Edgar Solomonik
Jeff Hammond
James Demmel

A preliminary analysis of Cyclops Tensor Framework

Edgar Solomonik
Univ. of California, Berkeley
Department of EECS
solomon@eecs.berkeley.edu

Jeff Hammond
Argonne National Lab
Leadership Computing Facility

James Demmel
Univ. of California, Berkeley
Department of EECS

ABSTRACT

Cyclops (cyclic-operations) Tensor Framework (CTF) [1] is a distributed library for tensor contractions. CTF aims to scale high-dimensional tensor contractions done in Coupled Cluster calculations on massively-parallel supercomputers. The framework preserves tensor symmetry by subdividing tensors cyclically, producing a highly regular parallel decomposition. The parallel decomposition effectively hides any high dimensional structure of tensors reducing the complexity of the distributed contraction algorithm to known linear algebra methods for matrix multiplication. We also detail the automatic topology-aware mapping framework deployed by CTF, which maps tensors of any dimension and structure onto torus networks of any dimension. We employ virtualization to provide completely general mapping support while maintaining perfect load balance. Performance of a preliminary version of CTF on the IBM Blue Gene/P and Cray XE6 supercomputers shows highly efficient weak-scaling, demonstrating the viability of our approach.

1. INTRODUCTION

Coupled Cluster (CC) is a computational method for computing an approximate solution to the time-independent Schrödinger equation of the form

$$H|\Psi\rangle = E|\Psi\rangle,$$

where H is the Hamiltonian, E is the energy, and Ψ is the wave-function. In CC, the approximate wave-function is defined in exponential form

$$|\Psi\rangle = e^{\hat{T}}|\Phi\rangle$$

where $\Phi\rangle$ is the Slater determinant. For an N-electron system the Slater determinant is

$$|\Phi\rangle = \frac{1}{\sqrt{N!}}|\phi_1\phi_2...\phi_N|\rangle.$$

The $\hat{T}$ operator in CC has the form

$$\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 \ldots$$

where $\hat{T}_n$ is a $2n$th rank (dimension) [2] tensor representing nth level electron excitations. Each $\hat{T}_n$ is computed via a series of tensor contractions on tensors of rank $r \in \{2, 4, \ldots 2n\}$. The specific tensor contractions depend on the variation of the Coupled Cluster method and can be derived differently. Calculating only $\hat{T}_1$ and $\hat{T}_2$ gives the method commonly known as CCSD (Coupled Cluster Singles and Doubles). Additional calculation of $\hat{T}_3$ gives the CCSDT (T - triples) method and $\hat{T}_4$ gives the CCSDTQ (Q - quadruples) method.

Computationally, tensor contractions can be reduced to matrix multiplication via index reordering (transposes). This approach is efficient and commonly used for contractions on fully dense tensors. However, the tensors which arise in Coupled Cluster methods usually have high-dimensional structure. In particular, permutational symmetry or anti-symmetry among a set of indices implies that any reordering of the index set within the tensor will give the same value (with a potential sign change for anti-symmetry). For example, given a 4D tensor A with permutational symmetry among indices (i, j, and l but not k), we know

$$\begin{aligned} A[i,j,k,l] &= A[i,l,k,j] = A[j,i,k,l] = A[j,l,k,i] \\ &= A[l,i,k,j] = A[l,j,k,i]. \end{aligned}$$

Permutational symmetry is typically physically motivated in Coupled Cluster, where matrix or tensor elements are representative of interchanges among electrons or orbitals, which may have symmetric or anti-symmetric effects on the energy.

In general, permutational symmetry of d indices, implies that only one of every $d!$ values in the full tensor is unique. This implies that it suffices to store only $1/d!$ of the tensor data. In higher-order methods such as CCSDT and CCSDTQ, which have 3-dimensional and 4-dimensional symmetries as well as multiple symmetric index groups in some tensors, this memory preservation becomes pervasive. Further, any symmetry preserved within a contraction (e.g. the output C contains indices that were symmetric in operands

[1] Software and documentation publicly available under a BSD license: `http://www.eecs.berkeley.edu/~solomon/cyclopstf/index.html`

[2] We will use the term dimension to refer to tensor rank or order.

