A massively parallel tensor contraction framework for coupled-cluster computations

Edgar Solomonik
Devin Matthews
Jeff Hammond
John Stanton
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-143
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-143.html

August 2, 2014

Copyright © 2014, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A massively parallel tensor contraction framework for coupled-cluster computations

Edgar Solomonik[1], Devin Matthews[2], Jeff R. Hammond[3], John F. Stanton[2], and James Demmel[1,4]

solomon@eecs.berkeley.edu, dmatthews@utexas.edu, jeff_hammond@acm.org, jfstanton@mail.utexas.edu, demmel@eecs.berkeley.edu

(1) *Department of EECS, University of California, Berkeley, USA*
(2) *Department of Chemistry and Biochemistry, University of Texas, Austin*
(3) *Parallel Computing Lab, Intel Corp.*
(4) *Department of Mathematics, University of California, Berkeley, USA*

August 2, 2014

Abstract

Precise calculation of molecular electronic wavefunctions by methods such as coupled-cluster requires the computation of tensor contractions, the cost of which has polynomial computational scaling with respect to the system and basis set sizes. Each contraction may be executed via matrix multiplication on a properly ordered and structured tensor. However, data transpositions are often needed to reorder the tensors for each contraction. Writing and optimizing distributed-memory kernels for each transposition and contraction is tedious since the number of contractions scales combinatorially with the number of tensor indices. We present a distributed-memory numerical library (Cyclops Tensor Framework (CTF)) that automatically manages tensor blocking and redistribution to perform any user-specified contractions. CTF serves as the distributed memory contraction engine in Aquarius, a new program designed for high-accuracy and massively-parallel quantum chemical computations. Aquarius implements a range of coupled-cluster and related methods such as CCSD and CCSDT by writing the equations on top of a C++ templated domain-specific language. This DSL calls CTF directly to manage the data and perform the contractions. Our CCSD and CCSDT implementations achieve high parallel scalability on the BlueGene/Q and Cray XC30 supercomputer architectures showing that accurate electronic structure calculations can be effectively carried out on top of general distributed memory tensor primitives.

1 Introduction

Quantum chemistry is the field of science focused on the application of quantum mechanics to the study of electrons in molecules. Among the most common methods of quantum chemistry are many-body (QMB) methods, which attempt to explicitly solve the Schrödinger equation using a variety of models. The explicit treatment of electrons in molecules leads to a steep computational cost, which is nonetheless often of polynomial complexity, but with the benefit of systematic improvement achieved through appropriate elaborations of the models. The coupled-cluster (CC) family of methods [54, 5, 11] is currently the most popular QMB method in chemistry due to its high accuracy, polynomial time and space complexity, and systematic improvability. Coupled-cluster excels in the description of molecular systems due to its ability to accurately describe electron correlation – the dynamic effect of each electron on the others. In simpler (and hence cheaper) methods, electron correlation is either neglected in favor of an averaged interaction (as in self-consistent field theory [43, 39]) or approximated by an assumed functional form as in DFT [38], while

correlation is treated explicitly in CC methods for pairs of electrons (CCSD), triples (CCSDT), and so on in a systematically improvable way. Additionally CC is rigorously size-extensive [5] and easily extensible to excited states [51], derivatives [45, 52], and properties [33]. This paper focuses on the fundamental kernels of coupled-cluster calculations – tensor contractions – and documents a completely new set of parallel algorithms implemented as a distributed-memory tensor contraction framework, Cyclops Tensor Framework (CTF) [1]. CTF has enabled coupled-cluster with excitations of up to three electrons (CCSDT) to scale on state-of-the-art architectures while achieving a high degree of efficiency in computation, communication and storage.

Any tensor contraction can be performed via a series of index reorderings and matrix multiplications. Parallel matrix multiplication is a well-studied problem and existing algorithms are capable of achieving high efficiency for the problem on a variety of scales/architectures. In the field of dense linear algebra, optimized numerical libraries have achieved success and popularity by exploiting the efficiency of primitives such as matrix multiplication to provide fast routines for matrix operations specified via a high-level interface. CTF raises the level of abstraction to provide contraction routines which employ library-optimized matrix multiplication calls to maintain efficiency and portability. In addition to contractions, CTF provides optimized distributed data transposes which can reorder tensor indices and extract sub-tensors. This capability allows CTF to dynamically make data-decomposition decisions, which maintain load-balance and communication efficiency throughout the execution of any given tensor contraction. The high-dimensional blocking used in CTF permits the capability to exploit tensor symmetry to lower computational and/or memory costs of contractions whenever possible.

We expose the generality of this framework in the Aquarius quantum chemistry program [2] via a concise interface that closely corresponds to Einstein notation, capable of performing arbitrary dimensional contractions on symmetric tensors. This interface is a domain specific language well-suited to theoretical chemists. To demonstrate correctness and performance we have implemented coupled-cluster methods with single, double, and triple excitations (CCSDT) using this infrastructure.

The contributions of this paper are

- a structured communication pattern for tensor contractions
- a cyclic tensor decomposition for symmetric tensors
- an automatic topology-aware mapping framework for tensor contractions
- a load-balanced blocking scheme for symmetric tensors
- optimized redistribution kernels for symmetric tensors
- an expressive and compact tensor domain specific language (DSL)
- a scalable implementation of coupled-cluster up to triple excitations

In this paper, we will start by detailing previous and related work in Section 2 and overviewing coupled-cluster theory at the start of Section 3. We give examples of some of the bottleneck contractions in coupled-cluster singles and double (CCSD) and singles, doubles, and triples (CCSDT), along with a preview of the computational mechanisms and tensor mappings which optimize these contractions. After presenting the theory, we will discuss the architecture of Cyclops Tensor Framework. Section 4 discusses how tensors are decomposed into blocks and how redistribution kernels migrate data between blocked layouts as well

[1] Cyclops Tensor Framework is publicly available under a BSD license: `https://github.com/solomonik/ctf`

[2] A pre-alpha version of Aquarius is publicly available under the New BSD license: `https://github.com/devinamatthews/aquarius`

as allow sparse input/output. We will show how CTF dynamically makes mapping decisions and performs contractions in Section 5. In Section 6, we evaluate CTF by measuring its performance for CCSD and CCSDT on Blue Gene/Q and Cray XC30 architectures, which show good weak scaling and outperform other codes designed for similar computations, namely NWChem [8].

2 Previous work

In this section, we provide an overview of existing applications and known algorithms for distributed memory CC and tensor contractions. We also discuss parallel numerical linear algebra algorithms which will serve as motivation and building blocks for the design of Cyclops Tensor Framework.

2.1 NWChem and TCE

NWChem [8] is a computational chemistry software package developed for massively parallel systems. NWChem includes an implementation of CC that uses tensor contractions, which are of interest in our analysis. We will detail the parallelization scheme used inside NWChem and use it as a basis of comparison for the Cyclops Tensor Framework design.

NWChem uses the Tensor Contraction Engine (TCE) [20, 6, 16], to automatically generate sequences of tensor contractions based on a diagrammatic representation of CC schemes. TCE attempts to form the most efficient sequence of contractions while minimizing memory usage of intermediates (computed tensors that are neither inputs nor outputs). We note that TCE or a similar framework can function with any distributed library which actually executes the contractions. Thus, TCE can be combined with Cyclops Tensor Framework since they are largely orthogonal components. However, the tuning decisions done by such a contraction-generation layer should be coupled with performance and memory usage models of the underlying contraction framework. In addition, one of the present authors (D.A.M) is working on a new, more flexible generator for CC contractions which could be more tightly coupled to CTF.

To parallelize and execute each individual contraction, NWChem employs the Global Arrays (GA) framework [34]. Global Arrays is a partitioned global-address space model (PGAS) that allows processors to access data (via explicit function calls, e.g., Put, Get and Accumulate) that may be laid out physically on a different processor. Data movement within GA is performed via one-sided communication, thereby avoiding synchronization among communicating nodes, while accessing distributed data on-demand. NWChem performs different block tensor sub-contractions on all processors using GA as the underlying communication layer to satisfy dependencies and obtain the correct blocks. NWChem uses dynamic load balancing among the processors because the work associated with block-sparse representation of tensors within GA is not intrinsically balanced. Further, since GA does not explicitly manage contractions and data redistribution, the communication pattern resulting from one-sided accesses is often irregular. The dynamic load-balancer attempts to alleviate this problem, but assigns work without regard to locality or network topology. Cyclops Tensor Framework eliminates the scalability bottlenecks of load imbalance and irregular communication, by using a regular decomposition which employs a structured communication pattern especially well-suited for torus network architectures.

2.2 ACES III and SIAL

The ACES III package uses the SIAL framework [31, 14] for distributed memory tensor contractions in coupled-cluster theory. Like the NWChem TCE, SIAL uses tiling to extract parallelism from each tensor contraction. However, SIAL has a different runtime approach that does not require one-sided communication, but rather uses intermittent polling (between tile contractions) to respond to communication requests, which allows SIAL to be implemented using MPI two-sided communication.

2.3 MRCC

MRCC [25] is unique in its ability to perform arbitrary-order calculations for a variety of CC and related methods. Parallelism is enabled to a limited extent by either using a multi-threaded BLAS library or by parallel MPI features of the program. However, the scaling performance is severely limited due to highly unordered access of the data and excessive inter-node communication. MRCC is currently the only tenable solution for performing any type of CC calculation which takes into account quadruple and higher excitations. MRCC uses a string-based approach to tensor contractions which originated in the development of Full CI codes [27, 36]. In this method, the tensors are stored using a fully-packed representation, but must be partially unpacked in order for tensor contractions to be performed. The indices of the tensors are then represented by index "strings" that are pre-generated and then looped over to form the final product. The innermost loop contains a small matrix-vector multiply operation (the dimensions of this operation are necessarily small, and become relatively smaller with increasing level of excitation as this loop involves only a small number of the total indices). The performance of MRCC is hindered by its reliance on the matrix-vector multiply operation, which is memory-bandwidth bound. Other libraries, including NWChem and our approach, achieve better cache locality by leveraging matrix multiplication.

2.4 QChem

The QChem [47] quantum chemistry package employs libtensor [15], a general tensor contraction library for shared-memory architectures. The libtensor framework exploits spatial and permutational symmetry of tensors, and performs tensor blocking to achieve parallelization. However, libtensor does not yet provide support for distributed memory tensor contractions and redistributions, as done by CTF. The libtensor module in QChem also contains a tensor contraction DSL somewhat similar to our own. One of the main differences between the two DSLs is that in libtensor, the index symmetry of the output tensor is implicitly described by the symmetry of the inputs and any explicit antisymmetrization operations to be performed. In our DSL, it is the other way around in that the antisymmetrization operators are implicitly specified by the symmetry of the inputs and output. We feel that the latter approach gives a more convenient and compact representation of the desired operation, as well as a more robust one in that antisymmetrization operators must be specified for each contraction, while the output tensor structure must be specified only once. In addition, the use of repeated and common indices in the tensor index labels and operator overloading in our approach is more general and flexible than providing functions for each type of common tensor operation as in libtensor.

2.5 Other tensor contraction libraries

There are other tensor contraction libraries, which are not part of standard chemistry packages previously mentioned in this paper. One concurrently-developed distributed memory tensor library effort is given by Rajbahandri et al [42], and shares many similarities with our work. This library employs similar matrix multiplication primitives (SUMMA and 3D algorithms) for distributed tensor contractions and mapping of data onto torus networks. A few of the important differences between our work and this framework are overdecomposition and the redistribution mechanisms. Recent studies have also demonstrated the efficacy of scheduling many different contractions simultaneously within coupled-cluster [29], an approach that is particularly useful for higher order coupled-cluster methods. Our paper focuses on parallel algorithms for the execution of a single tensor contraction, leaving it for future work to integrate this single-contraction parallelization with a second layer of multi-contraction parallelism.

