER†, GERALD BAUMGARTNER‡, DAVID E. BERNHOLDT*¶, ALINA BIBIREATA§, VENKATESH CHOPPELLA¶, DANIEL COCIORVA§, XIAOYANG GAO§, ROBERT HARRISON¶, SRIRAM KRISHNAMOORTHY§, SANDHYA KRISHNAN§, CHI-CHUNG LAM§, QINGDA LU§, MARCEL NOOIJEN‖, RUSSELL PITZER⊥, J. RAMANUJAM††, P. SADAYAPPAN§ and ALEXANDER SIBIRYAKOV§

†Technische Universität Chemnitz, Fakultät für Naturwissenschaften, Institut für Chemie, Straße der Nationen 62, D-09111 Chemnitz, Germany
‡Dept. of Computer Science, Louisiana State University, Baton Rouge, LA 70803 USA
§Dept. of Computer Science and Engineering, The Ohio State University, 2015 Neil Avenue, Columbus, OH 43210-1277 USA
¶Computer Science and Mathematics Division, Oak Ridge National Laboratory, P. O. Box 2008, Oak Ridge, TN 37831 USA
‖Dept. of Chemistry, University of Waterloo, Waterloo, Ontario N2L BG1, Canada
⊥Dept. of Chemistry, The Ohio State University, 100 W. 18th Aveune, Columbus, OH 43210 USA
††Dept. of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803 USA

(*Received 1 November 2004; in final form 11 December 2005*)

As both electronic structure methods and the computers on which they are run become increasingly complex, the task of producing robust, reliable, high-performance implementations of methods at a rapid pace becomes increasingly daunting. In this paper we present an overview of the Tensor Contraction Engine (TCE), a unique effort to address issues of both productivity and performance through automatic code generation. The TCE is designed to take equations for many-body methods in a convenient high-level form and acts like an optimizing compiler, producing an implementation tuned to the target computer system and even to the specific chemical problem of interest. We provide examples to illustrate the TCE approach, including the ability to target different parallel programming models, and the effects of particular optimizations.

1. Introduction

Over roughly the last twenty-five years, coupled cluster (CC) and other many-body methods have been developed into the dominant methodology for high-quality electronic structure calculations, thanks to the significant efforts of Prof. Rodney J. Bartlett and his research collaborators, as well as numerous other research groups. Starting from the simplest CCD method [1–3] through the recent implementation of CCSDTQP [4], these methods have grown to a level of theoretical sophistication and computational complexity that could hardly have been imagined when the first methods were published, and have come to represent enormous coding tasks. Similarly, the breadth of methods has expanded from straightforward single-reference CC methods to include a variety of multireference approaches [5–17], methods for excited states and various properties [5, 18–30].

At the same time, computer hardware has grown orders of magnitude more powerful, helping to deliver

*Corresponding author. Email: bernholdtde@ornl.gov
‡‡This paper is dedicated to Prof Rodney J. Bartlett on the occasion of his 60th birthday.

Molecular Physics
ISSN 0026–8976 print/ISSN 1362–3028 online
http://www.tandf.co.uk/journals
DOI: 10.1080/00268970500275780

on the promise of applying many-body methods to "real" chemical problems and synergistically spurring new scientific goals, new method development and new algorithms to take advantage of the ever-increasing computational power. In addition to new algorithms for "standard" methods [31, 32], new variations, such as "local" and atomic orbital (AO) based implementations [33–36], and approaches based on resolution of the identity, density fitting, and Cholesky decomposition [37, 38] are being developed in response to the size and physical characteristics of the chemical systems now within reach of many-body methods.

However, in order to achieve these performance improvements, computer hardware has also grown significantly more complex, with increasing disparities between the fundamental capabilities of the CPU and the abilities of the memory and other busses (i.e. disk) to move the data through the memory hierarchy at a pace that allows the CPU to perform up to its capabilities. This period has also seen the coming-of-age of parallel computing, with dual-processor desktop systems being routinely available, and ownership of larger Beowulf clusters or more highly-integrated parallel systems easily accessible to many research groups. Packages such as NWChem [39, 40] and MPQC [41–43] have been developed largely from scratch for large-scale parallel computing environments, and many other electronic structure packages have been incrementally parallelized, by transformation or rewriting of existing sequential code, resulting in parallel-capable code with a wide range of parallel scalability.

The complexity of modern software development in this field can be seen in a simple comparison of the number of terms in various coupled cluster methods and the number of source lines of code (SLOCs) [44] required to implement them. While the precise numbers will, of course, vary with the details of the implementation, table 1 illustrates the comparison for one particular case. The fact that roughly 300 lines of code are required per term, and that the number rises with the level of excitation included in the method, is an indication of the size of the semantic gap between the way quantum chemists think about their methods and what is required to implement them with current general-purpose programming languages, such as Fortran, C, or C++. Serious attempts to produce implementations with maximum sequential performance across a broad range of computer platforms would increase the ratio, and highly scalable parallelization would increase it even more, to the extent of requiring multiple implementations of some parts of the code to achieve high performance on all platforms. As a consequence, researchers are typically forced to choose between exploration of new methodological ideas and the creation of robust, high-performance implementations which are applicable to a wide range of chemical problems. Moreover, even when focused in this manner, new developments often require months of effort and still result in codes with lower levels of capability or performance than might be desired.

Table 1. Number of terms in various CC methods, the number of source lines of code (SLOC) required to implement them, and the ratio of the two. SLOC counts only code for evaluating the basic tensor contraction expressions and does not include integral evaluation or other necessary code. Counts are for code generated by the prototype Tensor Contraction Engine [45, 46], and are courtesy of So Hirata [47].

Method	Terms	SLOC	SLOC/Term
CCD	11	3,209	292
CCSD	48	13,213	275
CCSDT	102	33,932	333
CCSDTQ	183	79,901	437

To help address this problem, the electronic structure community has often turned to more advanced methods to accelerate the task of implementing the desired software, including the use of automatic code generation tools. Such tools are designed either to generate code for a specific method or problem, or to provide a "high-level language" which is generally much smaller than a general-purpose language and tailored for a class of methods or problems. Historically, these approaches have focused primarily on the "productivity" side of the complexity problem described above – the rate at which methods can be implemented – though a few have been aimed at enhancing performance, usually of a very particular problem or algorithm. In this paper, we offer an overview of an ongoing effort to develop a suite of tools, known as the Tensor Contraction Engine (TCE), intended to simultaneously address both productivity *and* performance for a broad range of many-body methods.

2. Background: advanced approaches to software development for electronic structure theory

The most common approach to increasing productivity in scientific software development is the abstraction of common code motifs into libraries which can be easily reused. Among the most familiar examples of this approach in electronic structure software are probably numerical libraries, such as the BLAS [48] for basic vector and matrix operations, and LAPACK [49, 50] for linear solvers, eigensolvers, and other basic linear algebra tools. The nature of high-end electronic structure methods is such that there are significant

similarities in the code required to implement one method or another, both in terms of overall structure and specific content. However the specific differences between methods can make it very challenging to directly reuse code from one to another. For example, in the development of many-body methods, the most easily recognized motif is probably the contraction of tensors (integrals, excitation amplitudes, etc.) to form other tensors (or, sometimes, scalars). However efficient implementations of tensor contractions will vary significantly depending on the details of how the tensors are stored, which indices are contracted, and even the size of the space covered by each index. As a result, attempts to abstract tensor contractions into libraries tend to be either very general and rather inefficient, or to provide many distinct routines for different types of contractions, and thus have too little of the desired abstraction.

Generalized algorithms that work across a family of methods are a higher-level alternative to direct reuse of code for simplifying the development of complex quantum chemical software. This approach involves the development of a computational formalism that allows an entire family of related methods to be expressed and evaluated within a single body of code. For example, formulations allowing the evaluation of properties to arbitrary orders have been developed by Dykstra and Jaisen at the SCF level [51, 52], and Piecuch and coworkers for coupled cluster linear response theory [53, 54]. Full-CI codes have often been adapted to provide generalized algorithms for families of other methods. Arbitrary-order perturbation theory (i.e. MBPT-*n*) has been demonstrated by Knowles and Handy [55, 56]. Coupled cluster methods with arbitrary levels of excitation have been implemented by Hirata [57], Kàllay [58], and Olsen [59]. Similar capabilities have been demonstrated for Equation of Motion coupled cluster (EOMCC) theories by Hirata *et al*. [60, 61] and Hald *et al*. [62], as well as perturbative corrections to EOMCC [63]. Recently Kallay has used string based methods that are used widely in CI methods (e.g. [64]) to generate CC energies of arbitrary order, using a single general procedure to effect the tensor contractions. In this implementation the antisymmetry of input and intermediate tensors is treated in an efficient way, and Kallay also introduced an elegant factorization scheme for CC based methods [65], such that the scaling of the method is similar to hand-coded implementations. The string based algorithm has been used subsequently to implement active space coupled cluster methods [66], and analytical gradients [67] and Hessians [68] for CC methods of arbitrary excitation level.