A or B), allows for a reduction in computational cost with respect to a non-symmetric contraction.

The challenge in exploiting high-dimensional symmetry is that the contractions can no longer be trivially reduced to matrix multiplication. Further, since the number of possible as well as encountered (partial) permutational symmetries grows exponentially with tensor dimension, it is difficult to generalize and tiresome to specialize. As a result, most implementations exploit tensor symmetry to a limited extent and perform redundant work and communication by unpacking or padding tensors.

We present a general parallel decomposition of tensors which exploits any partial or full tensor symmetry of any dimension, within any contraction. Our method performs minimal padding and has a regular decomposition that allows the algorithm to be mapped to a physical network topology and executed with no load imbalance. Along with the algorithms, we detail a preliminary implementation of Cyclops Tensor Framework (CTF), a distributed tensor contraction library suitable for symmetric high-dimensional tensors. On 1,536 nodes of a Cray XE6 supercomputer, the parallel decomposition used by CTF maintains 50% efficiency for symmetric 4D tensors and 25% efficiency for symmetric 8D tensors with respect to the peak theoretical floating-point performance. By exploiting symmetry for these tensors, the memory footprint is reduced by a factor of 4 for each 4D symmetric tensor, and by a factor of 576 for the 8D symmetric tensors (symmetry also reduces computational cost with respect to a nonsymmetric contraction by a large factor, but only for certain contractions).

The main contributions of Cyclops Tensor Framework are

- a symmetry-preserving algorithm for parallel tensor contractions
- a completely load balanced regular communication with minimal padding requirements
- an automatic topology-aware mapping framework for symmetric tensor contractions
- highly optimized tensor redistribution kernels
- an efficient implementation of virtualized mapping

In Section 2, we detail previous and related work. In Section 3, we describe the parallel tensor decomposition algorithm. We explain the virtualization and mapping framework in Section 4. Finally, we give preliminary performance results and analysis on IBM Blue Gene/P and Cray XE6 architectures in Section 5 and conclude in Section 6.

2. PREVIOUS WORK

We provide an overview of existing applications and known algorithms for distributed memory Coupled Cluster and tensor contractions. We also discuss parallel numerical linear algebra algorithms, in particular 2.5D algorithms [21], which will serve as a design objective and integrand of Cyclops Tensor Framework.

2.1 NWChem

NWChem [16] is a computational chemistry software package developed for massively parallel systems. NWChem includes implementations of Coupled Cluster and 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) [5, 3, 9], to automatically generate sequences of tensor contractions based on a diagrammatic representation of Coupled Cluster 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.

To parallelize and execute each individual contraction, NWChem employs the Global Arrays (GA) framework [17]. Global Arrays is a partitioned global-address space model (PGAS) and allows processors to access (fetch) data which 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 fetching 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. Since this dynamically scheduled scheme is not load balanced, NWChem uses dynamic load balancing among the processors. Further, since distribution is hidden by GA, the communication pattern is irregular and possibly unbalanced. Cyclops Tensor Framework attempts to eliminate the scalability bottlenecks of load imbalance and irregular communication, by using a regular decomposition which employs a structured communication pattern well-suited for torus network architectures.

2.2 2.5D algorithms

Since tensor contractions are closely related to matrix multiplication (MM), it is of much 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. In particular, we would like to minimize the communication (number of words of data moved across the network by any given processor) done to contract tensors. Since any tensor contraction can be transposed and done as a matrix multiplication, a lower bound on communication for MM is also valid for tensors.