There have also been efforts for efficient implementation of tensor contractions for coupled-cluster which do not focus on distributed memory parallelism. Hanrath and Engels-Putzka [18] give a sequential framework which performs tensor transpositions to efficiently exploit threaded matrix multiplication

primitives within tensor contractions. Parkhill and Head-Gordon [37] as well as Kats and Manby [26] give sequential implementations of sparse tensor contraction libraries, targeted at coupled-cluster methods which exploit spatial locality to evaluate a sparse set of interactions. Support for parallel sparse tensor contractions is not within the scope of this paper, but is being pursued as an important direction of future work since many of the parallel algorithmic ideas discussed in this paper extend to sparse tensors.

2.6 Matrix multiplication algorithms

Since tensor contractions are closely related to matrix multiplication (MM), it is of interest to consider the best known distributed algorithms for MM. Ideally, the performance achieved by any given tensor contraction should approach the efficiency of matrix multiplication, and generally the latter is an upper bound. Given a matrix multiplication or contraction that requires F/p multiplications on p processors, with M words of memory on each processor, it is known that some processor must communicate at least

$$W = \Omega\left(\frac{F}{p \cdot \sqrt{M}}\right) \tag{1}$$

words of data [22, 4, 21]. If the matrices are square and of size S, and there is enough memory for only $O(1)$ copies of the matrices ($S = \Theta(M \cdot p)$), matrix multiplication can be performed on a 2D grid of processors in a communication-optimal fashion. In particular, blocked Cannon's algorithm [9] and SUMMA [1, 53] achieve the bandwidth cost

$$W_{2D} = O\left(\frac{F}{\sqrt{p \cdot S}}\right),$$

which matches Equation 1 for $S = \Theta(M \cdot p)$. Given an unlimited amount of memory, it is possible to replicate the data to reduce the communication cost, which is realized by a 3D matrix multiplication algorithm [32, 12, 1, 2, 23] with a resulting bandwidth cost of

$$W_{3D} = O\left(\frac{F}{p^{2/3} \cdot \sqrt{S}}\right).$$

So, 3D algorithms lower the communication cost over 2D algorithms by a factor of $p^{1/6}$ and are in fact optimal independent of the amount of available memory [2, 3] (provided there is enough memory to perform the algorithm).

In practice, most applications run with some bounded amount of extra available memory. 2.5D algorithms minimize communication cost for any amount of physical memory [50]. In particular, if all operand tensors or matrices are of size $S = \Theta(M \cdot p/c)$, where $c \in [1, p^{1/3}]$, the communication cost achieved by 2.5D matrix multiplication is

$$W_{2.5D} = O\left(\frac{F}{\sqrt{p \cdot c \cdot S}}\right),$$

which is a factor of $\sqrt{c}$ smaller than the 2D algorithm (when $c = p^{1/3}$, the 2.5D algorithm reduces to a 3D algorithm). The additional tunable parallelization dimension provides an important additional parameter for mapping onto torus network architectures as demonstrated in Ref. [49].

Additional algorithmic considerations arise when we consider matrix multiplication of non-square matrices [44, 13]. In particular, it is possible to move any two of the three matrices in 2D algorithms, and to replicate only one of three in 3D algorithms. To reduce communication, it is best to move the smaller matrices, as the size of the matrices differs when they are non-square. The case of rectangular matrices is particularly relevant to tensor contractions in coupled-cluster calculations, which are performed on tensors with different edge-lengths and dimension, and therefore of varying size. Cyclops Tensor Framework attempts to avoid communication by exploiting replication for tensors of any given size by dynamically selecting a parallel decomposition with the least estimated communication cost.

3 Coupled-cluster Methods

Coupled-cluster [54, 5, 11] is an iterative process, where in each iteration, the new set of amplitudes $\mathbf{T}'$ is computed from the amplitudes from the previous iteration $\mathbf{T}$ and from the (fixed) Hamiltonian $\mathbf{H} = \mathbf{F} + \mathbf{V}$. The diagonal elements of the one-particle Hamiltonian $\mathbf{F}$ are separated out as a factor $-\mathbf{D}$, giving a final schematic form similar to a standard Jacobi iteration (although $\mathbf{V}$ still contains diagonal elements),

$$\mathbf{T}' = \mathbf{D}^{-1}\left[(\mathbf{F}' + \mathbf{V})(\mathbf{1} + \mathbf{T} + \frac{1}{2}\mathbf{T}^2 + \frac{1}{6}\mathbf{T}^3 + \frac{1}{24}\mathbf{T}^4)\right]_c.$$

The expansion of the exponential operator is complete at fourth order due to the fact that the Hamiltonian includes only one- and two-particle (two- and four-index) parts and must share an index with each $\mathbf{T}$ operator (this is the meaning of the c subscript). $\mathbf{F}'$, $\mathbf{V}$, and $\mathbf{T}$ are defined in terms of fundamental tensors by

$$\begin{aligned}
\mathbf{F}' &= (1-\delta_{ab})f_b^a + f_i^a + f_a^i + (1-\delta_{ij})f_j^i, \\
\mathbf{V} &= v_{cd}^{ab} + v_{ci}^{ab} + v_{bc}^{ai} + v_{bj}^{ai} + v_{ij}^{ab} + \\
&\quad v_{ab}^{ij} + v_{jk}^{ai} + v_{ak}^{ij} + v_{kl}^{ij}, \\
\mathbf{T} &= \mathbf{T_1} + \mathbf{T_2} + \cdots + \mathbf{T_n} \\
&= t_i^a + t_{ij}^{ab} + \cdots + t_{i_1\ldots i_n}^{a_1\ldots a_n},
\end{aligned} \tag{2}$$

where the $abcdef\ldots$ indices refer to virtual (unoccupied) orbitals while $ijklmn\ldots$ refer to occupied orbitals. The amplitude ($\mathbf{T}$) and integral ($\mathbf{F}'$ and $\mathbf{V}$) tensors are anti-symmetric with respect to permutation of indices which are both virtual or occupied and in the same (up or down) position, e.g. we have

$$v_{ij}^{ab} = -v_{ij}^{ba} = v_{ji}^{ba} = -v_{ji}^{ab}.$$

The contractions which must be done can be derived using either algebraic or diagrammatic techniques [5], however the result (denoted as $\mathbf{Z}$, for coupled-cluster this is the residual in an iterative Jacobi procedure for the solution of $\mathbf{T}$) is a sum of contractions exemplified by

$$z_i^a = \frac{1}{2}\sum_{efm} v_{ef}^{am} t_{im}^{ef}$$

$$z_{ij}^{ab} = \frac{1}{4}\sum_{efmn} v_{ef}^{mn} t_{ij}^{ef} t_{mn}^{ab}.$$

where the summation index f is not to be confused with the one-electron integral tensor. Contractions which involve multiple $\mathbf{T}$ tensors are factored into a sequence of contractions involving one or more intermediates, such that each contraction is a binary tensor operation.

The equations, as written above, are termed the "spin-orbital" representation in that the indices are allowed to run over orbitals of either α or β spin, while only amplitudes with certain combinations of spin are technically allowed. Some programs use this representation directly, checking each amplitude or block of amplitudes individually to determine if it is allowed (and hence should be stored and operated upon). However, an alternative approach is the use of the spin-integrated equations where each index is explicitly

spin-α, $abij\ldots$, or spin-β, $\bar{a}\bar{b}\bar{i}\bar{j}\ldots$. For example, the second contraction above becomes,

$$\begin{aligned} z_{ij}^{ab} &= \frac{1}{4}\sum_{efmn} v_{ef}^{mn} t_{ij}^{ef} t_{mn}^{ab}, \\ z_{i\bar{j}}^{a\bar{b}} &= \sum_{ef\bar{m}\bar{n}} v_{e\bar{f}}^{m\bar{n}} t_{i\bar{j}}^{e\bar{f}} t_{m\bar{n}}^{a\bar{b}}, \\ z_{\bar{i}\bar{j}}^{\bar{a}\bar{b}} &= \frac{1}{4}\sum_{\bar{e}\bar{f}\bar{m}\bar{n}} v_{\bar{e}\bar{f}}^{\bar{m}\bar{n}} t_{\bar{i}\bar{j}}^{\bar{e}\bar{f}} t_{\bar{m}\bar{n}}^{\bar{a}\bar{b}}, \end{aligned}$$

While the number of contractions is increased, the total amount of data which must be stored and contracted is reduced compared to a naive implementation of the spin-orbital method, and without the overhead of explicit spin-checking.

The spin-integrated Hamiltonian elements (f and v tensors) and amplitudes (t tensors) also have implicit permutational symmetry. Indices which appear together (meaning either both upper or lower indices of a tensor) and which have the same spin and occupancy may be interchanged to produce an overall minus sign. In practice this allows the amplitudes to be stored using the symmetric packing facilities built into CTF.

3.1 Domain Specific Language for Tensor Operations

Aquarius provides an intuitive domain specific language for performing tensor contractions and other tensor operations. This interface is implemented using operator overloading and templating in C++, with the end result that tensor contractions can be programmed in the exact same syntax as they are defined algebraically,

$$z_{i\bar{j}}^{a\bar{b}} = \sum_{ef\bar{m}\bar{n}} v_{e\bar{f}}^{m\bar{n}} t_{i\bar{j}}^{e\bar{f}} t_{m\bar{n}}^{a\bar{b}}$$

$$\Downarrow$$

$$\texttt{W}[\text{“}\texttt{MnIj}\text{”}] = \texttt{V}[\text{“}\texttt{MnEf}\text{”}] * \texttt{T}[\text{“}\texttt{EfIj}\text{”}];$$
$$\texttt{Z}[\text{“}\texttt{AbIj}\text{”}] = \texttt{W}[\text{“}\texttt{MnIj}\text{”}] * \texttt{T}[\text{“}\texttt{AbMn}\text{”}];$$

This interface naturally supports all types of tensor operations, not just contraction. The number and placement of the unique indices implicitly define the operation or operations which are to be performed. For example, the repetition of an index within an input tensor which does not appear in the output tensor defines a trace over that index. Similarly, an index which appears in all three tensors defines a type of “weighting” operation while an index which appears multiple times in the input and once in the output will operate on diagonal or semi-diagonal elements of the input only. The weighting operation deserves special attention as it is required in CC to produce the new amplitudes $\mathbf{T}'$ from $\mathbf{Z} = \mathbf{H}e^{\mathbf{T}}$,

$$\mathbf{T}' = \mathbf{D}^{-\mathbf{1}}\mathbf{Z}$$

$$\Downarrow$$

$$\texttt{Dinv}[\text{“}\texttt{AbIj}\text{”}] = 1/\texttt{D}[\text{“}\texttt{AbIj}\text{”}];$$
$$\texttt{T}[\text{“}\texttt{AbIj}\text{”}] = \texttt{Dinv}[\text{“}\texttt{AbIj}\text{”}] * \texttt{Z}[\text{“}\texttt{AbIj}\text{”}];$$
or simply,
$$\texttt{T}[\text{“}\texttt{AbIj}\text{”}] = \texttt{Z}[\text{“}\texttt{AbIj}\text{”}]/\texttt{D}[\text{“}\texttt{AbIj}\text{”}];$$

Additionally, Equation-of-Motion CC (EOM-CC) and many other related techniques have terms that require computation of only the diagonal elements of a tensor contraction or require replication of the result along one or more dimensions, both of which can be expressed easily and succinctly in this interface. For example, the diagonal tensor elements used in EOMIP-CCSD include terms such as,

$$\begin{aligned} \bar{H}^{ai\bar{j}}_{ai\bar{j}} &\leftarrow W^{i\bar{j}}_{i\bar{j}} \quad \text{and} \\ \bar{H}^{ai\bar{j}}_{ai\bar{j}} &\leftarrow \sum_{\bar{e}} v^{i\bar{j}}_{a\bar{e}} t^{a\bar{e}}_{i\bar{j}}, \end{aligned}$$

which can be expressed in Aquarius as,

```
Hbar["AIj"] += W["IjIj"];
Hbar["AIj"] += V["IjAe"] * T["AeIj"];
```

3.2 Application to CCSD and CCSDT

The CCSD model [41], where $\mathbf{T} = \mathbf{T_1} + \mathbf{T_2}$ (i.e. $n = 2$ in Equation 2), is one of the most widely used coupled-cluster methods as it provides a good compromise between efficiency and accuracy, and is straightforward to derive and implement. In particular, CCSD is only slightly more computationally expensive than the simpler CCD method [46] but provides greater accuracy, especially for molecular properties such as molecular structure and those derived from response theory. Extending this to CCSDT [30, 35], where $\mathbf{T} = \mathbf{T_1} + \mathbf{T_2} + \mathbf{T_3}$ ($n = 3$), gives an even more accurate method (often capable of what is commonly called "chemical accuracy"), at the expense of a steeper scaling with system size and vastly more complicated equations. Formally, CCSD and CCSDT have leading-order costs of $O(n_o^2 n_v^4)$ and $O(n_o^3 n_v^5)$, where n_o and n_v are the number of occupied and virtual orbitals, respectively.