Another approach to simplifying software development is automatic code generation. The code generation system embodies a formalized understanding of how to write the code for a given class of methods, capturing the similarities among the different methods, while allowing the specifics to be tailored to each method. In this case, the ''reuse'' is not at the level of the code itself, but rather the conceptual level above that. Code generation may be controlled by modifying the generator itself, through simple configuration parameters provided to the tool, or by a formally-defined language (with a grammar and parser), which is often referred to as a domain-specific language (in this case the scientific domain of electronic structure theory) or a high-level language, which should be contrasted with a general-purpose programming language like C or Fortran. The work of Janssen and Schaefer [69], Li and Paldus [70], and Nooijen and coworkers [71, 72] are among the published examples of the use of automatic code generation in the context of various coupled cluster methods. Recently, So Hirata created a prototype version of the Tensor Contraction Engine that accomplished the automated implementation of scalable parallel versions of CI, MBPT and CC methods up to quadruple excitation levels [45], and Equation of Motion Coupled Cluster energies and properties [46]. While in these examples the primary utility of the code-generation approach is to simplify and accelerate the implementation of complex many-body methods, code-generation approaches can also be applied to improving performance. For example Fermann and Valeev's Libint [73] uses automatic code generation to produce highly-optimized routines for integral evaluation analogous to the way ATLAS [74, 75] generates tuned implementations of the BLAS libraries. Code-generation approaches have some of the same limitations as generalized algorithms. The breadth of methods to which code generation can be applied tends not to be as limited as for generalized algorithms, but depends strongly on the level of effort put into the generality of the code generation tool. Performance of the generated code also strongly depends on the effort put into the generator. On the other hand, automatic code generation tools offer a number of unique advantages as an approach for complex large-scale electronic structure codes:

- New programming models, performance-related changes, and other implementation details can be applied to a wide range of methods by modifying the generator and regenerating the various methods of interest.
- It may be practical to generate implementations tailored to specific computer platforms, or to particular chemical problems in order to obtain better performance or better management of computational resources.

- Code generation driven by a high-level domain-specific language provides the user an opportunity to express the calculation to be performed in a form that is much closer to the way the researcher derived the equations than is possible with hand implementations in a traditional general-purpose programming language.

To date, applications of automatic code-generation ideas in electronic structure theory, and indeed in other areas of computational science as well, have focused essentially on *either* productivity (increasing the rate at which software can be created) *or* the performance of the resulting software. However there is no limitation intrinsic to the approach that prevents both goals from being pursued simultaneously. The Tensor Contraction Engine (TCE) is a unique effort to bring together the productivity and performance benefits of automatic code generation to quickly and easily produce high-performance parallel implementations of a broad spectrum of many-body electronic structure methods. The project team brings together a group of quantum chemists and computer scientists and treats the problem very much like the development of an optimizing compiler. In the following sections, we describe the overall approach and architecture of the TCE and provide some specific examples of the benefits of this approach in terms of flexibility, productivity, and performance.

3. The tensor contraction engine

The Tensor Contraction Engine is structured along the same basic lines as a compiler for a general-purpose language, such as Fortran: an input language is parsed into an internal representation, that internal representation is processed to transform and optimize it, and finally "executable" code is generated. In the case of the TCE, the input language is a high-level, domain-specific language which allows the user to express the equations to be implemented in a form natural to a quantum chemist, based on tensor expressions. The use of a high-level input language provides the TCE with a view of the problem in a natural form of expression, *before* it is translated to satisfy the constraints of a general-purpose language, such as Fortran. Based on this high-level view, the TCE can perform optimizations that are not possible once the desired operations have been translated into a general-purpose language. These optimizations can rigorously and systematically explore the space of possible transformations, compared to the far more empirical approach typically taken by software developers implementing such codes by hand. Moreover, it is possible to tailor the TCE's processing and optimizations to the resources and performance characteristics of each hardware platform on which the generated code will run, producing a different implementation tuned to each target system. In fact, the generated code can even be tailored to specific chemical problems, which may be useful for extremely large or complex calculations.

Figure 1 illustrates the overall architecture of the TCE, and the various components shown in the diagram are described in more detail below.

3.1. *Tensor expressions and the TCE language parser*

In order to illustrate the operation of the first two boxes in figure 1, consider the basic tensor expression

$$S_{abij} = \sum_{cefkl} A_{acik} B_{befl} C_{dfjk} D_{cdel}. \tag{1}$$

This equation might be rendered in the TCE's input language as shown in Listing 1. All indices appearing in TCE inputs are declared as being associated with a particular range of values (lines 4–5). Ranges must be

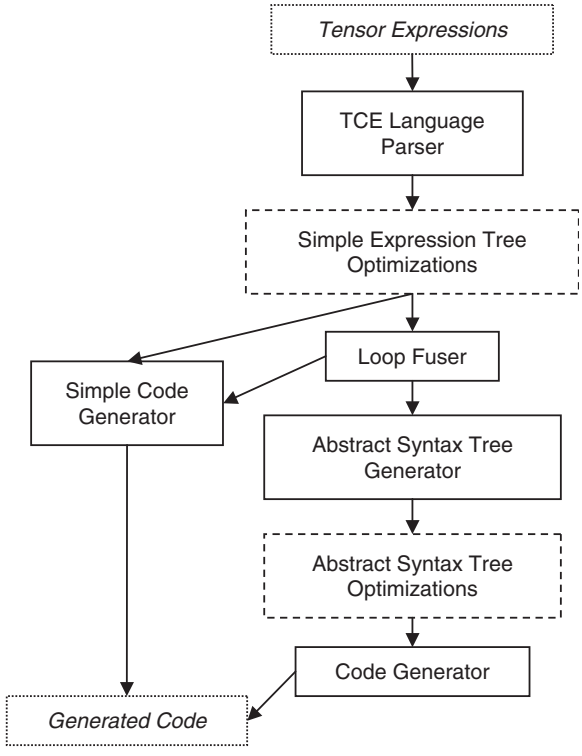

Figure 1. A general schematic representation of the architecture of the Tensor Contraction Engine software tools (both prototype and optimizing). "*Tensor Expressions*" and "*Generated Code*" boxes represent the inputs and outputs of the TCE tools. Dashed-line boxes represent one or more optimization modules which act on the simple expression tree or abstract syntax tree representations used within the TCE.

```
1  range V = 3000;
2  range O = 100;
3
4  index a,b,c,d,e,f : V;
5  index i,j,k,l : O;
6
7  mlimit = 100GB;
8
9  procedure P(in A[V,V,O,O], in B[V,V,V,O], in C[V,V,O,O], in D[V,V,V,O],
10             out S[V,V,O,O])=
11 begin
12   S[a,b,i,j] == sum[ A[a,c,i,k] * B[b,e,f,l] * C[d,f,j,k] * D[c,d,e,l],
13                      {c,e,f,k,l}];
14 end
```

Listing 1. TCE input corresponding to equation 1.

declared with a size (lines 1–2) to allow the compiler to compute the cost and resource requirements of each expression during the optimizations. These sizes amount to upper bounds – the generated code will work up to the indicated size, but may fail beyond it due to resource exhaustion. TCE `procedures` are named and declare both their input and output data (lines 9–10). Tensor expressions are written as explicit summations in notation reminiscent of that used by Mathematica [76] (lines 12–13). The `mlimit` declaration indicates the amount of main memory available on the target system. The TCE input language also allows external (i.e. not generated by the TCE) functions to be declared and used in tensor expressions, for example to obtain integrals.

The TCE input is parsed and converted into an expression tree, in which the tensor contraction expressions are represented as a binary tree, with nodes for each operator or summation in the expression and tensor elements (i.e. `A[a,c,i,k]`) as the leaves. This is the first of two ''internal representations'' used within the TCE, which can be transformed in systematic ways by various optimizations.

3.2. *Expression tree optimizations and loop fusion*

Operation minimization applies transformations based on the algebraic properties of commutativity and associativity of addition and multiplication, and the distributivity of multiplication over addition in order to obtain an equivalent form with the minimal scaling [77, 78]. For example, the four-fold contraction of equation 2, which is $O(o^4v^5)$ as written, would be transformed to the sequence of pairwise contractions costing $O(o^2v^4 + o^3v^3)$ shown in figure 2a. This optimization would also convert the usual $O(N^8)$ form in which a four-index integral transformation is typically written (as a single contraction involving five factors) into the $O(N^5)$ form in which it is usually implemented (a sequence of pairwise contractions). This optimization operates on the expression tree representation of the TCE input, and is represented by the upper dashed box in figure 1. At present, these transformations can be applied to all terms within a single statement. We are working to generalize this optimization so that it can work across multiple statements, which will allow recognization of common subexpressions and more effective factorization.

Because the operation minimization phase tends to introduce very large intermediate tensors which may exceed the available memory, *loop fusion* is used next to minimize the space required for temporaries (also known as *memory minimization*) [79]. This technique selectively moves loops for the generation of temporary tensors closer to where they are used, so that the temporary can be generated and consumed in subsections instead of having to evaluate the entire tensor, as illustrated in figure 2 (notice that in the fused code `I1f` is a scalar and `I2f` is rank-2 whereas in the unfused version, both are rank-4). Normally, the loop-fusion optimization would be applied so as to preserve the overall computational cost determined in the operation minimization step, but more aggressive fusion is also possible, with an increase in cost (effectively undoing some or all of the operation minimization). This may be useful, for example, if an important calculation is so large (or resources so constrained) that it cannot fit in the available memory and disk storage, and the user is willing to pay the additional computational cost.