Given a matrix multiplication of square matrices of dimension n on p processors, with M words of memory on each processor, it is known that some process must communicate at least

$$W = \Omega\left(\frac{n^3}{p \cdot \sqrt{M}}\right)$$

words of data [13, 4, 12]. If $M = \Omega(n^2/p)$, so the matrices just fit in memory the communication lower bound is

$$W_{2D} = \Omega\left(\frac{n^2}{\sqrt{p}}\right).$$

We label this lower bound as W_{2D} because it is achieved by algorithms that are most naturally described on a 2D processor grid. In particular, blocked Cannon's algorithm [7] and SUMMA [22] achieve this communication bandwidth lower bound. We can also see that, assuming the initial data is not replicated and load-balanced, there is an absolute (memory-size insensitive) lower-bound [2],

$$W_{3D} = \Omega(M) = \Omega\left(\frac{n^2}{p^{2/3}}\right).$$

This communication lower-bound can be achieved by performing 3D blocking on the computational graph rather than simply distributing the matrices. An old algorithm known as 3D matrix multiplication has been shown to achieve this communication cost [8, 1, 2, 14].

However, 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. In particular, given $M = O(cn^2/p)$, where $c \in [1, p^{1/3}]$, the communication lower bound is

$$W_{2.5D} = \Omega\left(\frac{n^2}{\sqrt{cp}}\right).$$

Using adaptive replication this communication lower-bound can be achieved for matrix multiplication as well as other dense linear algebra kernels via the algorithms presented in [21]. Its also important to note that 2.5D algorithms can map very efficiently to torus network architectures as demonstrated in [20]. We would like to achieve the same communication costs as well as retain good topological properties in the decomposition of tensors.

3. PARALLEL DECOMPOSITION OF SYMMETRIC TENSORS

To consider the computational and communication costs of tensor contractions we will define a model tensor contraction problem. We consider a partially-symmetric tensor of dimension d with all sides of length n. Both the operands and the result have 2 partial-symmetries among each half ($d/2$) of their indices. We can consider the cost of calculating a contraction over $d/2$ indices, in fully packed form. We can reduce the cost of the full contraction to some multiple of the cost of a contraction where all unpacked elements are assumed to be zero (to compute the full symmetric contraction might require one or multiple of such contractions). A 4D example of such a contraction looks like

$$C_{a\leq b}^{k\leq l} = \sum_{ij} A_{a\leq b}^{i\leq j} \cdot B_{i\leq j}^{k\leq l}.$$

We study this type of contraction because it has the combined complexity of high-dimensional symmetry as well as partial-symmetry. Further, this contraction can actually be folded into a matrix multiplication, so we can conveniently analyze the overhead of high-dimensional blocking with respect to the completely optimal approach of matrix multiplication (which is of course not suitable for general symmetries). In this model contraction, the packed size of each tensor per processor is

$$S(n,d,p) = \Theta\left(\frac{n^d}{p\cdot((d/2)!)^2}\right)$$

and the total number of multiplications (flops which must be performed) per processor is

$$F(n,d,p) = \Theta\left(\frac{n^{3d/2}}{p\cdot((d/2)!)^3}\right).$$

Efficient parallelization as well as sequential execution requires blocking of computation. For matrix multiplication, it is well known that blocking the matrices gives a communication-optimal algorithm when no extra memory is available. Blocking the computational graph gives communication-optimality when each processor has an unlimited amount of memory. Generally, the blocking should be done to minimize the surface-area to volume ratio of each computational block. For matrix multiplication of square matrices, this is achieved by picking out square blocks. For dense tensor contractions, it is ideal to pick out blocks for which the accumulated block-size of all dimensions which are contracted over is the same or near the accumulated block-size of all dimensions which are not contracted over. If half the indices are being contracted over (as in our model contraction), then to achieve optimality, it suffices to pick the same block-size in each dimension.

3.1 Padding overhead of blocking

Tensor symmetry complicates the process of blocking, since a symmetric packed tensor is not decomposable into equivalent contiguous blocks. One approach is to decompose the packed tensor into full and partially-full blocks and deal with each case-by-case. However, it becomes very difficult to maintain and specialize for each type of partially-full block that arises given some blocking of a tensor of some symmetry. Further, computation on partially-full blocks produces load-imbalance and an irregular decomposition. Therefore, to greatly simplify implementation, partially-full blocks are typically padded with zeros and treated as fully dense blocks. A block size of b implies that $O(b)$ padding is required along each dimension.