The most expensive contractions in each method are those which involve the highest order $\mathbf{T}$ tensor ($\mathbf{T_2}$ for CCSD, $\mathbf{T_3}$ for CCSDT), multiplied by either the integrals $\mathbf{V}$ or (equivalently in terms of computational cost) by a 4-dimensional intermediate tensor. For example, in CCSD the most expensive set of contractions is the "particle-particle ladder" term,

$$\begin{aligned} z^{ab}_{ij} &= \frac{1}{2}\sum_{ef} v^{ab}_{ef} t^{ef}_{ij}, \\ z^{a\bar{b}}_{i\bar{j}} &= \sum_{e\bar{f}} v^{a\bar{b}}_{e\bar{f}} t^{e\bar{f}}_{i\bar{j}}, \\ z^{\bar{a}\bar{b}}_{\bar{i}\bar{j}} &= \frac{1}{2}\sum_{\bar{e}\bar{f}} v^{\bar{a}\bar{b}}_{\bar{e}\bar{f}} t^{\bar{e}\bar{f}}_{\bar{i}\bar{j}}. \end{aligned}$$

When the spins of ab and ij are the same (the first and third contractions), the antisymmetry of these index pairs and of the ef index pair is conserved in all of the tensors. This is taken advantage of in CTF by folding each of these index pairs into a single index of length $n(n-1)/2$. Additionally, in the second contraction the $a\bar{b}$, $i\bar{j}$, and $e\bar{f}$ pairs can similarly be combined after the tensor has been properly remapped, if doing so will reduce the communication volume. Folding will be discussed in-depth in the next section. Then, the contractions reduce to (relatively) simple matrix multiplication. However, the situation is more complicated for the second-most expensive set of contractions (the "ring" term, designated as such by the

appearance of the associated diagrammatic representation),

$$
\begin{aligned}
z_{ij}^{ab} &= P_j^i P_b^a \left\{ \sum_{em} w_{ej}^{mb} t_{im}^{ae} + \sum_{\bar{e}\bar{m}} w_{j\bar{e}}^{b\bar{m}} t_{i\bar{m}}^{a\bar{e}} \right\}, \\
z_{i\bar{j}}^{a\bar{b}} &= \sum_{em} w_{e\bar{j}}^{m\bar{b}} t_{im}^{ae} + \sum_{\bar{e}\bar{m}} w_{\bar{e}\bar{j}}^{\bar{m}\bar{b}} t_{i\bar{m}}^{a\bar{e}} + \sum_{\bar{e}m} w_{i\bar{e}}^{m\bar{b}} t_{m\bar{j}}^{a\bar{e}} + \\
&\quad \sum_{e\bar{m}} w_{e\bar{j}}^{a\bar{m}} t_{i\bar{m}}^{e\bar{b}} + \sum_{em} w_{ie}^{am} t_{m\bar{j}}^{e\bar{b}} + \sum_{\bar{e}\bar{m}} w_{i\bar{e}}^{a\bar{m}} t_{\bar{m}\bar{j}}^{\bar{e}\bar{b}}, \\
z_{\bar{i}\bar{j}}^{\bar{a}\bar{b}} &= P_{\bar{j}}^{\bar{i}} P_{\bar{b}}^{\bar{a}} \left\{ \sum_{\bar{e}\bar{m}} w_{\bar{e}\bar{j}}^{\bar{m}\bar{b}} t_{\bar{i}\bar{m}}^{\bar{a}\bar{e}} + \sum_{em} w_{ej}^{m\bar{b}} t_{m\bar{i}}^{e\bar{a}} \right\},
\end{aligned}
$$

where w_{ej}^{mb} is an intermediate and the permutation operator $P_b^a[\ldots a \ldots b \ldots] = [\ldots a \ldots b \ldots] - [\ldots b \ldots a \ldots]$. Two complications arise here: first, the symmetric index groups, such as ab, $\bar{i}\bar{j}$, etc. in the output, or ae, $\bar{m}\bar{j}$, etc. in the input (none of the index groups on w is antisymmetric since the indices are of different occupancies) are "broken", as they do not always appear together. Second, the first and third equations require explicit antisymmetrization through the P_b^a operators. Broken symmetries, discussed more in-depth in the next section, prevent folding of the index pair and are handled in CTF either by unpacking the broken symmetries or by performing a sequence of triangular contractions on the packed data. Similarly the antisymmetrization from P_b^a for example can be incorporated into the handling of the ab symmetry either by explicitly antisymmetrizing while repacking the data if it has been unpacked, or by altering the sign appropriately for individual packed contractions.

These same types of contractions are again the most expensive ones for CCSDT, except that now the $\mathbf{T_3}$ tensor is involved. For the particle-particle ladder term,

$$
\begin{aligned}
z_{ijk}^{abc} &= P_c^{ab} \frac{1}{2} \sum_{ef} v_{ef}^{ab} t_{ijk}^{efc}, \\
z_{ij\bar{k}}^{ab\bar{c}} &= \frac{1}{2} \sum_{ef} v_{ef}^{ab} t_{ij\bar{k}}^{ef\bar{c}} + P_b^a \sum_{e\bar{f}} v_{e\bar{f}}^{b\bar{c}} t_{ij\bar{k}}^{ae\bar{f}}, \\
z_{i\bar{j}\bar{k}}^{a\bar{b}\bar{c}} &= \frac{1}{2} \sum_{\bar{e}\bar{f}} v_{\bar{e}\bar{f}}^{\bar{b}\bar{c}} t_{i\bar{j}\bar{k}}^{a\bar{e}\bar{f}} + P_{\bar{c}}^{\bar{b}} \sum_{e\bar{f}} v_{e\bar{f}}^{a\bar{b}} t_{ij\bar{k}}^{e\bar{f}\bar{c}}, \\
z_{\bar{i}\bar{j}\bar{k}}^{\bar{a}\bar{b}\bar{c}} &= P_{\bar{c}}^{\bar{a}\bar{b}} \frac{1}{2} \sum_{\bar{e}\bar{f}} v_{\bar{e}\bar{f}}^{\bar{a}\bar{b}} t_{\bar{i}\bar{j}\bar{k}}^{\bar{e}\bar{f}\bar{c}},
\end{aligned}
$$

where P_c^{ab} generalizes P_b^a by interchanging (antisymmetrizing) both a and b with c. Similarly to the case in CCSD, the contractions with sums over ef with the same occupancy represent preserved symmetries in ef and, where ef and c are of opposite occupancies, in ab as well. These index pairs can then be folded. However, in the first and fourth equations, the three-index symmetry over abc is broken, and explicit antisymmetrization is required. In the case of a three-index symmetry, though, the unpacking can be partial, in that the preserved symmetry of ab can be retained and these indices folded. Multiple triangular contractions (or more precisely, the higher-dimensional analogue) can also be performed on the fully-packed data.

In the Aquarius tensor DSL, optimizations due to preserved symmetries, handling of broken symmetries, and explicit permutation operators are implied by the symmetry relations in the specification of the input and output tensors. This results in significant reduction and simplification of the required code. For example, if two antisymmetric matrices are multiplied together and the result is then antisymmetrized (which is signaled

by specifying that the output matrix is itself antisymmetric), the result, in terms of fully packed storage requires six separate operations to handle the broken symmetry and explicit antisymmetrization. All of these operations can be represented by a single contraction call in the Aquarius interface if the output tensor is specified to have antisymmetry,

$$\begin{aligned} C[ab] &= P_b^a A[ac] \times B[cb], \\ C[a<b] &= \sum_c \big\{ A[a<c]B[c<b] - A[a<c]B[b<c] \\ &\quad -A[c<a]B[c<b] - A[b<c]B[c<a] \\ &\quad +A[b<c]B[a<c] + A[c<b]B[c<a] \big\} \end{aligned}$$

$$\Downarrow$$

```
/* A, B, and C are antisymmetric */
C["ab"] = A["ac"] * B["cb"];
```

An interface layer to automatically produce the necessary spin-integrated contractions has also been implemented, so that the code can be written entirely in terms of the simple spin-orbital quantities. This means that, for example, each of the contraction sets above is actually written as a single and fairly simple contraction. With these simplifications, the total amount of code to perform a single CCSD iteration is only 41 lines, and to perform a CCSDT iteration is only 62 lines, whereas traditional implementations generally require thousands of lines of hand-written code and automatically-generated programs generally consist of many thousands of lines (for example the TCE-based CCSDT module in NWChem is almost 12,000 lines).

3.3 Higher-order coupled-cluster

Higher order CC methods (CCSDTQ [28], CCSDTQP, etc.) are theoretically very similar to CCSD and CCSDT, however, several important computational distinctions arise. First, as the order increases, the highest set of $\mathbf{T_n}$ amplitudes becomes vastly larger than the Hamiltonian elements and the other $\mathbf{T}$ amplitudes. The computation time in terms of FLOPS is dominated by just a few contractions involving this largest amplitude set. However, the sheer number of small contractions which must be done in addition can still dominate the wall time if they are not performed as efficiently or do not parallelize as well. Thus, the efficiency of small tensor contractions and strong-scalability of the parallel algorithm become relatively much more important for higher order CC, than for the simpler models such as CCSD.

Second, since the total memory and/or disk space available for the computation is effectively constant, high orders of CC effectively demand the use of a smaller number of occupied and virtual orbitals. This shrinks the length of each tensor dimension, threatening vectorization and increasing indexing overhead. While extremely short edge lengths could still cause excessive overhead in this scheme (due to the padding needed to fill out the fixed-length tiles), good performance should be retained in most circumstances. In addition, the ratio of floating point operations to tensor size (and hence memory operations) necessarily shrinks for higher orders of CC for the same reasons, so that choosing distributions which minimize data movement of the largest tensors is critical. CTF accomplishes this by searching over the possible mappings and automatically choosing the optimal one in terms of data movement and memory cost.

4 Algorithms for tensor blocking and redistribution

CTF decomposes tensors into blocks which are cyclic sub-portions of the global tensor and assigns them to processors in a regular fashion. The partitioning is contraction-specific and tensor data is transferred between different distributions when necessary. Further, it is necessary to allow the user to modify, enter,

and read the tensor data in a general fashion. In this section, we explain how the cyclic blocking scheme works and give data redistribution algorithms which are designed to shuffle the data around efficiently.

4.1 Cyclic tensor blocking

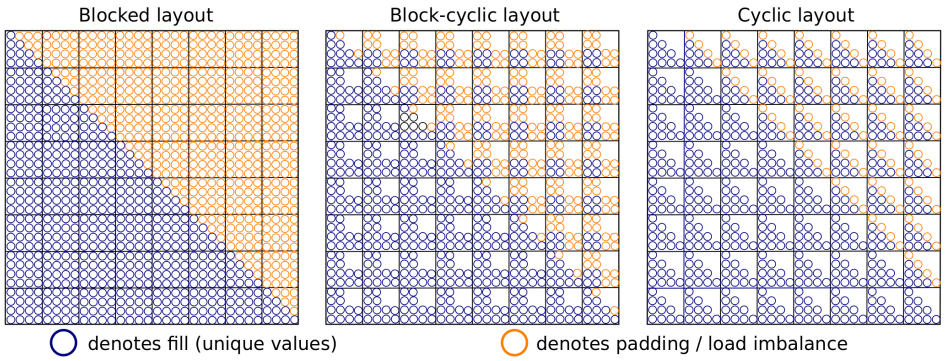


Figure 1: The load-imbalance incurred or padding necessary for blocked, block-cyclic, and cyclic layouts.