Because loop fusion can interact strongly with subsequent optimizations, the loop fuser does not actually try to select the ''best'' set of loop fusions, but rather annotates the expression tree with information

$$I1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel}$$
$$I2_{bcjk} = \sum_{df} I1_{bcdf} \times C_{dfjk}$$
$$S_{abij} = \sum_{ck} I2_{bcjk} \times A_{acik}$$

(a) Formula sequence

```
I1=0; I2=0; S=0;
for b, c, d, e, f, l
[ I1bcdf += Bbefl Dcdel
for b, c, d, f, j, k
[ I2bcjk += I1bcdf Cdfjk
for a, b, c, i, j, k
[ Sabij += I2bcjk Aacik
```

(b) Direct implementation (unfused code)

```
S = 0;
for b, c
[ I1f = 0; I2f = 0;
  for d, f
  [ for e, l
    [ I1f += Bbefl Dcdel
    for j, k
    [ I2fjk += I1f Cdfjk
  for a, i, j, k
  [ Sabij += I2fjk Aacik
```

(c) Memory-reduced implementation (fused)

Figure 2. Example illustrating use of loop fusion for memory reduction based on the tensor contraction in equation 2.

about a set of the highest-ranking fusion options to allow for flexibility later in the optimization process.

3.3. *Abstract syntax tree generation and optimizations*

The expression tree used in the upper part of figure 1 is a very simple representation of the desired computations which allows only limited opportunities for optimization. Further optimization requires a more detailed representation in which the individual loops, I/O operations, and other features that must appear in the final generated code can be represented. This representation, known in computer science as an *abstract syntax tree* (AST) [80], is generated by the TCE from the output of the loop fusion operation.

The flexibility of the AST allows a much broader range of optimizations (represented collectively as the lower dashed box in figure 1). Similar representations are used within compilers for general-purpose languages, and there are many similar features in the types of optimizations available in general-purpose compilers and those being used in the TCE. An important difference, however, is that TCE optimizations can take advantage of the fact that the problem space is limited to electronic structure methods to perform transformations that would not be applicable in a more general context. At present, the primary optimizations available at this stage of the TCE's execution are:

- **Data distribution and partitioning:** In order to generate efficient parallel code, attention must be paid to the details of how the data is distributed across the machine. This component seeks distributions of the data that will minimize the parallel communication required to evaluate the tensor equations. Since the data distribution pattern affects the memory usage on the parallel machine, this component is closely coupled with the memory-minimization component [81].
- **Space-time transformation:** Depending on the method and the size of the chemical problem, the earlier loop-fusion optimization may be able to reduce memory usage so that the entire computation fits into main memory, memory plus available disk space, or it may not even be able to fit the calculation into the available storage at all. In the last case, the calculation cannot be performed unless it is possible to trade storage for recomputation of certain quantities; if the calculation requires the use of disk storage, there may be performance advantages to recomputation instead of storing on disk. The space-time transformation module systematically explores opportunities to trade storage space for recomputation to either fit the problem into available storage, or to improve performance. In the former case, if no satisfactory transformation is found, feedback is provided to the memory minimization module, causing it to seek a different solution through more aggressive loop fusion [82]. 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:** On modern computer architectures, the idea of blocking (also known as *tiling*) a calculation to make effective use of the CPU's cache memory is widely used to improve performance. Such optimizations are based on maximizing the *locality* of the computation, so that data can be brought into cache and reused as much as possible before it is evicted. For a disk-based calculation, the computer's main memory can be treated as the "cache" and the same ideas can be applied to optimizing the movement of data between disk and main memory. The data-locality optimization module determines the optimum tile sizes to obtain the best overall performance from the memory hierarchy [83, 84].

3.4. *Code generation*

Once all desired transformations and optimizations are complete, the output code must be generated. In the case of a general-purpose compiler this would be binary object code, but for simplicity, the TCE produces its output in a general purpose programming language (we have arbitrarily chosen Fortran). The code-generation phase of the TCE is depicted in two places in figure 1: the *Code Generator* at the bottom of the diagram, and the *Simple Code Generator* near the middle. The Simple Code Generator represents the option to skip the more complex optimizations which act on the detailed abstract syntax tree representation and generate code directly from the expression tree representation. This option reflects the way the TCE has been developed. The longer path, including all possible opportunities for optimization represents the structure of the ''optimizing TCE'' or o-TCE, while the shorter path represents the structure of the ''prototype TCE'' or p-TCE.

The optimizations of the o-TCE acting on the detailed AST representation are the most complex aspect of the development of the TCE – many of the optimizations currently implemented were newly developed as part of this project. The p-TCE, initially developed by So Hirata [45, 46], provides a simpler environment in which to explore certain aspects of the code generation problem, though of course it lacks much of the optimization capability of the o-TCE. In order for the o-TCE framework to be as extensible as possible, both within the area of many-body methods, and in the longer term beyond it, we are striving to carefully distinguish the origins of various code generation aspects and optimizations possible in many-body methods as arising from the mathematical properties of tensors, the physics of fermionic systems, or the chemical or physical nature of the system being studied, for example. In this respect, the p-TCE has proven very useful in allowing us to systematically capture and explore aspects of automatic code generation for many different electronic structure methods and implementation models without having to worry about the complexity of having fully general AST-based optimizations that will work universally.

For either code-generation module, the code being produced must be targeted to a particular environment. This includes how it is interfaced with an existing electronic structure package to obtain the integrals and other inputs required, as well as the specific parallel programming model. At present, we have chosen to focus on NWChem [39, 40] as the electronic structure package (though code generated by the p-TCE has also been interfaced with UTChem [85, 86]). For parallel computing, we are using the Global Array Toolkit [87–89], which is widely used in parallel electronic structure packages [39, 90–94]. Since, at present, the TCE is only capable of evaluating tensor contraction expressions themselves, the generated code must be wrapped up into a (hand-written) driver to handle the overall sequencing and iteration of the tensor expressions, and we also assume that various numerical and other libraries are available (such as the BLAS).

An important feature of this approach is the flexibility provided by the code-generation process. Retargeting the code to interface with a different electronic structure package, parallel programming model, or other variations in the environment is primarily a matter of modifying the code-generation portion of the TCE (though in some cases some optimizations may also need to change). Once the modifications are made to the code generator, an entire range of methods can be generated targeting the new environment. The resulting changes often suffuse the entire code, so hand implementations would require significant modification for each and every method separately. In the following section, we present an example which takes advantage of the flexibility of the code generation approach, as well as examples of some of the other optimizations implemented in the TCE.

4. Examples of TCE components in operation

In order to provide further insight into the operation and capabilities of the TCE, in this section we present examples of various modules of the TCE in operation: our algorithm for operation minimization, which is providing results very close to the best known manual CCSD implementation; the effect of loop fusion and data locality optimizations on an out-of-core four-index transformation; and finally, an example of a new parallel CCSD(T) algorithm implemented using the TCE.

4.1. *The operation-minimization optimization*

As discussed in the previous section, transforming the input equations into a form that has the appropriate computational scaling, often known as operation minimization or strength reduction, is one of the most critical optimizations in the TCE. When a method is being implemented by hand, these kinds of transformations are generally made on a largely empirical basis, informed by the developers experience, familiarity with the literature, etc., and tend to evolve and improve slowly over time. In an automatic code-generation environment like the TCE, the power of the computer can be used to perform a much more rigorous (and perhaps even a *complete*) search of the alternatives for operation minimization or other kinds of optimizations

Table 2. Comparison of the costs of various forms of the CCSD T_2 equation. The original input expressions (first line) served as the starting point for the direct-descent and random + descent optimization algorithms described in the text. These results are compared with the best known manual formulation of the equations, by Stanton *et al.* [97]. o and v are the size of the occupied and virtual orbital spaces, respectively, with $o + v = N$. Only the leading terms (N^6 and N^5) are shown; N^4 and smaller terms are neglected.

Equations	N^6	N^5
Original input	$\frac{1}{4}o^2v^4 + \frac{15}{2}o^3v^3 + \frac{11}{2}o^4v^2 +$	$ov^4 + \frac{19}{2}o^2v^3 + \frac{49}{2}o^3v^2 + 7o^4v$
TCE direct descent	$\frac{1}{4}o^2v^4 + 4o^3v^3 + \frac{1}{2}o^4v^2 +$	$2ov^4 + 8o^2v^3 + 9o^3v^2 + 3o^4v$
TCE random + descent	$\frac{1}{4}o^2v^4 + 4o^3v^3 + \frac{1}{2}o^4v^2 +$	$2ov^4 + 8o^2v^3 + 8o^3v^2 + 3o^4v$
Stanton *et al.*	$\frac{1}{4}o^2v^4 + 4o^3v^3 + \frac{1}{2}o^4v^2 +$	$ov^4 + 6o^2v^3 + 10o^3v^2 + o^4v$

in order to find the best one. Moreover, optimizations in the TCE can easily be applied to the whole range of methods for which the TCE can produce code, while in traditional software development, each method's implementation would have to be hand optimized separately.

Operation minimization acts on the expression-tree representation, which is a straightforward translation of the input equations into a binary tree of variables (i.e. tensor elements) and mathematical operations (i.e. multiplication, addition, summation). The input equations can be transformed into equivalent expressions through algebraic manipulations based on properties such as commutativity, distributivity, associativity, etc. Each variant would give rise to a new expression tree which differs from the original input form only with respect to the specific algebraic manipulations that have been performed on it. Each tree also has a unique cost associated with it based on the number and types of operations to be performed. (For example, $A \times B + A \times C$ costs $4N^3 + N^2$ operations if all matrices are $N \times N$, while $A \times (B+C)$ costs $2N^3 + N^2$.) The task of operation minimization is to search among all these equivalent expression trees to find the one with the minimum cost.