In matrix computations, padding is almost always a trivially low-order cost but in tensors of high-dimension it can incur a significant computational overhead. Given a padding thickness of one in each dimension, the padding has a total size proportional to the surface area of the tensor. The surface area of a tensor is one-dimension less than the tensor itself, so the surface-area to volume ratio grows with respect to tensor dimension. In particular, for our model problem, given blocking (padding) of size b, the computational cost grows in proportion to

$$\begin{aligned} F_{\text{padded}}(n,d,p,b) &= O\left(b^{3d/2}\frac{(1+\frac{n}{b})^{3d/2}}{p\cdot((d/2)!)^3}\right) \\ &= O\left(\frac{n^{3d/2}}{p\cdot((d/2)!)^3} + \frac{(3d/2)b\cdot n^{3d/2-1}}{p\cdot((d/2)!)^3} + \dots\right) \\ &= O\left((1+(3d/2)b/n)\cdot F(n,d,p)\right). \end{aligned}$$

Thus, padding can increase the computation performed significantly if $b \approx n$.

3.2 Cyclic distribution of blocks

A blocked distribution implies each processor owns a contiguous piece of the original tensor, while a cyclic distribution implies that each processor owns every element of the tensor with a defined cyclic phase. In a cyclic distribution, a cyclic phase defines the periodicity of the set of indices

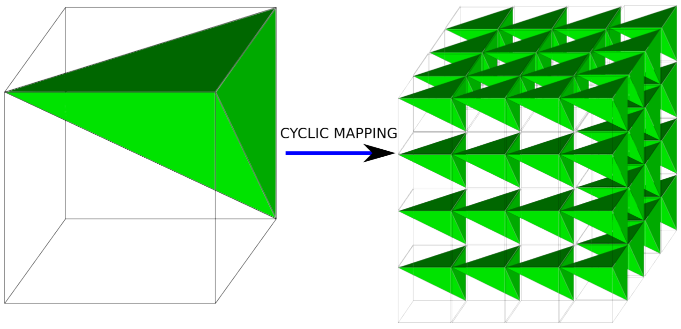


Figure 1: A cyclic decomposition of a 3D symmetric tensor into symmetric blocks on a 4-by-4-by-4 topology.

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 $(p_0, p_1, \cdots, p_{d-1})$, such that processor $P_{i_0,i_1,\cdots,i_{d-1}}$ owns all tensor elements whose index $(j_0, j_1, \cdots, j_{d-1})$ satisfies

$$j_k = i_k \text{mod}(p_k)$$

for all $k \in \{0, 1, \cdots, d\}$. 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) [6] and Elemental (cyclic) [18].

Like matrix multiplication, tensor contractions are invariant with respect to a similarity permutation on A and B,

$$PCP^T = PA \cdot BP^T = (PAP^T) \cdot (PBP^T)$$

This invariance means that we can permute the ordering of rows in columns in a matrix or slices of a tensor, so long as we do it to both A and B and permute PCP^T back to C. This property is particularly useful when considering cyclic and blocked distributions of matrices and tensors. We can define a permutation, P, that permutes a tensor elements from a blocked to a cyclic layout. Conversely, we can run an algorithm on a cyclic distribution and get the answer in a blocked distribution by applying a permutation, or run *exactly* the same algorithm on a blocked distribution and get the cyclic answer by applying a permutation.

The main idea behind Cyclops Tensor Framework is to employ a cyclic distribution to preserve symmetry in subtensors, minimize padding, and generate a completely regular decomposition, susceptible to classical linear algebra optimizations. Each processor owns a cyclic sub-tensor, with a symmetric choice of cyclic phases in each dimension. By maintaining the same cyclic phase in each dimension, the algorithm insures that each the sub-tensor owned by any processor has the same symmetry and structure as the whole tensor (Figure 1). Further, minimal padding on each sub-tensor insures that every sub-tensor has the exact same dimensions, now only with different values. Figure 2 demonstrates the difference in padding (or load-imbalance) required to store exactly the same sub-tensors on each processor. It is evident that only a cyclic layout can preserve symmetry as well as maintain load balance. Overall the amount of padding required for CTF, is equivalent to setting the block size $b = p^{1/d}$, since we must add up the padding on each processor.