A blocked distribution implies each processor owns a contiguous piece of the original tensor. In a cyclic distribution, a cyclic phase defines the periodicity of the set of indices whose elements are owned by a single processor. For example, if a vector is distributed cyclically among 4 processors, each processor owns every fourth element of the vector. For a tensor of dimension d, we can define a set of cyclic phases $(p_1, p_2, \ldots, p_d)$, such that processor $P_{i_1,i_2,\ldots,i_d}$ owns all tensor elements whose index $(j_1, j_2, \ldots, j_d)$ satisfies

$$j_k \equiv i_k \bmod p_k$$

for all $k \in \{1, \ldots, d\}$ and where p_k gives the length of the processor grid in the $k-th$ dimension. A block-cyclic distribution generalizes blocked and cyclic distributions, by distributing contiguous blocks of any size b cyclically among processors. Cyclic decompositions are commonly used in parallel numerical linear algebra algorithms and frameworks such as ScaLAPACK (block-cyclic) [7] and Elemental (cyclic) [40]. Our method extends this decomposition to tensors.

Cyclops Tensor Framework employs a cyclic distribution in order to preserve packed symmetric full structure in sub-tensors, minimize padding, and generate a completely regular decomposition, susceptible to classical linear algebra optimizations. Each processor owns a cyclic sub-tensor, where the choice of cyclic phases in each dimension has the same phase for all symmetric indices. By maintaining the same cyclic phase in each dimension, the algorithm ensures that each of the sub-tensors owned by any processor has the same fill structure as the whole tensor. For instance, CTF might decompose the integral tensor $\mathbf{V}$ which is anti-symmetric in two index pairs, into 36 blocks $\hat{\mathbf{V}}_{w_1w_2w_3w_4}$, where $w_1, w_2 \in \{0, 1\}$ and $w_3, w_4 \in \{0, 1, 2\}$ so that

$$v_{ij}^{ab} \in \hat{\mathbf{V}}_{w_1w_2w_3w_4} : \forall v_{ij}^{ab} \in \mathbf{V}, a \equiv w_1 \bmod 2,\ b \equiv w_2 \bmod 2, a < b \\ i \equiv w_3 \bmod 3,\ j \equiv w_4 \bmod 3, i < j.$$

Each sub-tensor $\hat{\mathbf{V}}_{w_1w_2w_3w_4}$ has the same structure as the unique part of $\mathbf{V}$ (v_{ij}^{ab} for $a < b, i < j$), though some blocks have extra elements (e.g. $\hat{\mathbf{V}}_{0101}$ is larger than $\hat{\mathbf{V}}_{1101}$ since $v_{ij}^{aa} = 0$). However, it is important to clarify that while structure is preserved, cyclicity does not preserve symmetry. If we drop the conditions $a < b$ and $i < j$ from the definition of the $\hat{\mathbf{V}}_{w_1w_2w_3w_4}$ block, only the blocks $\hat{\mathbf{V}}_{w_1w_1w_3w_3}$ will have the same symmetry as $\mathbf{V}$ (the same would be true if the blocks were selected contiguously rather than cyclically).

The cyclic blocks differ in fill, but padding along the diagonals of symmetric indices allows all the blocks to be defined in the same shape. Figure 1 demonstrates the padding required to store a lower-triangular matrix in a 4-by-4 cyclic layout. For comparison, we also show the amount of padding one would need to preserve identical block structure in block-cyclic and contiguously-blocked layouts. The cyclic layout is the only one able to preserve the fill structure and does not require a significant amount of padding. Preserving a regular layout, regardless of symmetry, also necessitates padding to make the tensor edge lengths divisible by the processor grid edge lengths. In a cyclic layout, the amount of padding required due to symmetry and divisibility grows with the number of processors, presenting a potential limit to strong scaling (e.g. if the number of processors is equal to the number of tensor elements, cyclic and blocked layouts are the same and no structure is preserved). However, the relative padding overhead decreases as we increase the tensor size, so we expect the cyclic layout to at least maintain good weak scalability.

4.2 Overdecomposition

Our cyclic decomposition allows us to preserve symmetric fill structure so long as we satisfy the requirement that all symmetric tensor indices are decomposed with the same cyclic phase. This requirement presents a significant challenge to the naive method of assigning a single block to each processor, since for a n-dimensional symmetric index group, it would require the number of processors to have an integer nth root. This restriction could be overcome by using a subset of the processors to do the computation, but this would sacrifice efficiency and generality. Our solution is to add a layer of abstraction between the decomposition and the machine by assigning multiple sub-tensors to each processor.

Cyclops Tensor Framework has the capability to overdecompose the tensor into a number of blocks that is a multiple of the number of processors. We can represent a d-dimensional torus processor grid as a d-tuple $(p_1, \dots p_d)$, where $p_d \in \{1, 2, \dots\}$. In CTF, each n-dimensional tensor has n mappings $(m_1 \dots m_n)$ for each index, where each mapping consists of a list of processor grid dimension p, and an integer overdecomposition factor v, so that $m_i = (v, q)$ where $v \in \{1, 2, \dots\}$, $q \in \{0, 1, \dots d\}$. Defining $p_0 = 1$, the cyclic phase of the tensor sub-block along dimension i is then given by $v \cdot p_q$.

CTF attempts to use the least amount of overdecomposition, since larger blocks typically achieve higher matrix multiplication efficiency. The overdecomposition factor is then set to be equal to the least common multiple (lcm) of the physical dimensions to which indices in the symmetric group are mapped. For instance if a tensor has a 3-index symmetry with the first two indices mapped along processor grid dimensions p_1 and p_2, we will have the three indices mapped as

$$m_1 = (\text{lcm}(p_1, p_2)/p_1, 1)$$
$$m_2 = (\text{lcm}(p_1, p_2)/p_2, 2)$$
$$m_3 = (\text{lcm}(p_1, p_2), 0).$$

We note that the overall phase of these mappings is the same, so the symmetric fill structure of the index group is preserved within each sub-tensor block. Further, we see that overdecomposition allows a tensor to be distributed among any number of processors (in this case 6), whereas a naive decomposition would have required a processor count of $\{1, 8, 27, \dots\}$, and would have very limited capability for topology-aware mapping.

We do not use a dynamically scheduled overdecomposition approach such as that of the Charm++ runtime system [24]. Instead, our overdecomposition is set so that the dimensions of the cyclic decomposition are a multiple of the physical torus dimensions (by the factors v as above) and generate a regular mapping. For dense tensors, our approach maintains perfect load-balance and achieves high communication and task granularity by managing each overdecomposed sub-grid of blocks explicitly within each processor. However, we are exploring the utility of dynamic scheduling in allowing CTF to efficiently support sparse tensors.

4.3 Redistribution of data

As we have seen with overdecomposition, the symmetries of a tensor place requirements on the cyclic decomposition of the tensor. Further parallel decomposition requirements arise when the tensor participates in a summation or contraction with other tensors, in order to properly match up the index mappings among the tensors. Therefore, it is often necessary to change the mapping of the tensor data between each contraction. In general, we want to efficiently transform the data between any two given index to processor grid mappings.

Further, redistribution of data is necessary if the framework user wants to read or write parts of the tensor. We enable a user to read/write to any set of global indices from any processor in a bulk synchronous sparse read/write call. Additionally, CTF allows the user to extract any sub-block from a tensor and define it as a new CTF tensor. CTF has three redistribution kernels of varying generality and efficiency, all of which are used in the execution of coupled-cluster calculations. The functionality of each kernel is summarized as follows (the kernels are listed in order of decreasing generality and increasing execution speed)

- sparse redistribution – write or read a sparse set of data to or from a dense mapped CTF tensor
- dense redistribution – shuffles the data of a dense tensor from one CTF mapping to any other CTF mapping
- block redistribution – shuffling the assignment of the tensor blocks to processors without changing the ordering of data within tensor blocks

Of these three kernels, the second is the most complex, the most optimized, and is used the most intensively during CCSD/CCSDT calculations. The third kernel serves as a faster version of dense redistribution. The sparse redistribution kernel currently serves for input and output of tensor data, but in the future could also be used internally to execute sparse tensor contractions. We detail the architecture of each kernel below.

4.3.1 Sparse redistribution

There are multiple reasons for CTF to support sparse data redistribution, including data input and output, verification of the dense redistribution kernel, and support for sparse tensor storage. Our algorithm for reading and writing sparse data operates on a set of key-value pairs to read/write. The key is the global data index in the tensor. For a tensor of dimension d and edge lengths $(l_1, l_2, \ldots, l_d)$, the value at the position described by the tuple $(i_1, i_2, \ldots, i_3),\ 0 \leq i_k < l_k\ :\ \forall k$ is given by,

$$i_{global} = \sum_{k=1}^{d} \left(i_k \prod_{m=1}^{k-1} l_m \right).$$

The sparse algorithm is as follows:

1. sort the keys by global index
2. place the local key/value pairs into bins based on the indices' block in the final distribution
3. collect the bins for each block into bins for each destination processor
4. exchange keys among processors via all-to-all communication
5. combine the received keys for each block and sort them by key (the data is in global order within a block, but not across multiple blocks)

6. iterate over the local data of the target tensor, computing the key of each piece of data and performing a read or write if the next sparse key is the same

7. if the operation is a read, send the requested values back to the processors which requested them

This sparse read/write algorithm can be employed by a kernel which goes between two mappings of a tensor by iterating over local data and turning it into sparse format (key-value pairs), then calling a sparse write on a zero tensor in the new mapping. Further, by extracting only the values present in the sparse block of a tensor and changing the offsets of the keys, we allow general sub-block to sub-block redistribution.

This redistribution kernel is very general and fairly straightforward to implement and thread, however, it clearly does redundant work and communication for a redistribution between two mappings due to the formation, sorting, and communication of the keys. If the amount of data stored locally on each processor is n, this algorithm requires $O(n \log n)$ local work and memory traffic for the sort. The next kernel we detail is specialized to perform this task in a more efficient manner.

4.3.2 Dense redistribution

Between contractions, it is often the case that a CTF tensor must migrate from one mapping to another. These mappings can be defined on processor grids of different dimension and can have different overall cyclic phases along each dimension. This change implies that the padding of the data also changes, and it becomes necessary to move only the non-padded data. So, we must iterate over the local data, ignoring the padding, and send each piece of data to its new destination processor, then recover the data on the destination processor and write it back to the new local blocks in the proper order.

Our dense redistribution kernel utilizes the global ordering of the tensor to avoid forming or communicating keys with the index of data values. The kernel places the values into the send buffers in global order, performs the all-to-all then retrieves them from the received buffers using the knowledge that the data is in global order and thereby computing which processor sent each value. The same code with swapped parameters is used to compute the destination processor from the sender side, as well as to compute the sender processor from the destination side. The kernel is threaded by partitioning the tensor data among threads according to global order. We ensure a load-balanced decomposition of work between threads, by partitioning the global tensor in equal chunks according to its symmetric-packed layout, and having each thread work on the local part of this global partition, which is now balanced because the layout is cyclic. Another important optimization to the kernel, which significantly lowered the integer-arithmetic cost, was precomputing the destination blocks along each dimension of the tensor. For instance, to redistribute an n-by-n matrix from a p_r-by-p_c grid to another distribution, we precompute two destination vectors v and w of size n/p_r and n/p_c, respectively, on each processor, which allow us to quickly determine the destination of local matrix element a_{ij} via look ups to v_i and w_j.

The code necessary to implement this kernel is complex because it requires iterating over local tensor blocks data in global order, which requires striding over blocks, making the index arithmetic complex and the access pattern non-local. However, overall the algorithm performs much less integer arithmetic than the sparse redistribution kernel, performing $O(n)$ work and requiring $O(n)$ memory reads (although with potentially with less locality than sorting), if the local data size is n. Further, the all-to-all communication required for this kernel does not need to communicate keys along with the data like in the sparse redistribution kernel. In practice, we found the execution time to be roughly 10X faster than the sparse kernel. Nevertheless, these dense redistributions consume a noticeable fraction of the execution time of CTF during most CC calculations we have tested.