Unfortunately, the number of alternative representations grows very quickly with the complexity of the input equation, and the optimization is *NP-complete* [95], meaning that the known algorithms to solve it are exponential in cost, making it infeasible in general to perform an exhaustive search for the minimum. Instead, we use a heuristic algorithm to search for a solution approximating the global-minimum operation cost.

To understand the algorithm, consider each of the alternative expression trees above as a single node in a larger graph. Adjacent nodes differ from each other by a single algebraic transformation. Each node has a number of neighbors corresponding to every possible algebraic transformation that can be applied to every element of the tree. A simple search strategy would be to examine all the immediate neighbors of the starting node (i.e. those that differ by exactly one algebraic transformation somewhere in the expression tree) and choose the one with the least cost. The procedure is repeated at the new node, resulting in a *direct-descent* algorithm. The direct descent algorithm will find a *local* minimum in the graph, but there is no guarantee of finding the *global* minimum in this way. An improvement on this algorithm is to perform a number of trials in which the first (or first several) step is taken at random and then direct descent is used from there to find a local minimum. The final result of the algorithm is the minimum of the local minima found in each trial.

Preliminary results of applying these search strategies to the CCSD T_2 equations [96, 97] are shown in table 2. For the "random + descent" search algorithm, we perform 100 trials consisting of a single random step from the input equations followed by direct descent to a local minimum and taking the lowest result from all trials. As we can see, both search algorithms obtain results that are significantly better than the original input form, and the random + descent algorithm obtains a slightly lower coefficient for the o^3v^2 term. Comparing the search results with the best known manual CCSD formulation by Stanton and coworkers [97], we see that N^6 terms match. In the N^5 terms, the search results are fairly close to those of Stanton *et al.*, but somewhat higher. We consider these preliminary results to be very positive, especially when considering they were obtained with a completely automatic operation-minimization

algorithm which uses purely algebraic properties, without chemistry-specific information of any kind. Moreover, the manual result we are comparing with did not appear until fully nine years after the first implementation of the CCSD method appeared [96].

We are now exploring the use of this approach to operation minimization for CC methods with higher excitations, where much less effort has been expended to date in manually minimizing the operation counts. We are also working to enhance the optimization algorithm itself. Currently, it operates across all terms in a single statement, but ultimately it will be able to work on multiple statements to optimize an entire TCE procedure, or even multiple procedures, adding the capability to recognize and take advantage of common subexpressions across a set of equations. We are also exploring more exhaustive search strategies such as simulated annealing or genetic algorithms.

4.2. *Loop-fusion and Tiling optimizations for the four-index transformation*

As an example to illustrate the impact of the loop-fusion and optimized tiling transformations being implemented in the TCE, we consider the AO-to-MO integral transformation (virtual orbitals only).†

$$B(a,b,c,d) = \sum_{\mu,\nu,\lambda,\sigma} C(\sigma,d)C(\lambda,c)C(\nu,b)C(\mu,a)A(\mu,\nu,\lambda,\sigma)$$

where indices a–d denote virtual orbitals and μ–σ denote the full orbital space.

The operation-minimal way of computing B would be through the use of four steps, with temporary intermediate tensors $I1$, $I2$, and $I3$, as follows:

$$I1(a,\nu,\lambda,\sigma) = \sum_{\mu} C(\mu,a)A(\mu,\nu,\lambda,\sigma)$$

$$I2(a,b,\lambda,\sigma) = \sum_{\nu} C(\nu,b)I1(a,\nu,\lambda,\sigma)$$

$$I3(a,b,c,\sigma) = \sum_{\lambda} C(\lambda,c)I2(a,b,\lambda,\sigma)$$

$$B(a,b,c,d) = \sum_{\sigma} C(\sigma,d)I3(a,b,c,\sigma)$$

If the integrals are too large to fit in memory, tiling of the loops will be required, so that the tensors are processed in blocks that can fit within memory. A straightforward way of doing this is to directly tile the loops corresponding to each of the four steps, using uniform tile sizes on all dimensions. Thus, the loops for the first step would be tiled with the same tile size t along all indices μ, ν, λ, σ, and a. Due to the rank-4 tensors, the maximum tile size is limited to the order of the fourth root of memory size. Tile sizes may also be optimized individually [98, 99].

Table 3. Total disk I/O and execution times for code generated with the benefit of various optimizations, as described in the text. Results were obtained on a 900 MHz Itanium 2 system with 4 GB of memory (denoted M below), for 140 virtual orbitals and 150 AO orbitals.

Optimizations included and omitted	Total disk I/O I/O time (s)	Total execution time (s)
No fusion, tile size $= \sqrt[4]{\frac{M}{3}}$	1241	1957
No fusion, optimized tiling	748	1262
Fusion + optimized tiling	248	955

The overhead of disk I/O can be reduced by using loop fusion, so that the ranks of some of the intermediate tensors are reduced, allowing them to be completely memory resident to avoid disk I/O (see figure 2, Section 3.2, and [98, 99]). While it is impossible to perform loop fusions to reduce the ranks of all three temporaries simultaneously, there are a number of ways of using loop fusion to reduce the ranks of two of the three intermediates.

In table 3 we compare timings for three different implementations of this four-index transformation:

1. **No Fusion, Simple Tiling:** Neither loop-fusion nor tile-size optimization is enabled; equi-sized tiles along all dimensions are used, based on the 4th root of the memory size.
2. **No Fusion, Optimized Tiling:** No loop fusion is used (i.e. all intermediates $I1$, $I2$, and $I3$ are fully produced and written out to disk and then read back to be consumed), but the o-TCE tiling optimization is enabled, so that different combinations of tile sizes are explored and the best chosen.
3. **Fusion + Optimized Tiling:** The o-TCE loop-fusion and tiling optimizations are used to search amongst a large space of loop structures corresponding to different combinations of loop fusion and tile sizes.

It can be seen that the combined use of fusion and tiling optimizations results in code that has 80% less disk I/O than the version with the simple tiling, and a 66% reduction in total execution time.

4.3. *Code generation for a loop-fused replicated-data parallel CCSD(T) implementation*

A long-term goal of the TCE project is, given an accurate performance model for virtually any platform,

†Since spatial and permutational symmetry are not yet modeled in the o-TCE, the data presented here is for fully dense tensors, but the optimization approach is expected to be similarly effective for the computations that exploit symmetry.

to be able to generate near-optimal code – automatically taking into account the processor and network performance, memory and disk resources and other features. However since this is not yet possible, we have taken advantage of the code-generation capabilities of the TCE to produce a CCSD(T) [100] implementation designed for the capabilities of modestly-sized, low-cost commodity Beowulf cluster computers which are now widely accessible, even in the individual laboratories of many computational chemists. In this section, we outline our on going work in this area.

CCSD(T) has become a ''workhorse'' method of modern quantum chemistry because it provides a relatively low-cost way of introducing some triple-excitation effects based on a converged CCSD wavefunction. The development of parallel algorithms in which the data (t–amplitudes, integrals, etc.) are fully distributed across the system is extremely complex and finding a highly scalable distributed-data parallel formulation with the TCE is still some ways off. Nevertheless, it would be useful to be able to speed up such calculations using widely available commodity-cluster systems. By replicating the key data across all processes of the parallel computer, we can eliminate much of the complex data communications that would be required for a fully distributed algorithm. Instead we have a single, simple communication phase in each CCSD iteration in which partial contributions to the t–amplitude residuals are summed and the result redistributed to all processes. Loop-fusion techniques are used to insure that the CCSD iteration can take place without the need for additional communication of intermediates. The code for such a replicated-data algorithm is straightforward to generate using the TCE, and affords the opportunity to take advantage of modest numbers of CPUs to speed up calculations.

To illustrate our approach, consider a representative term from the CCSD equations,

$$R_{ij}^{ab} = \left[V_{kl}^{cd} T_{ij}^{cd} T_l^b + V_{ij}^{kb} \right] T_k^a \tag{2}$$

in which a contribution to the residual for the double excitation amplitudes, R_{ij}^{ab} is computed from integrals V_{ij}^{pa}, and amplitudes T_{ij}^{ab} and T_i^a, where $i \ldots l$ denote occupied orbitals, and $a \ldots d$ virtual orbitals.

In a straightforward implementation, intermediate terms in this equation would be calculated in their entirety and used as inputs to the next step of the calculation. In pseudocode, the typical implementation might look like figure 3.