3.3 Communication cost

Since the dense version of this contraction is equivalent to multiplication, we can employ the matrix multiplication communication lower bound. In particular, given any block of data of each of A, B, and C of size $O(M)$, no more than

$$F_{\text{blk}}(M) = O(M^{3/2})$$

meaningful computational work can be performed [13, 4, 12]. Therefore, the amount of communication required to perform the contraction can be lower bounded by the number of blocks (of size equal to processor memory) needed to perform the contraction

$$\begin{aligned} W(n,d,p,M) &= \Omega(F(n,d,p) \cdot M/F_{\text{blk}}(M)) \\ &= \Omega\left(\frac{n^{3d/2}}{p \cdot M^{1/2} \cdot ((d/2)!)^3}\right). \end{aligned}$$

If we assume the tensor is the same asymptotic size as the available memory, the block size is

$$\begin{aligned} M &= S(n,d,p) \\ &= \Theta\left(\frac{n^d}{p \cdot ((d/2)!)^2}\right), \end{aligned}$$

which gives us a lower bound on the number of words moved as a function of n, d and p,

$$W(n,d,p) = \Omega\left(\frac{n^d}{\sqrt{p} \cdot ((d/2)!)^2}\right).$$

3.3.1 Communication in Cyclops Tensor Framework

For this model tensor contraction, the number of words moved by Cyclops Tensor Framework (CTF) the size of a packed tensor block, multiplied by the number of times it is broadcast during the contraction. The CTF decomposition employs a d-dimensional processor grid, where each edge-length of the processor grid is $p^{1/d}$. The algorithm employed would be a blocked recursive SUMMA ($d/2$ levels of recursion, one per contraction index), with each block of size equal to the packed tensor size per processor. The number of words communicated per processor is then

$$\begin{aligned} W_{\text{CTF}}(n,d,p) &= O(S(n,d,p) \cdot (p^{1/d})^{d/2}) \\ &= O\left(\frac{n^d}{\sqrt{p} \cdot ((d/2)!)^2}\right). \end{aligned}$$

So the amount of data sent is optimal. We note that this is not necessarily the case for any given tensor contraction, but

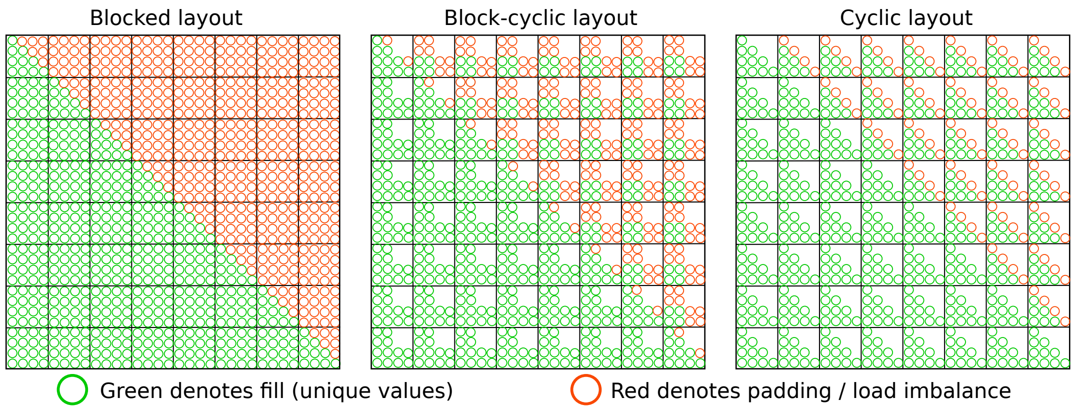

Blocked layout
Block-cyclic layout
Cyclic layout
Green denotes fill (unique values)
Red denotes padding / load imbalance

Figure 2: Padding or load-imbalance in tensor mapped with different block-sizes.

is true for the model problem. In general, CTF can have a small theoretical communication overhead (with respect to the lower-bound) if the flop-to-byte ratio in the symmetric contraction is smaller than the flop-to-byte ratio of a dense matrix multiplication.