4.3.3 Block redistribution

Often, it is necessary to perform a redistribution that interchanges the mapping assignments of tensor dimensions. For instance, if we would like to symmetrize or desymmetrize a tensor, we need to add the tensor to itself with two indices transposed. In a distributed memory layout, this requires a distribution if either of the two indices are mapped onto a processor grid dimension. If the tensor was in a mapping where the two indices had the same cyclic phase (it had to if the indices were symmetric), it suffices to redistribute the blocks of the tensor. With this use-case in mind, we wrote a specialized kernel that goes from any pair of mappings with the same overall cyclic phases and therefore the same blocks.

We implemented this type of block redistribution by placing asynchronous receive calls for each block on each processor, sending all the blocks from each processor, and waiting for all the exchanges to finish. Since blocks are stored contiguously they are already serialized, so this kernel requires no local computational work and has a low memory-bandwidth cost. However, this kernel still incurs the network communication cost of an all-to-all data exchange. This block redistribution kernel is used whenever possible in place of the dense redistribution kernel.

5 Algorithms for tensor contraction

In the previous section we've established the mechanics of how CTF manages tensor data redistribution among mappings. This section will focus on detailing the algorithms CTF uses to select mappings for contractions and to perform the tensor contraction computation. We start by discussing what happens to tensor symmetries during a contraction and how symmetry may be exploited. Then we detail how the intra-node contraction is performed and how the inter-node communication is staged. Lastly, we detail how full contraction mappings are automatically selected based on performance models of redistribution and distributed contraction.

5.1 Tensor folding

Any tensor contraction of $\mathbf{A} \in \mathbb{R}^{I_{a_1} \times \ldots I_{a_l}}$ and $\mathbf{B} \in \mathbb{R}^{I_{b_1} \times \ldots I_{b_m}}$ into $\mathbf{C} \in \mathbb{R}^{I_{c_1} \times \ldots I_{c_n}}$ is an assignment of index labels to each dimension of the three tensors, such that every label is assigned to two dimensions in different tensors. This labeling is a tripartite graph $G = (V, E)$, with partitions $V = V_A \cup V_B \cup V_C$, where each vertex is a tensor dimension corresponding to tensor indices i_A, i_B, and i_C. Each vertex in V must be adjacent to a single edge in E, which corresponds to its matching index. The two tensor edge lengths of the matched index must be the same. We assign $t = |E| = |V|/2 = (l + m + n)/2$ labels to the set of edges E, i_E, and define three projection and permutation mappings $p_A \in i_E \rightarrow i_A$, $p_B \in i_E \rightarrow i_B$, and $p_C \in i_E \rightarrow i_C$. These projection mappings (indices) now completely define the element-wise contraction over the index space $\{I_1, \ldots I_t\}$,

$$\forall i_E \in \{1, \ldots I_1\} \times \ldots \{1, \ldots I_t\} \ : \ c_{p_C(i_E)} = c_{p_C(i_E)} + a_{p_A(i_E)} \cdot b_{p_B(i_E)}.$$

For example, matrix multiplication ($\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$) may be defined by mappings $p_A = (i, j, k) \rightarrow (i, k)$, $p_B = (i, j, k) \rightarrow (k, j)$ and $p_C = (i, j, k) \rightarrow (i, j)$. These projections are provided to the framework via the domain specific language in Section 3.1. The code is in fact somewhat more general than the definition of contractions we analyze here, as it allows the same index to appear in all three tensors (weigh operation) as well as only one of three tensors (reduction).

Tensor contractions reduce to matrix multiplication via index folding. Index folding corresponds to transforming sets of indices into larger compound indices, and may necessitate transposition of indices. We

define a folding of index set $s = \{i_1, \dots i_n\}$ into a single compound index q as a one-to-one map

$$f = \{1, \dots I_1\} \times \dots \{1, \dots I_n\} \to \{1, \dots \prod_{i=1}^{n} I_i\},$$

for instance with $s = \{i, j, k\}$, we have $q = f(s) = i + I_1 \cdot j + I_1 \cdot I_2 \cdot k$. Given a two-way partition function $(\pi_1(s), \pi_2(s))$, and two index set foldings f and g, we define a matrix folding $m(s) = (f(\pi_1(s)), g(\pi_2(s)))$ as a folding of a tensor's indices into two disjoint compound indices. Therefore, $\bar{\mathbf{A}} = m(\mathbf{A})$ with elements $\bar{a}_{m(s)} = a_s$.

Any contraction can be folded into matrix multiplication in the following manner, by defining mappings f_{AB}, f_{BC}, and f_{AC} and matrix foldings

$$\begin{aligned} m_A(i_A) &= (f_{AC}(i_{AC}), f_{AB}(i_{AB})), \\ m_B(i_B) &= (f_{AB}(i_{AB}), f_{BC}(i_{BC})), \\ m_C(i_C) &= (f_{AC}(i_{AC}), f_{BC}(i_{BC})), \end{aligned}$$

where $i_X = p_X(i_E)$ for all $X \in \{A, B, C\}$ and $i_{XY} = p_X(i_E) \cap p_Y(i_E)$ for all $X, Y \in \{A, B, C\}$. Now the contraction may be computed as a matrix multiplication of $\bar{\mathbf{A}} = m_A(\mathbf{A})$ with $\bar{\mathbf{B}} = m_B(\mathbf{B})$ into $\bar{\mathbf{C}} = m_C(\mathbf{C})$

$$\bar{c}_{ij} = \bar{c}_{ij} + \sum_k \bar{a}_{ik} \cdot \bar{b}_{kj}$$

Tensors can also have symmetry, we denote antisymmetric (skew-symmetric) index groups as

$$t_{[i_1, \dots i_j, \dots i_k, \dots i_n]} = -t_{[i_1, \dots i_k, \dots i_j, \dots i_n]}$$

for any $j, k \in [1, n]$. For the purpose of this analysis, we will only treat antisymmetric tensors, for symmetric tensors the non-zero diagonals require more special consideration. Using the notation $I^{\otimes n} = I \underbrace{\times \dots \times}_{(n-1)\text{-times}} I$, we denote a packed (folded) antisymmetric compound index as an onto map from a packed set of indices to a interval of size binomial in the tensor edge length

$$\hat{f}(i_1, \dots i_n) = \{1, \dots I\}^{\otimes n} \to \{1, \dots \binom{I}{n}\}$$

so given a simple contraction of antisymmetric tensors, such as,

$$c_{[i_1 \dots i_{k-s}], [i_{k-s+1} \dots i_m]} = \sum_{j_1 \dots j_s} a_{[i_1 \dots i_{k-s}], [j_1 \dots j_s]} \cdot b_{[j_1 \dots j_s], [i_{k-s+1} \dots i_m]},$$

we can compute it in packed antisymmetric layout via matrix foldings

$$\begin{aligned} m_C(\{i_1, \dots i_m\}) &= (\hat{f}(i_1, \dots i_{k-s}), \hat{g}(i_{k-s+1}, \dots i_m)), \\ m_A(\{i_1, \dots i_{k-s}, j_1, \dots j_s\}) &= (\hat{f}(i_1, \dots i_{k-s}), \hat{h}(j_1, \dots j_s)), \\ m_B(\{j_1, \dots j_s, i_{k-s+1}, i_m\}) &= (\hat{h}(j_1, \dots j_s), \hat{g}(i_{k-s+1}, \dots i_m)). \end{aligned}$$

So that, for $\bar{\mathbf{A}} = m_A(\mathbf{A})$, $\bar{\mathbf{B}} = m_B(\mathbf{B})$, and $\bar{\mathbf{C}} = m_C(\mathbf{C})$,

$$\begin{aligned} \forall \{i_1 \dots i_{k-s}\} \in \{1, \dots I\}^{\otimes k-s}, \{i_{k-s+1} \dots i_m\} \in \{1, \dots I\}^{\otimes m-k+s}, \\ \{j_1 \dots j_s\} \in \{1, \dots I\}^{\otimes s} \ : \\ \bar{c}_{m_C(\{i_1, \dots i_m\})} = \bar{c}_{m_C(\{i_1, \dots i_m\})} + s! \cdot \bar{a}_{m_A(\{i_1, \dots i_{k-s}, j_1, \dots j_s\})} \cdot \bar{b}_{m_B(\{j_1, \dots j_s, i_{k-s+1}, i_m\})} \end{aligned}$$

$$\forall i \in \left\{1, \dots \binom{I}{k-s}\right\}, j \in \left\{1, \dots \binom{I}{m-k+s}\right\} \ : \ \bar{c}_{ij} = s! \sum_{k=1}^{\binom{I}{s}} \bar{a}_{ik} \cdot \bar{b}_{kj}.$$

The above contraction is an example where all symmetries are *preserved*. Any preserved symmetries must be symmetries of each tensor within the whole contraction term. We can consider a tensor $\mathbf{Z}$ corresponding to the uncontracted set of scalar multiplications whose entries are pairs associated with scalar multiplications,

$$z_{i_1 \ldots i_m j_1 \ldots j_s} = (a_{i_1 \ldots i_{k-s} j_1 \ldots j_s}, b_{j_1 \ldots j_s i_{k-s+1} \ldots i_m}).$$

If $\mathbf{Z}$ is symmetric in a pair of indices, we call this symmetry preserved. *Broken* symmetries are symmetries which exist in one of $\mathbf{A}$, $\mathbf{B}$, or $\mathbf{C}$, but not in $\mathbf{Z}$. For example, we can consider the contraction

$$c_{[ij]kl} = \sum_{pq} a_{[ij][pq]} \cdot b_{pk[ql]}$$

which corresponds to contraction graph elements $z_{[ij]klpq}$. The symmetry $[ij]$ is preserved but the symmetries $[pq]$ and $[ql]$ are broken. While for each preserved symmetry in a contraction we achieve a reduction in floating point operations, for broken symmetries we currently only exploit preservation of storage. The broken symmetries can be unpacked and the contraction computed as

$$\bar{c}_{\hat{f}(i,j),k,l} = \sum_{p,q} \bar{a}_{\hat{f}(i,j),p,q} \cdot b_{p,k,q,l}$$

or the broken symmetries can remain folded, in which case multiple permutations are required,

$$\begin{aligned} \bar{c}_{\hat{f}(i,j),k,l} &= \sum_{p<q} \left[\bar{a}_{\hat{f}(i,j),\hat{g}(p,q)} \cdot \bar{b}_{p,k,\hat{h}(q,l)} - \bar{a}_{\hat{f}(i,j),\hat{g}(p,q)} \cdot \bar{b}_{p,k,\hat{h}(l,q)} \right] \\ &- \sum_{q<p} \left[\bar{a}_{\hat{f}(i,j),\hat{g}(q,p)} \cdot \bar{b}_{p,k,\hat{h}(q,l)} - \bar{a}_{\hat{f}(i,j),\hat{g}(q,p)} \cdot \bar{b}_{p,k,\hat{h}(l,q)} \right] \end{aligned}$$

Our framework makes dynamic decisions to unpack broken symmetries in tensors or to perform the packed contraction permutations, based on the amount of memory available. Unpacking each pair of indices is done via a tensor summation which requires either a local transpose of data of a global transpose with interprocessor communication, which can usually be done via a block-wise redistribution which was detailed in Section 4.3.3. It may be possible to efficiently reduce or avoid the desymmetrization and symmetrization calls necessary to deal with broken symmetries and we are currently exploring possible new algorithms that can achieve this. However, currently unpacking leads to highest efficiency due to the ability to fold the unpacked indices into the matrix multiplication call.