However such an implementation is not attractive in a replicated-data parallel environment. Each process would compute contributions to the intermediates that would need to be summed up and redistributed to all processes before the next step of the algorithm could occur. This would significantly increase the amount and frequency of communication and synchronization. However we observe that in the term above, there is a common index `k` in all summations. Taking advantage of the concept of loop fusion, introduced in Section 3.2, the `k`–loop can be pulled to the outside, so that the intermediates `I1` and `I2` can essentially be treated as rank-3 tensors for fixed `k`. If the `k` values are distributed across the parallel processes, then each process can compute the intermediates entirely locally, without any need for communication, as shown in figure 4.

```
for k < l, i < j
[ I2(k,l,i,j) = sum (c<d) v(k,l,c,d) t(c,d,i,j)
for k,b, i<j
[ I1(k,b,i,j) = sum(l) I2(k,l,i,j)*t(b,l) + v(k,b,i,j)
for a<b,i<j
[ R(a,b,i,j) = sum (k) I1(k,b,i,j) * t(a,k)
```

Figure 3. Pseudocode for a simple implementation of equation 2. I1 and I2 denote intermediate quantities.

```
for k_tile
[ assign task to next available processor
  for l, i < j
  [ I2(l,i,j; k) = sum (c<d) v(k; l,c,d)t(c,d,i,j)
  for b, i<j
  [ I1(b,i,j; k) = sum(l) I2(l,i,j;k)*t(b,l) + v(b,i,j;k)
  for a<b, i<j
  [ R(a,b,i,j) += sum(k_withintile) I1(b,i,j; k) * t(a; k)
```

Figure 4. Pseudocode for a perfectly fused implementation of equation 2.

In this representation, we separate `k` from the other indices with a semicolon ("`;`") as a reminder that this index is fixed. The `k`–loop can be tiled according to the available memory and disk resources, so that `k` is not a single index value, but a small range. The likelihood of fitting the smaller, fused intermediates into memory is another significant advantage of this approach.

Coupled-cluster-type equations can always be factored in such a way that it is possible to perfectly fuse them and pull one or more indices to the outside of the entire term. A rule of thumb to obtain a suitable factorization is that the first intermediate (`I2` here) must always include the Hamiltonian term `v`, and that the indices that can be pulled out are the summation indices of the contraction of the final intermediate (`I1` here) to update the residual (`R`).

It is worth noting that taking advantage of the loop-fusion opportunity requires giving up the permutational symmetry `k < l` which was present in the original algorithm. Therefore, the fused algorithm includes some redundant computation. This is a common trade-off, and one which we are working on including in the o-TCE, so that the choice to use the permutational symmetry or take advantage of a loop fusion opportunity can be made based on the relative costs of the two alternatives.

The parallelization strategy for this approach is already indicated in the pseudocode above. The work is farmed out across the processors at the level of the `k`-loop, and each task can be carried out locally, without requiring communication. Using this parallelization strategy, it is possible, within a CCSD iteration, for each process to calculate its contribution to the *t*–amplitude residuals in an entirely local fashion using the replicated integrals and amplitudes, and the processes need not be synchronized during this computation. Once all local residual contributions are complete, they must be accumulated to form the complete residual, and then distributed back to all processes.

To reduce the replicated storage required for integrals, we have chosen to implement the terms involving integrals with four virtual orbital labels (referred to as the *abcd* term) in the AO basis. This involves back-transforming the *t*–amplitudes from the MO to the AO basis (at a cost of roughly $o^2v^2N + o^2vN^2$ operations for N AO basis functions), contracting with the pre-computed AO integrals, and transforming the result back to the MO basis. This transformation may be carried out as a parallel operation or the computation may simply be carried out redundantly local to each process. The parallel option tends to perform best with high-performance network interconnects, while on systems with slow networks the redundant sequential algorithm gives better overall performance. A secondary benefit of using the AO basis representation of the *abcd* term is that the resulting parallel tasks tend to be relatively small. By performing this step after all of the MO-basis steps in the CCSD iteration, these small tasks can be dynamically allocated to "even out" any load imbalance that may have occurred in the MO steps, which includes a small number of relatively large tasks.

The equations for the (T) correction can be summarized as:

$$T_{ijk}^{abc} = P(i/jk)P(a/bc)\sum_m T_{im}^{ab} V_{jk}^{cm} + \sum_e T_{ij}^{ae} V_{ek}^{bc}$$

$$\tilde{T}_{ijk}^{abc} = P(i/jk)P(a/bc)T_i^a V_{jk}^{bc}$$

$$E^{[4]} = \sum_{abcijk} \frac{T_{ijk}^{abc} T_{ijk}^{abc}}{D_{ijk}^{abc}}$$

$$E^{[5]} = \sum_{abcijk} \frac{T_{ijk}^{abc} \tilde{T}_{ijk}^{abc}}{D_{ijk}^{abc}}$$

Where P stands for permutation operator, e.g.:

$$P(i/jk)f(i,j,k) = f(i,j,k) - f(j,i,k) - f(k,j,i)$$

and D is the usual SCF energy denominator. The idea of loop fusion is used in practically all implementations of the perturbative triples correction to avoid storage of 6-index quantities, although it is usually not called by that name. As the indices *ijk* and *abc* occur in all terms, perfect loop fusion is possible. These indices are taken as the outer loops, and for a given batch of indices all contributions, including the various terms implied by the permutation operators, are accumulated in memory. This results in an algorithm which does not require any storage of T_3 amplitudes, as all fragments of the intermediate quantities can be held in memory and consumed immediately. In a replicated-data environment, the inputs to these equations are available in their entirety on all processors, and parallelization is a straightforward matter and can be carried out without communication except for the final accumulation of energy contributions from the individual processes.

It should be clear that the loop-fused replicated-data parallel CCSD(T) algorithm is significantly different in structure from the straightforward implementation of the method. Implementing our approach by hand would have required doing it from scratch rather than adapting an existing code. One of the virtues of the TCE, however, is that the code generation is just another module of the tool, and can be changed to target a different programming model and can be used to

Table 4. Cluster node configurations.

Cluster	Mpp2	Watsci
Processor	Intel itanium 2 (IA-64)	Intel celeron (IA-32)
Clock speed	1.5 GHz	2.0 GHz
CPUs per node	2	1
Network	Quadrics QsNet (Elan-3)	100 Mb Ethernet
Memory	8192 MB	256 MB
Local disk space	430 GB	10 GB

generate implementations of any many-body method for which the model is appropriate. We have made extensive use of the prototype TCE [45, 46] in implementing our new approach. Minor modifications were required to the generated code in various places, and the AO-basis *abcd* terms were hand-coded since support for generating AO-basis algorithms in the TCE is still under development. The current implementation of the AO-basis *abcd* term does not take advantage of molecular symmetry.

To illustrate this work, we present preliminary timing results for a CCSD(T) calculation on ethylene run on two different Beowulf clusters with very different configurations. One is the "Mpp2 (phase 2a)" system at the Molecular Science Computing Facility (MSCF) [101] at Pacific Northwest National Laboratory (PNNL), and the other is the "watsci" cluster at the University of Waterloo. Mpp2 was designed and configured to be used for large-scale parallel computing, while watsci was intended primarily as a processor farm for sequential jobs. Their node configurations are summarized in table 4. The test case is the lowest cation state of the ethylene molecule using the cc-pVTZ basis [102] and a UHF reference state, giving $o = 8/7$ (alpha/beta spin) and $v = 107/108$ (beta spin). The initial calculations were performed using D_{2h} point-group symmetry except for the AO-basis *abcd* term. The CCSD calculation took 11 iterations to converge, and the cost of the (T) correction is comparable to a single CCSD iteration (primarily due to the AO-based implementation of the *abcd* term).

Table 5 shows timings and parallel speedups for this test case, broken down by the different tasks performed by the code: the initial transformation of the AO integrals to the MO basis, the iterative solution of the CCSD equations (times are presented for a single iteration), and finally, the (T) correction. We ignore the SCF calculation required for the reference wavefunction for the CCSD(T). Timings are broken down into phases which are characteristic of replicated-data algorithms: work that is performed in parallel across all processes, work that is replicated and performed redundantly on each process, and communications phases in which partial results from each process are accumulated and the results broadcast back to all processes. On Mpp2, the MO/AO transformations in the *abcd* term are performed using a parallel algorithm, while on watsci, they are replicated. Also, on the Mpp2 cluster, only 2500 MB of memory per processor were actually used.

Looking at the results, we see that the scalability of the actual parallel work is very good in all aspects of the calculation on Mpp2 (15 out of 16 processors) and somewhat lower on watsci (10–13 processors out of 16). This portion of the effort is, in principle, perfectly parallel, so that any reduction from full scalability is due to load imbalance. Because of the poor network on watsci, the dynamic load balancing for the AO-*abcd* term was done at a very coarse grain, which helps to explain the poorer results on that cluster. The replicated steps, by definition, should not scale at all, and variations from a speedup of 1.0 are the result of variations in external factors such as the activation of operating system daemons. The communication-intensive steps actually show slow-downs in all cases. Although the data volume being communicated in this calculation is not large (approximately 21 MB/process), the fact that all processors are communicating simultaneously increases the likelihood of contention on the network. This is particularly true for Ethernet, where only one node can be sending data at a time on a given switch (all nodes on the watsci cluster are connected to a single switch). We are investigating these results and the details of the all-to-all communication to look for opportunities to improve the communication performance.

The overall results illustrate that although the parallel work itself scales very well, the poor scaling of the communications phase, along with the non-scalable replicated work can strongly impact the total performance. On Mpp2, the overall scaling is 8.8 out of 16 processors, while on watsci it is only 2.8. We can see that on both systems, the four-index transformation eventually dominates the overall calculation, even though it is only $O(N^5)$ (while the triples correction is $O(N^7)$). This is not an aspect of the calculation in which we have invested much effort so far, and there is certainly much room for improvement here.