5.2 On-node contraction of tensors

To prepare a folded form for the on-node contractions, we perform non-symmetric transposes of each tensor block. In particular, we want to separate out all indices which take part in a broken symmetric group within the contraction and linearize the rest of the indices. If a symmetry group is not broken, we can simply fold the symmetric indices into one bigger dimension linearizing the packed layout. We perform an ordering transposition on the local tensor data to make dimensions which can be folded, as fastest increasing and the broken symmetric dimensions as slowest increasing within the tensors. To do a sequential contraction, we can then iterate over the broken symmetric indices (or unpack the symmetry) and call matrix multiplication over the linearized indices. For instance, the contraction from the start of this section,

$$c_{[ij]kl} = \sum_{pq} a_{[ij][pq]} \cdot b_{pk[ql]}$$

would be done as a single matrix multiplication for each block, if the $[pq]$ and $[ql]$ symmetries are unpacked. However, if all the broken symmetries are kept folded, the non-symmetric transpose would make fastest increasing the folded index corresponding to $\hat{f}(i,j)$, as well as the k index, so that the sequential kernel could iterate over p, q, l and call a vector outer-product operation over $\hat{f}(i,j), k$ for each p, q, l. The underlying call is generally a matrix multiplication, though in this case it reduces to an outer-product of two vectors, and in other cases could reduce to a vector or scalar operation. No matter what the mapping of the tensor is, the non-symmetric transpose required to fold the tensor blocks can be done locally within each block and requires no network communication. Therefore, while the non-symmetric permutations present an overhead to the sequential performance, their cost decreased comparatively as we perform strong or weak scaling.

5.3 Distributed contraction of tensors

In CTF, tensors are distributed in cyclic layouts over a virtual torus topology. The distributed algorithm for tensor contractions used by CTF is a combination of replication, as done in 3D and 2.5D algorithms [50] and a nested SUMMA [1, 53] algorithm. If the dimensions of two tensors with the same contraction index are mapped onto different torus dimensions, a SUMMA algorithm is done on the plane defined by the two torus dimensions. For each pair of indices mapped in this way, a nested level of SUMMA is done. The indices must be mapped with the same cyclic phase so that the blocks match, adding an extra mapping restriction which necessitates overdecomposition.

We employ three versions of SUMMA, each one communicates a different pair of the the three tensors and keeps one tensor in place. The operand tensors are broadcast before each subcontraction, while the output tensor is reduced after each subcontraction. So, in one version of SUMMA (the outer-product version) two broadcasts are done (one of each operand tensor block) and in the other two of the three versions of SUMMA one reduction is done (instead of one of the broadcasts) to accumulate partial sums to blocks of the output. If the processor dimensions onto which the pair of indices corresponding to a SUMMA call are mapped are of different lengths, the indices must be overdecomposed to the same cyclic phase, and our SUMMA implementation does as many communication steps as the cyclic phase of the indices.

Blocking the computation on 3D grids (replicating over one dimension) allows for algorithms which achieve better communication efficiency in the strong scaling limit. Strong scaling efficiency is necessary for higher order CC methods which require many different small contractions to be computed rapidly. When a small contraction is computed on a large number of processors there is typically much more memory is available than the amount necessary to store the tensor operands and output. The additional memory can be exploited via replication of tensor data and reduction of overall communication volume. We always replicate the smallest one of the three tensors involved in the contraction to minimize the amount of memory and communication overhead of replication.

5.4 Topology-aware network mapping

Each contraction can place unique restrictions on the mapping of the tensors. In particular, our decomposition needs all symmetric tensor dimensions to be mapped with the same cyclic phase. Further, we must satisfy special considerations for each contraction, that can be defined in terms of indices (we will call them paired tensor dimensions) which are shared by a pair of tensors in the contraction. These considerations are

1. dimensions which are paired must be mapped with the same phase
2. for the paired tensor dimensions which are mapped to different dimensions of the processor grid (are mismatched)
   (a) the mappings of two pairs of mismatched dimensions cannot share dimensions of the processor grid

(b) the subspace formed by the mappings of the mismatched paired dimensions must span all input data

When the physical network topology is a d-dimensional toroidal partition specified by the d-tuple $(p_1, \ldots p_d)$. CTF considers all foldings of physical topology that preserve the global ordering of the processors (e.g. p_2 may be folded with p_3 but not only with p_4). If the physical network topology is not specified as a torus, CTF factorizes the number of processes up to 8 factors and treats the physical topology as an 8-dimensional processor grid and attempts to map to all lower-dimensional foldings of it. When there are more tensor indices than processor grid dimensions, additional grid dimensions of length 1 are added.

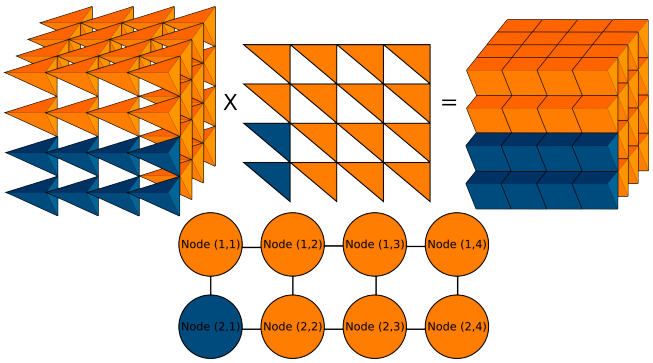


Figure 2: Overdecomposition as used in CTF to perform contractions. This diagram demonstrates a mapping for a contraction of the form $c_{[kl]i} = \sum_j a_{[jkl]} \cdot b_{[ij]}$. In this case, we have a 4-by-2 processor grid, and a 4-by-4-by-4 set of blocks.

For any given topology, CTF attempts to map all three possible pairs of tensors so that their indices are mismatched on the processor grid and they are communicated in the contraction kernel. The mapping of two tensors automatically defines the mapping of the third, though the overdecomposition factor must be adjusted to satisfy all symmetries and matching indices among tensors. CTF also considers partial replication of tensors among the processor grid dimensions. Figure 2 shows an example of an overdecomposed mapping with a properly matched cyclic phase along all tensor dimensions.

The search through mappings is done entirely in parallel among processors, then the best mapping is selected across all processors. The mapping logic is done without reading or moving any of the tensor data and is generally composed of integer logic that executes in an insignificant amount of time with respect to the contraction. We construct a 'ghost' mapping for each valid topology and each ordering of tensors. The distributed contraction algorithm is constructed on each ghost mapping, and its communication and memory overheads are evaluated. If the ghost mapping is suboptimal it is thrown out without ever dictating data movement. Once a mapping is decided upon, the tensors are redistributed. Currently, our performance cost models predominantly consider communication, with controls on the amount of overdecomposition and memory usage. We are seeking to extend this cost model and to add infrastructure for training the parameters of our performance models.

6 Application performance

The implementation of CTF employs MPI [17] for interprocessor communication, BLAS for matrix multiplication and summation, as well as OpenMP for threading. All other functionalities in CTF were developed from scratch and have no other dependencies. We used vendor provided optimized on-node parallel BLAS implementations (IBM ESSL and Cray LibSci) on all architectures for benchmarking. While the interface

is C++, much of the internals of CTF are in C-style with C++ library support. Outside of special network topology considerations on BlueGene/Q, CTF does not employ any optimizations which are specific to an architecture or an instruction set. Performance profiling is done by hand and with TAU [48].

6.1 Architectures

Cyclops Tensor Framework targets massively parallel architectures and is designed to take advantage of network topologies and communication infrastructure that scale to millions of nodes. Parallel scalability on commodity clusters should benefit significantly from the load balanced characteristics of the workload, while high-end supercomputers will additionally benefit from reduced inter-processor communication which typically becomes a bottleneck only at very high degrees of parallelism. We collected performance results on two state-of-the-art supercomputer architectures, IBM Blue Gene/Q and Cray XC30. We also tested sequential and multi-threaded performance on a Xeon desktop.

The sequential and non-parallel multi-threaded performance of CTF is compared to NWChem and MRCC. The platform is a commodity dual-socket quad-core Xeon E5620 system. On this machine, we used the sequential and threaded routines of the Intel Math Kernel Library. This platform, as well as the problem sizes tested reflect a typical situation for workloads on a workstation or small cluster, which is where the sequential performance of these codes is most important. Three problem sizes are timed, spanning a variety of ratios of the number of virtual orbitals to occupied orbitals.

The second experimental platform is 'Edison', a Cray XC30 supercomputer with two 12-core Intel "Ivy Bridge" processors at 2.4GHz per node (19.2 Gflops per core and 460.8 Gflops per node). Edison has a Cray Aries high-speed interconnect with Dragonfly topology ($0.25\mu s$ to $3.7\mu s$ MPI latency, 8 GB/sec MPI bandwidth). Each node has 64 GB of DDR3 1600 MHz memory (four 8 GB DIMMs per socket) and two shared 30 MB L3 caches (one per Ivy Bridge). Each core has its own L1 and L2 caches, of size 64 KB and 256 KB.

The final platform we consider is the IBM Blue Gene/Q (BG/Q) architecture. We use the installations at Argonne and Lawrence Livermore National Laboratories. On both installations, IBM ESSL was used for BLAS routines. BG/Q has a number of novel features, including a 5D torus interconnect and 16-core SMP processor with 4-way hardware multi-threading, transactional memory and L2-mediated atomic operations [19], all of which serve to enable high performance of the widely portable MPI/OpenMP programming model. The BG/Q cores run at 1.6 GHz and the QPX vector unit supports 4-way fused multiply-add for a single-node theoretical peak of 204.8 GF/s. The BG/Q torus interconnect provides 2 GB/s of theoretical peak bandwidth per link in each direction, with simultaneous communication along all 10 links achieving 35.4 GB/s for 1 MB messages [10].

6.2 Results

We present the performance of a CCSD and CCSDT implementation contained within Aquarius and executed using Cyclops Tensor Framework. For each contraction, written in one line of Aquarius code, CTF finds a topology-aware mapping of the tensors to the computer network and performs the necessary set of contractions on the packed structured tensors.

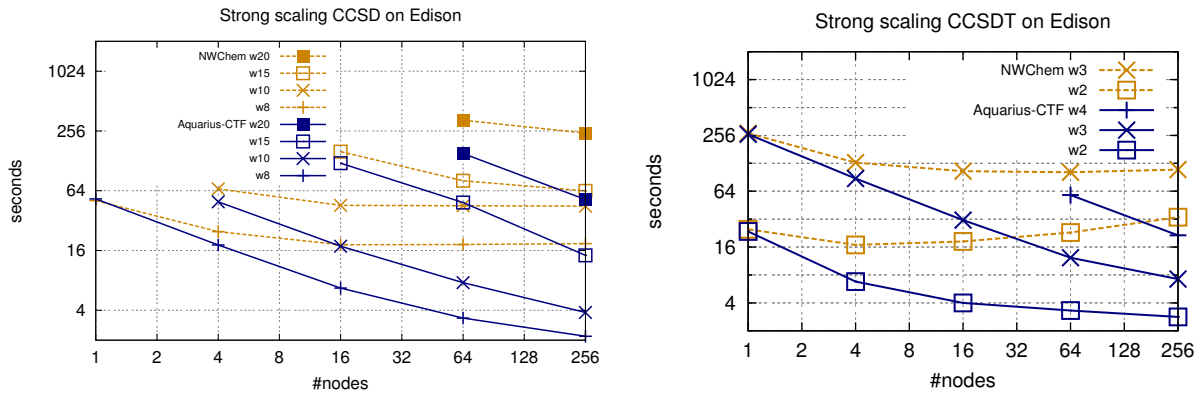
6.2.1 Sequential CCSD performance

The results of the sequential and multi-threaded comparison are summarized in Table 1. The time per CCSD iteration is lowest for NWChem in all cases, and similarly highest for MRCC. The excessive iteration times for MRCC when the $\frac{n_v}{n_o}$ ratio becomes small reflect the fact that MRCC is largely memory-bound, as contractions are performed only with matrix-vector products. The multi-threaded speedup of CTF is

significantly better than NWChem, most likely due to the lack of multi-threading of tensor transposition and other non-contraction operations in NWChem.

Table 1: Sequential and non-parallel multi-threaded performance comparison of CTF, NWChem, and MRCC. Entries are average time for one CCSD iteration, for the given number of virtual (n_v) and occupied (n_o) orbitals.

		$n_v = 110$ $n_o = 5$	$n_v = 94$ $n_o = 11$	$n_v = 71$ $n_o = 23$
NWChem	1 thread	6.80 sec	16.8 sec	49.1 sec
CTF	1 thread	23.6 sec	32.5 sec	59.8 sec
MRCC	1 thread	31.0 sec	66.2 sec	224. sec
NWChem	8 threads	5.21 sec	8.60 sec	18.1 sec
CTF	8 threads	9.12 sec	9.37 sec	18.5 sec
MRCC	8 threads	67.3 sec	64.3 sec	86.6 sec

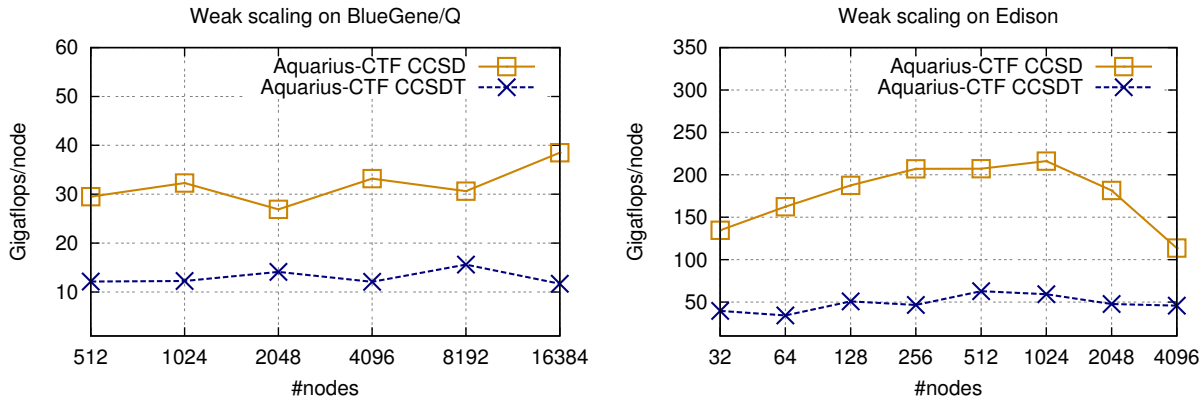


(a) CCSD (b) CCSDT

Figure 3: CCSD and CCSDT strong scaling of water clusters with cc-pVDZ basis set on Edison (Cray XC30).

6.2.2 Strong scaling

On the Cray XC30 machine, we compared the performance of our CCSD implementation with that of NWChem. We benchmarked the two codes for a series of water systems – wn, where n is the number of water monomers in the system. Water clusters give a flexible benchmark while also representing an interesting chemical system for their role in describing bulk water properties. In Figure 3(a), we compare the scaling of CCSD using CTF with the scaling of NWChem. Our version of NWChem 6.3 used MPI-3 and was executed using one MPI process per core. The tile size was 40 for CCSD and 20 for CCSDT. The CCSD performance achieved by CTF becomes significantly higher than NWChem when the number of processors used for the calculation is increased. CTF can both solve problems in shorter time than NWChem (strong scale) and solve larger problems faster (weak scale). Figure 3(b) gives the CCSDT strong scaling comparison. CCSDT for system sizes above 3 molecules did not work successfully with our NWChem build, but a large performance advantage for CTF is evident for the smaller systems.



(a) IBM BlueGene/Q (Mira)

(b) Cray XC30 (Edison)

Figure 4: CCSD and CCSDT weak scaling on water clusters with cc-pVDZ basis set.

Table 2: A performance breakdown of important kernels for a CCSD iteration done by CTF on a system with $n_o = 125$ occupied orbitals and $n_v = 600$ virtual orbitals on 256 nodes of Edison (XC30) and 1024 nodes of Mira (BG/Q). Complexity is in terms of p processors and M memory per processor.

kernel	BG/Q	XC30	complexity	architectural bounds
matrix mult.	49%	35%	$O(n_v^4 n_o^2/p)$	flops/mem bandwidth
broadcasts	24%	37%	$O(n_v^4 n_o^2/p\sqrt{M})$	multicast bandwidth
prefix sum	10%	4%	$O(p)$	allreduce bandwidth
data packing	4%	3%	$O(n_v^2 n_o^2/p)$	integer ops
all-to-all-v	3%	10%	$O(n_v^2 n_o^2/p)$	bisection bandwidth
tensor folding	6%	5%	$O(n_v^2 n_o^2/p)$	memory bandwidth
other	4%	6%		

6.2.3 Weak scaling

The parallel weak scaling efficiency of our CCSD and CCSDT implementation on Blue Gene/Q is displayed in Figure 4(a) and on Cray XC30 is displayed in Figure 4(b). This weak scaling data was collected by doing the largest CCSD and CCSDT run that would fit in memory on each node count and normalizing the efficiency by the number of floating point operations performed by CTF. Going from 512 to 16384 nodes (256K cores), the efficiency actually often increases, since larger problems can be solved, which increases the ratio of computation over communication. For CCSD, CTF maintains 30 GF/node on BG/Q, which is about one sixth of peak. On Edison, CCSD surpasses 200 GF/node, which is over 50% of peak. The application was run with 4 MPI processes per node and 16 threads per process on BG/Q and with 4 MPI processes per node and 6 threads per process on Edison.

Table 2 lists profiling data for a CCSD iteration of CTF on 1024 nodes (16K cores) of BG/Q and 256 nodes (6K cores) of Edison on 25 water molecules with a basis set size of 600 virtual orbitals. This problem took 228 seconds on 1024 nodes of BG/Q and 167 seconds on 256 nodes of Edison. The table reports the percentage of execution time of a CCSD iteration spent in the main kernels in CTF. The table also lists the architectural bounds for each kernel, demonstrating the components of the hardware being utilized by each computation. The time inside matrix multiplication reflects the time spent working on tensor contractions sequentially on each processor, while broadcasts represent the time spent communicating data for replication and the nested SUMMA algorithm. The prefix sum, data packing, and all-to-all-v operations are all

Table 3: A performance breakdown of important kernels for a CCSDT iteration done by CTF on a system with 8 water molecules ($n_o = 40$ occupied orbitals) and a cc-pVDZ basis set ($n_v = 192$ virtual orbitals) on 256 nodes of Edison (XC30) and 1024 nodes of Mira (BG/Q). Complexity is in terms of p processors and M memory per processor.

kernel	BG/Q	XC30	complexity	architectural bounds
matrix mult.	20%	16%	$O(n_v^5 n_o^3/p)$	flops/mem bandwidth
broadcasts	16%	11%	$O(n_v^5 n_o^3/p\sqrt{M})$	multicast bandwidth
prefix sum	5%	3%	$O(p)$	allreduce bandwidth
data packing	9%	9%	$O(n_v^3 n_o^3/p)$	integer ops
all-to-all-v	19%	20%	$O(n_v^3 n_o^3/p)$	bisection bandwidth
tensor folding	26%	39%	$O(n_v^3 n_o^3/p)$	memory bandwidth
other	5%	2%		

part of tensor redistribution, which is a noticeable overall overhead. Tensor folding corresponds to local transposition of each tensor block.

Table 2 yields the somewhat misleading initial observation that nearly half the execution is spent in matrix multiplication on BG/Q, 14% more than than the fraction spent in matrix multiplication on Edison. In fact, we observe this difference for two reasons, because the BG/Q run uses four times the number of processors causing there to be more redundant computation on padding in the decomposition, and because the matrix multiplication library being employed on BG/Q achieves a significantly lower fraction of peak than on Edison for the invoked problem sizes. Overall, the computational efficiency of our CCSD implementation is more favorable on Edison than on BG/Q.

For CCSDT, Aquarius and CTF maintain a lower percentage of peak than for CCSD, but achieves good parallel scalability. The lower fraction of peak is expected due to the CCSDT increased relative cost and frequency of transpositions and summations with respect to CCSD. In particular CCSD performs $O(n^{3/2})$ computation with $O(n)$ data while CCSDT performs $O(n^{4/3})$ computation with $O(n)$ data. The third-order excitation tensor $\mathbf{T_3}$ present only in CCSDT has 6 dimensions and its manipulation (packing, unpacking, transposing, and folding) becomes an overhead. We observe this trend in the CCSDT performance profile on Table 3, where tensor transposition and redistribution take up a larger fraction of the total time than in the CCSD computation considered in Table 2. The 8-water CCSDT problem in Table 3 took 15 minutes on 1024 nodes of BG/Q and 21 minutes on 256 nodes of Edison. The strong scalability achievable for this problem is significantly better on Edison, increasing the number of nodes by four, BG/Q performs such a CCSDT iteration (using 4096 nodes) in 9 minutes while Edison computes it (using 1024 nodes) in 6 minutes.

7 Future work

CTF provides an efficient distributed-memory approach to dense tensor contractions. The infrastructure presented in the paper may be extended conceptually and is being improved in practice by more robust performance modelling and mapping logic. Another promising direction is the addition of inter-contraction parallelism, allowing for smaller contractions to be scheduled concurrently. CTF already supports working on subsets of nodes and moving data between multiple instances of the framework, providing the necessary infrastructure for a higher level scheduling environment.

From a broader applications standpoint the addition of sparsity support to CTF is a particularly attractive future direction. Working support for distributed memory tensor objects would allow for the expression of a wide set of numerical schemes based on sparse matrices. Further tensors with banded sparsity structure can be decomposed cyclically so as to preserve band structure in the same way CTF preserves symmetry.

Tensors with arbitrary sparsity can also be decomposed cyclically, though the decomposition may need to perform load balancing in the mapping and execution logic.

Cyclops Tensor Framework will also be integrated with a higher-level tensor expression manipulation framework as well as CC code generation methods. We have shown a working implementation of CCSD and CCSDT on top of CTF, but aim to implement more complex methods. In particular, we are targeting the CCSDTQ method, which employs tensors of dimension up to 8 and gets the highest accuracy of any desirable CC method (excitations past quadruples have a negligible contribution).

Acknowledgments

ES and DM were supported by a Department of Energy Computational Science Graduate Fellowship, grant number DE-FG02-97ER25308. We acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016. This research used resources of the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, *A three-dimensional approach to parallel matrix multiplication*, IBM J. Res. Dev., 39 (1995), pp. 575–582.

[2] Alok Aggarwal, Ashok K. Chandra, and Marc Snir, *Communication complexity of PRAMs*, Theoretical Computer Science, 71 (1990), pp. 3 – 28.

[3] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz, *Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds*, in Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures, SPAA '12, New York, NY, USA, 2012, ACM, pp. 77–79.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in linear algebra*, SIAM J. Mat. Anal. Appl., 32 (2011).

[5] R. J. Bartlett and M. Musiał, *Coupled-cluster theory in quantum chemistry*, Reviews of Modern Physics, 79 (2007), pp. 291–352.

[6] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, *Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models*, Proceedings of the IEEE, 93 (2005), pp. 276 –292.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK user's guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[8] E. J. BYLASKA ET. AL., *NWChem, a computational chemistry package for parallel computers, version 6.1.1*, 2012.

[9] LYNN ELLIOT CANNON, *A cellular computer to implement the Kalman filter algorithm*, PhD thesis, Bozeman, MT, USA, 1969.

[10] DONG CHEN, NOEL A. EISLEY, PHILIP HEIDELBERGER, ROBERT M. SENGER, YUTAKA SUGAWARA, SAMEER KUMAR, VALENTINA SALAPURA, DAVID L. SATTERFIELD, BURKHARD STEINMACHER-BUROW, AND JEFFREY J. PARKER, *The IBM Blue Gene/Q interconnection network and message unit*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, NY, USA, 2011, ACM, pp. 26:1–26:10.

[11] T. D. CRAWFORD AND H. F. SCHAEFER III, *An introduction to coupled cluster theory for computational chemists*, Reviews in Computational Chemistry, 14 (2000), p. 33.

[12] ELIEZER DEKEL, DAVID NASSIMI, AND SARTAJ SAHNI, *Parallel matrix and graph algorithms*, SIAM Journal on Computing, 10 (1981), pp. 657–675.

[13] JAMES DEMMEL, DAVID ELIAHU, ARMANDO FOX, SHOAIB KAMIL, BENJAMIN LIPSHITZ, ODED SCHWARTZ, AND OMER SPILLINGER, *Communication-optimal parallel recursive rectangular matrix multiplication*, in IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2013.