Table 5. CCSD(T) calculation using the cc-pVTZ basis set for the Ethylene molecule in D_{2h} symmetry. Wall time in seconds, speedup factor with respect to one processor in parenthesis. For the Transformation and CCSD Iteration steps, timings are further broken down by the phase of the computation.

	Number of processors				
Algorithm step	1	2	4	8	16
Mpp2 cluster					
Transformation	699.3	382.1 (1.8)	210.9 (3.3)	132.5 (5.3)	99.4 (7.0)
Replicated	4.5	4.7 (0.9)	4.8 (0.9)	4.7 (1.0)	5.0 (0.9)
Parallel	640.1	336.3 (1.9)	163.8 (3.9)	82.5 (7.8)	42.2 (15.2)
Communication	12.1	13.8 (0.9)	20.3 (0.7)	27.2 (0.6)	32.5 (0.6)
CCSD iteration	83.5	42.1 (2.0)	21.9 (3.8)	12.0 (6.9)	7.6 (11.0)
Parallel	77.0	38.2 (2.0)	19.2 (4.0)	9.7 (7.9)	5.0 (15.5)
MO–AO (comm.)	6.0	3.4 (1.7)	2.2 (2.7)	1.7 (3.4)	1.8 (3.3)
Communication	0.4	0.5 (0.9)	0.5 (0.8)	0.6 (0.7)	0.8 (0.5)
(T) Correction	90.9	46.8 (1.9)	23.5 (3.9)	11.8 (7.7)	6.0 (15.1)
Overall CCSD(T)	1729	904.0 (1.9)	483.5 (3.6)	284.1 (6.1)	196.0 (8.8)
Watsci cluster					
Transformation	1075	756.4 (1.4)	708.5 (1.5)	852.9 (1.3)	982.4 (1.1)
Replicated	16.7	15.8 (1.1)	16.2 (1.0)	16.6 (1.0)	17.1 (1.0)
Parallel	934.3	491.6 (1.9)	263.6 (3.5)	144.8 (6.5)	71.0 (13.2)
Communication	51.8	156.8 (0.3)	288.6 (0.2)	496.4 (0.1)	658.4 (0.1)
CCSD iteration}	246.7	156.8 (1.6)	84.4 (2.9)	50.5 (4.9)	33.2 (7.4)
Parallel	240.6	148.0 (1.6)	75.7 (3.2)	40.1 (6.0)	19.7 (12.2)
MO–AO (repl.)	5.5	5.7 (1.0)	5.4 (1.0)	5.1 (1.1)	5.1 (1.1)
Communication	0.7	3.1 (0.2)	3.4 (0.2)	5.3 (0.1)	8.4 (0.1)
(T) Correction	207.2	119.9 (1.7)	68.0 (3.1)	36.6 (5.7)	20.8 (10.0)
Overall CCSD(T)	4049	2679 (1.5)	1760 (2.3)	1515 (2.7)	1453 (2.8)

Since in this example, the triples correction is very close in cost to that of a CCSD iteration, we also performed a series of calculations in C_1 symmetry on Mpp2 to provide an example in which the triples correction represents a much larger portion of the overall computation. These results are shown in table 6.

In this case, we observe much better overall scaling (80% parallel efficiency vs. 55% for the with-symmetry calculation on Mpp2). The triples correction dominates the total time for the calculation, and scales excellently.†

As a side note, it is interesting to observe that in this case, using molecular symmetry is not a major performance gain in some parts of the calculation. The CCSD iteration time is essentially unaffected, and the transformation slows down substantially when symmetry is used. The likely cause of this is that with symmetry, many of the blocks in the matrix multiplication used to implement these steps are too small to drive the processor to peak performance – the lower overall operation count is offset by the fact that the processor cannot perform them as efficiently. With appropriate performance models, we anticipate that this trade-off could be evaluated in advance and the TCE-generated code could choose whether it was advantageous to take advantage of symmetry or not.

In summary, we have taken advantage of the p-TCE's code generation capabilities to implement a new loop-fused replicated-data parallel CCSD(T) algorithm. Initial results indicate that the approach performs reasonably well on cluster computers. Additional effort is clearly needed to understand and improve the performance of the communication phases of the computation, as well as the transformation.

In the longer term, we anticipate that this particular type of experimentation with the TCE will become unnecessary. As has been mentioned several times, our ultimate goal with the o-TCE is to be able to generate code that accommodates a very broad range of

†We attribute the "super-linear" speedup observed here (4.2 vs 4.0 ideal) to the fact that the speedup computation is referenced to the four-processor time (the smallest configuration on which the calculation could be run) instead of the usual single-processor time.

Table 6. CCSD(T) calculation using the cc-pVTZ basis set for the Ethylene molecule in C_1 symmetry on the Mpp2 cluster. Wall time in seconds, speedup factor with respect to four processors in parenthesis. For Transformation and CCSD Iteration steps, timings are further broken down by the phase of the computation.

	Number of processors		
Algorithm step	4	8	16
Transformation	94.1 (1.0)	83.5 (1.1)	90.0 (1.0)
Replicated	5.7 (1.0)	6.0 (0.9)	5.2 (1.1)
Parallel	36.1 (1.0)	19.0 (1.9)	11.6 (3.3)
Communication	30.4 (1.0)	38.5 (0.8)	52.3 (0.7)
CCSD iteration	23.5 (1.0)	13.5 (1.7)	10.0 (2.4)
Parallel	20.2 (1.0)	10.2 (2.0)	6.1 (3.3)
MO–AO (comm.)	0.9 (1.0)	1.0 (1.0)	1.2 (0.8)
Communication	2.4 (1.0)	2.3 (1.0)	2.67 (0.9)
(T) Correction	1210 (1.0)	593.9 (2.0)	287.5 (4.2)
Overall CCSD(T)	1574 (1.0)	835.1 (1.9)	495.5 (3.2)

platforms based on their performance characteristics rather than forcing the user to select the algorithm to be used based primarily on intuition. The decision to use a fully distributed algorithm or a replicated-data one, decisions about which loops to fuse, where to take advantage of geometric and permutational symmetries, and other options can be evaluated automatically in order to provide the user with the best performance for the target platform and target problem. While this goal remains some ways off, the essential optimization and decision-making techniques are known and many have been outlined in this paper.

5. Current status and future plans

In its conception, the Tensor Contraction Engine is one of the most general and most complex efforts in automatic code generation undertaken to date in the chemistry community. Good progress has been made already, with the prototype TCE (p-TCE) capable of routine use and having already been used to generate parallel implementations of more than 25 different electronic structure methods, including CC, QCI, EOM-CC, and relativistic versions of many of them, thanks to the efforts of So Hirata [45–47]. We conservatively estimate that these implementations would have taken five years or more of effort to implement by hand, but they were done with the TCE in little more than the time it took to transfer the equations to the appropriate form. Many of these implementations are available in the latest versions of the NWChem [39, 40] and UTChem [85, 86] packages.

The capabilities of the optimizing TCE (o-TCE) lag somewhat those of the p-TCE because of the additional complexities of the desired optimization capabilities and the need to have a rigorous and detailed understanding of how they interact with characteristics such as permutational, spin, and spatial symmetries inherent in the chemistry and physics of the problem. The o-TCE is current capable of generating code for a broad range of methods and applying the various optimizations discussed in Section 3. However incorporation of permutational, spin, and spatial symmetries is presently underway. We also plan to revise the way the generated code interfaces with the host electronic structure package in order to make it easier to hook TCE-generated code to a broader range of host packages.

As has been mentioned, performance and parallel scalability of the generated code are central to our conception of the TCE. The optimization capabilities of the o-TCE are constantly improving as we gain more experience. The development of the tool has now progressed to a point where we are beginning to seriously examine performance issues in greater depth. A code generation-based environment like the TCE offers unprecedented benefits with respect to delivering high-performance code. With appropriate performance models for the target computer (accounting for memory-access costs and other factors), TCE optimizations can seek to minimize the actual execution time of the generated code, rather than just minimizing simpler metrics such as the number of floating-point operations. Moreover, the TCE can *predict* what the performance of the generated code should be, based on the performance model, so that deviations from the expected performance can be flagged and used to refine the performance model.

We plan to explore a broader range of parallel-programming models within the TCE, relating to both code-generation and performance issues. In the previous section we discussed a parallel-programming model that was targeted for computers with large memory/disk resources, but a poor parallel interconnect. Our long-term vision is a unified model that would be able to generate code for a broad range of parallel computers based on the computer's performance model and resource information. Systems with large memory and poor interconnects would naturally end up with TCE-generated implementations based on replicated data while those with better interconnects might get partly or fully distributed algorithms. The size of the chemical problem, also part of the input to the TCE, plays a role as well. For example, the TCE would be able to determine if the problem is too large for a replicated-data algorithm on the target system and instead produce a partly or fully distributed implementation if necessary.

The richest opportunities lie in the optimizations. Most of the optimizations currently available in the TCE were developed specifically for this project, and new and improved optimization algorithms are constantly being developed. There are also interesting research questions around how multiple optimizations interact, and how the order in which optimizations are applied affects the final results.