[14] ERIK DEUMENS, VICTOR F. LOTRICH, AJITH PERERA, MARK J. PONTON, BEVERLY A. SANDERS, AND RODNEY J. BARTLETT, *Software design of ACES III with the super instruction architecture*, Wiley Interdisciplinary Reviews: Computational Molecular Science, 1 (2011), pp. 895–901.

[15] EVGENY EPIFANOVSKY, MICHAEL WORMIT, TOMASZ KU, ARIE LANDAU, DMITRY ZUEV, KIRILL KHISTYAEV, PRASHANT MANOHAR, ILYA KALIMAN, ANDREAS DREUW, AND ANNA I. KRYLOV, *New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations*, Journal of Computational Chemistry, (2013).

[16] XIAOYANG GAO, SRIRAM KRISHNAMOORTHY, SWARUP SAHOO, CHI-CHUNG LAM, GERALD BAUMGARTNER, J. RAMANUJAM, AND P. SADAYAPPAN, *Efficient search-space pruning for integrated fusion and tiling transformations*, in Languages and Compilers for Parallel Computing, vol. 4339 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 215–229.

[17] WILLIAM GROPP, EWING LUSK, AND ANTHONY SKJELLUM, *Using MPI: portable parallel programming with the message-passing interface*, MIT Press, Cambridge, MA, USA, 1994.

[18] MICHAEL HANRATH AND ANNA ENGELS-PUTZKA, *An efficient matrix-matrix multiplication based antisymmetric tensor contraction engine for general order coupled cluster*, The Journal of Chemical Physics, 133 (2010).

[19] R.A. HARING, M. OHMACHT, T.W. FOX, M.K. GSCHWIND, D.L. SATTERFIELD, K. SUGAVANAM, P.W. COTEUS, P. HEIDELBERGER, M.A. BLUMRICH, R.W. WISNIEWSKI, A. GARA, G.L.-T. CHIU, P.A. BOYLE, N.H. CHIST, AND CHANGHOAN KIM, *The IBM Blue Gene/Q compute chip*, Micro, IEEE, 32 (2012), pp. 48 –60.

[20] SO HIRATA, *Tensor Contraction Engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories*, The Journal of Physical Chemistry A, 107 (2003), pp. 9887–9897.

[21] DROR IRONY, SIVAN TOLEDO, AND ALEXANDER TISKIN, *Communication lower bounds for distributed-memory matrix multiplication*, Journal of Parallel and Distributed Computing, 64 (2004), pp. 1017 – 1026.

[22] HONG JIA-WEI AND H. T. KUNG, *I/O complexity: The red-blue pebble game*, in Proceedings of the thirteenth annual ACM symposium on Theory of computing, STOC '81, New York, NY, USA, 1981, ACM, pp. 326–333.

[23] S. LENNART JOHNSSON, *Minimizing the communication time for matrix multiplication on multiprocessors*, Parallel Comput., 19 (1993), pp. 1235–1257.

[24] LAXMIKANT V. KALE AND SANJEEV KRISHNAN, *CHARM++: a portable concurrent object oriented system based on C++*, in Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '93, New York, NY, USA, 1993, ACM, pp. 91–108.

[25] MIHÁLY KÁLLAY AND PÉTER R. SURJÁN, *Higher excitations in coupled-cluster theory*, The Journal of Chemical Physics, 115 (2001), p. 2945.

[26] DANIEL KATS AND FREDERICK R. MANBY, *Sparse tensor framework for implementation of general local correlation methods*, The Journal of Chemical Physics, 138 (2013), pp. –.

[27] P.J. KNOWLES AND N.C. HANDY, *A new determinant-based full configuration interaction method*, Chemical Physics Letters, 111 (1984), pp. 315 – 321.

[28] STANISLAW A. KUCHARSKI AND RODNEY J. BARTLETT, *Recursive intermediate factorization and complete computational linearization of the coupled-cluster single, double, triple, and quadruple excitation equations*, Theoretica Chimica Acta, 80 (1991), pp. 387–405.

[29] PAI-WEI LAI, KEVIN STOCK, SAMYAM RAJBHANDARI, SRIRAM KRISHNAMOORTHY, AND P. SADAYAPPAN, *A framework for load balancing of tensor contraction expressions via dynamic task partitioning*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, New York, NY, USA, 2013, ACM, pp. 13:1–13:10.

[30] YOON S. LEE, STANISLAW A. KUCHARSKI, AND RODNEY J. BARTLETT, *A coupled cluster approach with triple excitations*, Journal of Chemical Physics, 81 (1984), p. 5906.

[31] VICTOR LOTRICH, NORBERT FLOCKE, MARK PONTON, BEVERLY A. SANDERS, ERIK DEUMENS, RODNEY J. BARTLETT, AND AJITH PERERA, *An infrastructure for scalable and portable parallel programs for computational chemistry*, in Proceedings of the 23rd international conference on Supercomputing, ICS '09, New York, NY, USA, 2009, ACM, pp. 523–524.

[32] W. F. MCCOLL AND A. TISKIN, *Memory-efficient matrix multiplication in the BSP model*, Algorithmica, 24 (1999), pp. 287–297.

[33] HENDRIK J. MONKHORST, *Calculation of properties with the coupled-cluster method*, International Journal of Quantum Chemistry, 12 (1977), p. 421432.

[34] JAROSLAW NIEPLOCHA, ROBERT J. HARRISON, AND RICHARD J. LITTLEFIELD, *Global Arrays: A nonuniform memory access programming model for high-performance computers*, The Journal of Supercomputing, 10 (1996), pp. 169–189.

[35] JOZEF NOGA AND RODNEY J. BARTLETT, *The full CCSDT model for molecular electronic structure*, Journal of Chemical Physics, 86 (1987), p. 7041.

[36] JEPPE OLSEN, BJÖRN O. ROOS, POUL JØRGENSEN, AND HANS JØRGEN AA. JENSEN, *Determinant based configuration interaction algorithms for complete and restricted configuration interaction spaces*, The Journal of Chemical Physics, 89 (1988), pp. 2185–2192.

[37] JOHN A. PARKHILL AND MARTIN HEAD-GORDON, *A sparse framework for the derivation and implementation of fermion algebra*, Molecular Physics, 108 (2010), pp. 513–522.

[38] ROBERT G PARR AND YANG WEITAO, *Density-functional theory of atoms and molecules*, Oxford University Press ; Clarendon Press, New York; Oxford, 1989.

[39] J. A. POPLE AND R. K. NESBET, *Self-consistent orbitals for radicals*, Journal of Chemical Physics, 22 (1954), p. 571.

[40] JACK POULSON, BRYAN MAKER, JEFF R. HAMMOND, NICHOLS A. ROMERO, AND ROBERT VAN DE GEIJN, *Elemental: A new framework for distributed memory dense matrix computations*, ACM Transactions on Mathematical Software. in press.

[41] GEORGE D PURVIS AND RODNEY J BARTLETT, *A full coupledcluster singles and doubles model: The inclusion of disconnected triples*, The Journal of Chemical Physics, 76 (1982), pp. 1910–1918.

[42] SAMYAM RAJBHANDARI, AKSHAY NIKAM, PAI-WEI LAI, KEVIN STOCK, SRIRAM KRISHNAMOORTHY, AND P SADAYAPPAN, *Framework for distributed contractions of tensors with symmetry*, Preprint, Ohio State University, (2013).

[43] C. C. J. ROOTHAAN, *New developments in molecular orbital theory*, Reviews of Modern Physics, 23 (1951), pp. 69 – 89.

[44] MARTIN SCHATZ, JACK POULSON, AND ROBERT A VAN DE GEIJN, *Scalable universal matrix multiplication algorithms: 2D and 3D variations on a theme*, ACM Transactions on Mathematical Software, (2012).

[45] ANDREW C. SCHEINER, GUSTAVO E. SCUSERIA, JULIA E. RICE, TIMOTHY J. LEE, AND HENRY F. SCHAEFER, *Analytic evaluation of energy gradients for the single and double excitation coupled cluster (CCSD) wave function: Theory and application*, Journal of Chemical Physics, 87 (1987), p. 5361.

[46] G. E. SCUSERIA AND H. F. SCHAEFER III, *Is coupled cluster singles and doubles (CCSD) more computationally intensive than quadratic configuration interaction (QCISD)?*, Journal of Chemical Physics, 90 (1989), pp. 3700–3703.

[47] YIHAN SHAO, LASZLO FUSTI MOLNAR, YOUSUNG JUNG, JRG KUSSMANN, CHRISTIAN OCHSENFELD, SHAWN T. BROWN, ANDREW T. B. GILBERT, LYUDMILA V. SLIPCHENKO, SERGEY V. LEVCHENKO, DARRAGH P. ONEILL, ROBERT A. DISTASIO JR, ROHINI C. LOCHAN, TAO WANG, GREGORY J. O. BERAN, NICHOLAS A. BESLEY, JOHN M. HERBERT, CHING YEH LIN, TROY VAN VOORHIS, SIU HUNG CHIEN, ALEX SODT, RYAN P. STEELE, VITALY A. RASSOLOV, PAUL E. MASLEN, PRAKASHAN P. KORAMBATH, ROSS D. ADAMSON, BRIAN AUSTIN, JON BAKER, EDWARD F. C. BYRD, HOLGER DACHSEL, ROBERT J. DOERKSEN, ANDREAS DREUW, BARRY D. DUNIETZ, ANTHONY D. DUTOI, THOMAS R. FURLANI, STEVEN R. GWALTNEY, ANDREAS HEYDEN, SO HIRATA, CHAO-PING HSU, GARY KEDZIORA, RUSTAM Z. KHALLIULIN, PHIL KLUNZINGER, AARON M. LEE, MICHAEL S. LEE, WANZHEN LIANG, ITAY LOTAN,

NIKHIL NAIR, BARON PETERS, EMIL I. PROYNOV, PIOTR A. PIENIAZEK, YOUNG MIN RHEE, JIM RITCHIE, EDINA ROSTA, C. DAVID SHERRILL, ANDREW C. SIMMONETT, JOSEPH E. SUBOTNIK, H. LEE WOODCOCK III, WEIMIN ZHANG, ALEXIS T. BELL, ARUP K. CHAKRABORTY, DANIEL M. CHIPMAN, FRERICH J. KEIL, ARIEH WARSHEL, WARREN J. HEHRE, HENRY F. SCHAEFER III, JING KONG, ANNA I. KRYLOV, PETER M. W. GILL, AND MARTIN HEAD-GORDON, *Advances in methods and algorithms in a modern quantum chemistry program package*, Physical Chemistry Chemical Physics, 8 (2006), pp. 3172–3191.

[48] SAMEER S. SHENDE AND ALLEN D. MALONY, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, 20 (Summer 2006), pp. 287–311.

[49] EDGAR SOLOMONIK, ABHINAV BHATELE, AND JAMES DEMMEL, *Improving communication performance in dense linear algebra via topology aware collectives*, in ACM/IEEE Supercomputing, Seattle, WA, USA, Nov 2011.

[50] EDGAR SOLOMONIK AND JAMES DEMMEL, *Communication-optimal 2.5D matrix multiplication and LU factorization algorithms*, in Springer Lecture Notes in Computer Science, Proceedings of Euro-Par, Bordeaux, France, Aug 2011.

[51] J. F. STANTON AND R. J. BARTLETT, *The equation of motion coupled-cluster method. a systematic biorthogonal approach to molecular excitation energies, transition probabilities, and excited state properties*, Journal of Chemical Physics, 98 (1993), p. 7029.

[52] J. F. STANTON AND J. GAUSS, *Analytic second derivatives in high-order many-body perturbation and coupled-cluster theories: computational considerations and applications*, International Reviews in Physical Chemistry, 19 (2000), pp. 61–95.

[53] R. A. VAN DE GEIJN AND J. WATTS, *SUMMA: scalable universal matrix multiplication algorithm*, Concurrency: Practice and Experience, 9 (1997), pp. 255–274.

[54] JIŘÍ ČÍŽEK, *On the correlation problem in atomic and molecular systems. calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods*, The Journal of Chemical Physics, 45 (1966), pp. 4256–4266.