Finally, we are interested in broadening the TCE as far as possible within electronic structure theory, including, for example, support for local correlation approaches, and introducing the technology into other scientific domains. The recent renaissance of coupled-cluster methods for nuclear structure theory [103, 104] has given us our first target, but we anticipate many others as well. Optimizations and related technologies developed within the TCE can also be adapted into more general environments. For example, the data-locality optimization for managing out-of-core calculations could be introduced into a traditional Fortran compiler (with appropriate language extensions) to provide a tool that would greatly simplify the writing of general (not chemistry-specific) out-of-core algorithms [105].

6. Conclusions

We have described the Tensor Contraction Engine, a tool for the automatic generation of code for a broad spectrum of many-body methods. The overall design of the TCE follows closely the architecture of traditional optimizing compilers, and has been developed by a team of both computational chemists and computer scientists working in close collaboration.

The TCE addresses a number of different needs within the high-end electronic structure community. The nature of code generation itself, and the fact that it is driven by a high-level language very similar to the way the equations are expressed when they are derived, allows a wide variety of methods to be expressed and implemented quickly. In this sense, we anticipate that the TCE will become a catalyst for new method development by making the process so much simpler and quicker compared to hand coding.

The ability to automatically generate parallel implementations is valuable for expanding the capabilities of researchers who want to tackle larger or more expensive problems, but who lack experience with parallelization of chemistry software, or lack the time to do it by hand. For many applications of this type, an extremely high-performance parallelization is not necessary to provide benefits to the user, and indeed this has already been observed in the methods already implemented using the p-TCE, as mentioned in the previous section. Many of these methods received their first-ever parallel implementation via the TCE.

Finally, we view the TCE as a tool for high-end computational chemistry, where in order to solve the problems of interest, there is a need to extract the utmost performance from the wide variety of architectures now available at the edge of high-performance parallel computing. As we improve the code generation and optimization capabilities, we anticipate that it will become possible to routinely generate code with the TCE that meets or exceeds the performance of hand-coded implementations in most cases.

With some further effort, we anticipate that the TCE will be able to produce near-optimal high-performance parallel code for a wide range of electronic structure methods in a fraction of the time required for hand coding. We believe that the widespread availability and use of tools of this kind will facilitate a shift of effort away from the complex and often tedious programming challenges associated with advanced electronic structure methods and toward a focus on development of the methods themselves and the chemical problems they can address.

Acknowledgements

We gratefully acknowledge the many contributions of So Hirata (PNNL) to the development of the TCE, including innumerable discussions as well as the original development of the p-TCE.

This work has been supported in part (1) by the National Science Foundation, grants CHE-0121676, CHE-0121706, CCR-0073800 and EIA-9986052; (2) by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725; and (3) by Discovery Grant 262942-03 from the Natural Sciences and Engineering Research Counsel of Canada (NSERC).

This research was performed in part using the Molecular Science Computing Facility in the William R. Wiley Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the U.S. Department of Energy's Office of Biological and Environmental Research and located at the Pacific Northwest National Laboratory. PNNL is operated for the Department of Energy by Battelle.

References

[1] R.J. Bartlett, G.D. Purvis. *Int. J. quantum Chem.*, **14**, 561 (1978).

[2] J.A. Pople, R. Krishnan, H.B. Schlegel, J.S. Binkley. *Int. J. quantum Chem.*, **14**, 545 (1978).
[3] P.R. Taylor, G.B. Bacskay, N.S. Hush, A.C. Hurley. *J. chem. Phys.*, **69**, 1971 (1978).
[4] M. Musial, S.A. Kucharski, R.J. Bartlett. *J. chem. Phys.*, **116**, 4382 (2002).
[5] I. Lindgren. *Int. J. quantum Chem. Symp.*, **12**, 33 (1978).
[6] B. Jeziorski, H. Monkhorst. *Phys. Rev. A*, **24**, 1668 (1981).
[7] D. Mukherjee, S. Pal. *Adv. Quantum Chem.*, **20**, 292 (1989).
[8] J. Paldus, X.Z. Li. *Adv. Chem. Phys.*, **110**, 1 (1999).
[9] U.S. Mahapatra, B. Datta, D. Mukherjee. *J. chem. Phys.*, **110**, 6171 (1999).
[10] U.S. Mahapatra, B. Datta, D. Mukherjee. *J. phys. Chem. A*, **103**, 1822 (1999).
[11] J. Pittner, P. Nachtigall, P. Carsky, J. Masik, I. Hubac. *J. chem. Phys.*, **110**, 10275 (1999).
[12] X. Li, J. Paldus. *J. chem. Phys.*, **107**, 6257 (1997).
[13] X. Li, J. Paldus. *J. chem. Phys.*, **110**, 2844 (1999).
[14] X. Li, J. Paldus. *J. chem. Phys.*, **113**, 9966 (2000).
[15] M. Nooijen, R.J. Bartlett. *J. chem. Phys.*, **104**, 2652 (1996).
[16] S.A. Perera, H. Sekino, R.J. Bartlett. *J. chem. Phys.*, **101**, 2186 (1994).
[17] S.A. Perera, M. Nooijen, R.J. Bartlett. *J. chem. Phys.*, **104**, 3290 (1996).
[18] H.J. Monkhorst. *Int. J. quantum Chem. Symp.*, **11**, 421 (1997).
[19] D. Mukherjee, R.K. Moitra, A. Mukhopadhyay. *Mol. Phys.*, **33**, 955 (1977).
[20] D. Mukherjee, P.K. Mukherjee. *Chem. Phys.*, **39**, 325 (1979).
[21] H. Koch, H.J.A. Jensen, P. Jorgensen, T. Helgaker. *J. chem. Phys.*, **93**, 3345 (1990).
[22] K. Hirao, H. Nakatsuji. *J. chem. Phys.*, **69**, 4535 (1978).
[23] J.F. Stanton, R.J. Bartlett. *J. chem. Phys.*, **98**, 7029 (1993).
[24] R.J. Bartlett, J.F. Stanton. *Rev. Comp. Chem.*, **5**, 65 (1994).
[25] J.F. Stanton, R.J. Bartlett, C.M.L. Rittby. *J. chem. Phys.*, **97**, 5560 (1992).
[26] M. Nooijen, R.J. Bartlett. *J. chem. Phys.*, **106**, 6441 (1997).
[27] M. Nooijen, R.J. Bartlett. *J. chem. Phys.*, **106**, 6449 (1997).
[28] M. Nooijen, R.J. Bartlett. *J. chem. Phys.*, **107**, 6812 (1997).
[29] O. Christiansen, J. Gauss, J. Stanton. *Chem. Phys. Letters*, **292**, 437 (1998).
[30] J. Gauss, O. Christiansen, J. Stanton. *Chem. Phys. Letters*, **296**, 117 (1998).
[31] H. Koch, O. Christiansen, R. Kobayashi, P. Jørgensen, T. Helgaker. *Chem. Phys. Lett.*, **228**, 233 (1994).
[32] H. Koch, A. Sanchez de Meras, T. Helgaker, O. Christiansen. *J. chem. Phys.*, **104**, 4157 (1996).
[33] C. Hampel, H.-J. Werner, *J. chem. Phys.*, **104**, 6286 (1996).
[34] M. Schütz, H.-J. Werner. *J. chem. Phys.*, **114**, 661 (2001).
[35] M. Schütz. *J. chem. Phys.*, **116**, 8772 (2002).
[36] G.E. Scuseria, P.Y. Ayala. *J. chem. Phys.*, **111**, 8330 (1999).
[37] H. Koch, A. Sánchez de Merás. *J. chem. Phys.*, **113**, 508 (2000).
[38] H. Koch, A. Sánchez de Merás, T.B. Pedersen. *J. chem. Phys.*, **118**, 9481 (2003).
[39] T.P. Straatsma, E. Aprà, T.L. Windus, E.J. Bylaska, W. de Jong, S. Hirata, M. Valiev, M. Hackler, L. Pollack, R. Harrison, M. Dupuis, D.M.A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, A.A. Auer, E. Brown, G. Cisneros, G. Fann, H. Früchtl, J. Garza, K. Hirao, R. Kendall, J. Nichols, K. Tsemekham, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K. Dyall, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffee, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, M. Rosing, G. Sandrone, M. Stave, H. Taylor, G. Thomas, van J. Lenthe, A. Wong, Z. Zhang. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6*, Pacific Northwest National Laboratory, Richland, Washington 99352, USA, 2004, http://www.emsl.pnl.gov/docs/nwchem/.
[40] R.A. Kendall, E. Aprà, D.E. Bernholdt, E.J. Bylaska, M. Dupuis, G.I. Fann, R.J. Harrison, J. Ju, J.A. Nichols, J. Nieplocha, T.P. Straatsma, T.L. Windus, A.T. Wong. *Comp. Physics Comm.*, **128**, 260 (2000).
[41] C. Janssen, E. Seidl, M. Colvin. Object-Oriented Implementation of Parallel Ab Initio Programs, In *ACS Symposium Series, Parallel Computing in Computational Chemistry*, Vol. 592, American Chemical Society, Washington, D.C. (1995).
[42] C.L. Janssen, I.M. Nielsen, B., M.E. Colvin. Parallel Processing for ab Initio Quantum Mechanical Methods, in *Encyclopedia of Computational Chemistry*, John Wiley & Sons, Chichester, UK (1998).
[43] MPQC homepage, http://www.mpqc.org.
[44] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill Inc., New York (1991).
[45] S. Hirata. *J. phys. Chem. A*, **107**, 9887 (2003).
[46] S. Hirata. *J. chem. Phys.*, **121**, 51 (2004).
[47] S. Hirata. private communication.
[48] BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/.
[49] LAPACK – Linear Algebra PACKage, http://www.netlib.org/lapack/.
[50] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. *LAPACK Users' Guide*, 3rd edition, Society for Industrial and Applied Mathematics (SIAM), Philadelphia (1999).
[51] C.E. Dykstra, P.G. Jaisen. *Chem. Phys. Lett.*, **109**, 388 (1984).
[52] C.E. Dykstra. *J. chem. Phys.*, **82**, 4120 (1985).
[53] A.E. Kondo, P. Piecuch, J. Paldus. *J. chem. Phys.*, **104**, 8566 (1996).
[54] P. Piecuch, J. Paldus. *J. Math. Chem.*, **21**, 51 (1997).
[55] P.J. Knowles, K. Somasundram, N.C. Handy, K. Hirao. *Chem. Phys. Lett.*, **113**, 8 (1985).
[56] N.C. Handy, P.J. Knowles, K. Somasundram. *Theor. Chim. Acta*, **68**, 87 (1985).
[57] S. Hirata, R.J. Bartlett. *Chem. Phys. Lett.*, **321**, 216 (2000).
[58] M. Kàllay, P. Surjàn. *J. chem. Phys.*, **113**, 1359 (2000).
[59] J. Olsen. *J. chem. Phys.*, **113**, 7140 (2000).
[60] S. Hirata, M. Nooijen, R.J. Bartlett. *Chem. Phys. Lett.*, **326**, 255 (2000).

[61] S. Hirata, M. Nooijen, R.J. Bartlett. *Chem. Phys. Lett.*, **328**, 459 (2000).
[62] K. Hald, P. Jorgensen, J. Olsen, M. Jaszunski. *J. chem. Phys.*, **115**, 617 (2001).
[63] S. Hirata, M. Nooijen, I. Grabowski, R.J. Bartlett. *J. chem. Phys.*, **114**, 3919 (2001).
[64] W. Duch. *J. phys. A*, **18**, 3283 (1985).
[65] M. Kàllay, P. Surjàn. *J. chem. Phys.*, **115**, 2945 (2001).
[66] M. Kàllay, P.G. Szalay, P.R. Surjàn. *J. chem. Phys.*, **117**, 980 (2002).
[67] M. Kàllay, J. Gauss, P. Szalay. *J. chem. Phys.*, **119**, 2991 (2003).
[68] M. Kàllay, J. Gauss. *J. chem. Phys.*, **120**, 6841 (2004).
[69] C.L. Janssen, H.F. Schaefer III. *Theor. Chim. Acta*, **79**, 1 (1991).
[70] X. Li, J. Paldu. *J. chem. Phys.*, **101**, 8812 (1994).
[71] M. Nooijen, V.L. Lotrich. *J. Mol. Struct.-THEOCHEM*, **547**, 253 (2001).
[72] M. Nooijen. *Int. J. Mol. Sci.*, **3**, 656 (2002).
[73] J.T. Fermann, E.F. Valeev. Libint: Machine-Generated Library for Efficient Evaluation of Molecular Integrals over Gaussians, http://www.ccmst.gatech.edu/evaleev/libint/ (2003).
[74] R.C. Whaley, A. Petitet, J.J. Dongarra. Parallel Computing 27, 3, 2001, Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).
[75] ATLAS homepage, http://math-atlas.sourceforge.net/.
[76] Wolfram Inc. Research, Mathematica, version 5.0, http://www.wolfram.com (2000).
[77] C. Lam, P. Sadayappan, R. Wenger. *Parallel Processing Letters*, **7**, 157 (1997).
[78] C. Lam, P. Sadayappan, R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines, in *Proc. of Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, SIAM, Philadelphia (1997).
[79] C. Lam, D. Cociorva, G. Baumgartner, P. Sadayappan. Optimization of memory usage and communication requirements for a class of loops implementing multi-dimensional integrals, in *Proc. of Twelfth LCPC Workshop*, Springer, Berlin, New York (1999).
[80] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass. (1986).
[81] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Memory-Constrained Communication Minimization for a Class of Array Computations, in *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC '02)*, College Park, Maryland (2002).
[82] D. Cociorva, G. Baumgartner, C.P. Lam, J.R. Sadayappan, M. Nooijen, D. Bernholdt, R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations, in *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, p. 177–186 (2002).
[83] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization, in *Proc. of the Int. Conf. on High Performance Computing*, Vol. 2228, pp. 237–248, Springer-Verlag (2001).
[84] D. Cociorva, J. Wilkins, C. Lam, G. P.S. Baumgartner, J. Ramanujam. Loop optimization for a class of memory-constrained computations, in *Proc. of the Fifteenth ACM International Conference on Supercomputing (ICS'01)*, pp. 500–509 (2001).
[85] T. Yanai, M. Kamiya, Y. Kawashima, T. Nakajima, H. Nakano, Y. Nakao, H. Sekino, J. Paulovic, T. Tsuneda, S. Yanagisawa, K. Hirao. UTChem 2004, Department of Applied Chemistry, School of Engineering, University of Tokyo, Japan, http://utchem.qcl.t.u-tokyo.ac.jp.
[86] T. Yanai, H. Nakano, T. Nakajima, T. Tsuneda, S. Hirata, Y. Kawashima, Y. Nakao, M. Kamiya, H. Sekino, K. Hirao. UTChem – A Program for *ab inito* Quantum Chemistry, in *Computational Science – ICCS 2003*, P.M. Sloot, A., D. Abramson, A.V. Bogdanov, J.J. Dongarra, A.Y. Zomaya, Y.E. Gorbachev (Ed.), Vol. 2660, of *Lecture Notes in Computer Science*, pp. 84–95, Berlin, Heidelberg, Springer-Verlag (2003).
[87] J. Nieplocha, R.J. Harrison, R.J. Littlefield. Global arrays: a portable programming model for distributed memory computers, in *Supercomputing*, pp. 340–349 (1994).
[88] J. Nieplocha, R.J. Harrison, R.J. Littlefield. *The Journal of Supercomputing*, **10**, 169 (1996).
[89] Pacific Northwest National Laboratory, Global Array Toolkit homepage, http://www.emsl.pnl.gov/docs/global/ (2003).
[90] H.-J. Werner, P.J. Knowles, R. Lindh, M. Schütz, P. Celani, T. Korona, F.R. Manby, G. Rauhut, R.D. Amos, A. Bernhardsson, A. Berning, D.L. Cooper, M.J.O. Deegan, A.J. Dobbyn, F. Eckert, C. Hampel, G. Hetzer, A.W. Lloyd, S.J. McNicholas, W. Meyer, M.E. Mura, A. Nicklass, P. Palmieri, R. Pitzer, U. Schumann, H. Stoll, A.J. Stone, R. Tarroni, T. Thorsteinsson. *MOLPRO, version*, 2002.6, a package of ab initio programs, 2003, see http://www.molpro.net.
[91] H. Lischka, T. Mueller. The Columbus Parallel CI Program Project, http://www.itc.univie.ac.at/hans/Columbus/columbus parallel.html (2004).
[92] Molcas 6, http://www.teokem.lu.se/molcas/.
[93] Q-Inc. Chem, Q-Chem, http://www.q-chem.com/ (2004).
[94] M. Guest, Computing for Science, http://www.dl.ac.uk/CFS/cfs.html (2004).
[95] M. Garey, D. Johnson. *Computers And Intractability: A Guide to the Theory of NP-completeness*, W.H. Freeman and Company, New York (1979).
[96] G.D. Purvis III, R.J. Bartlett. *J. chem. Phys.*, **76**, 1910 (1982).
[97] J.F. Stanton, J. Gauss, J.D. Watts, R.J. Bartlett. *J. chem. Phys.*, **94**, 4334 (1991).
[98] A. Bibireata, S. Krishnan, G. Baumgartner, D. Cociorva, C.-C. Lam,, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, V. Choppella. Memory-Constrained Data Locality Optimizations for Tensor Contractions, in *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, College Station, Texas (2003).
[99] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C.-C. Lam,, P. Sadayappan, J. Ramanujam,

D.E. Bernholdt, V. Choppella. Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms, in *Proc. of 10th Annual International Conference on High Performance Computing (HiPC)*, Springer Verlag (2003).
[100] K. Raghavachari, G.W. Trucks, J.A. Pople, M. Head-Gordon. *Chem. Phys. Lett.*, **157**, 479 (1989).
[101] W.R. Wiley. Environmental Molecular Science Laboratory, Pacific Northwest National Laboratory, Molecular Science Computing Facility, http://mscf.emsl.pnl.gov/.
[102] T.H. Dunning. *J. chem. Phys.*, **90**, 1007 (1989).
[103] K. Kowalski, D.J. Dean, M. Hjorth-Jensen, T. Papenbrock, P. Piecuch. *Phys. Rev. Lett.*, **92**, 132501 (2004).
[104] D.J. Dean, M. Hjorth-Jensen. *Phys. Rev. C*, **69**, 054320 (2004).
[105] D.E. Bernholdt, J. Nieplocha, P. Sadayappan. Raising the Level of Programming Abstraction in Scalable Programming Models, in *Proc. of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004), Madrid, Spain*, IEEE Computer Society (2004).