The only other significant communication costs in Cyclops Tensor Framework are the all-to-all communication necessary for tensors redistribution. Since this communication requires no processor to send more words than its local subtensor size, it is a low order cost with respect to the contraction.

3.3.2 *Communication in NWChem*

In NWChem, a tile-size b is selected in each dimension of the tensors. For each tile sub-contraction, the data is retrieved via one-sided communication from its location. In our model contraction, tiles with edge-length of b in each dimension can be contracted by performing $b^{3d/2}$ flops. Therefore, the amount of data communicated by each processor is

$$W_{\text{NW}}(n,d,p) = O\left(\frac{n^{3d/2}}{p\cdot b^{3d/2}\cdot((d/2)!)^2}\cdot b^d\right)$$
$$= O\left(\frac{n^{3d/2}}{p\cdot b^{d/2}\cdot((d/2)!)^2}\right).$$

To achieve communication optimality, this block-size must be picked to be $b=\Omega(n/p^{1/d})$,

$$W_{\text{NW}}(n,d,p) = O\left(\frac{n^{3d/2}}{p\cdot(n/p^{1/d})^{d/2}\cdot((d/2)!)^2}\right)$$
$$= O\left(\frac{n^d}{\sqrt{p}\cdot((d/2)!)^2}\right).$$

However, picking this block size implies that NWChem needs to perform a large amount of padding, increasing the computational cost by a factor of

$$F_{\text{NW-padded}}(n,d,p) = O\left((1+(3d/2)/p^{1/d})\cdot F(n,d,p)\right).$$

This factor grows with tensor dimension and has a significant overhead for tensors of high dimension in practice. On the other hand, CTF has a padding computational overhead of

$$F_{\text{CTF-padded}}(n,d,p) = O\left((1+(3d/2)p^{1/d}/n)\cdot F(n,d,p)\right).$$

Since n grows proportionally to $p^{1/d}$ with both p as well as d, this overhead is significantly smaller.

4. TOPOLOGY-AWARE MAPPING

To preserve symmetry in sub-tensor of a tensor we must ensure that the parallel decomposition uses the same cyclic phase among symmetric dimensions of that tensor. However, simultaneously its necessary to enforce the same cyclic phase among a dimension of two tensors which is being contracted over (shared index). Such parameters severely limited the possible block decompositions that are available. However, we would like to have the flexibility of mapping onto any physical topology. This motivates a virtualized mapping of the tensors, which overdecomposes the tensors and then maps the blocks onto the processing grid.

4.1 Virtualized mapping scheme

Cyclops Tensor Framework performs virtualization to create a level of indirection between the task decomposition and the physical network topology. We provide a virtualization scheme that is guaranteed to generate a load balanced decomposition for any given tensor contraction (tensors of any symmetry, any dimension, and any index map defining the contraction). Further, we parameterize the virtual decomposition so that it is effectively a multiple of the processor grid, which insures that each processor owns the same number of sub-blocks. This scheme effectively reduces the problem of mapping a tensors with symmetry to mapping padded tensors with no symmetry. For example, in Figure 3 the 3D virtualized mapping is decomposed among the processors so that each processor is contracting a matrix of symmetric tensors with a vector of symmetric tensors into a matrix of symmetric tensors. The mapping is defined so that by the time the distributed contraction algorithm is executed, it need not be aware of the symmetry of the sub-tensors but only of their size.

We do not use a dynamically scheduled virtualization approach such as the overdecomposition embodied by the Charm++ runtime system [15]. Instead, we define the virtualization so that its dimensions are a multiple of the physical torus dimensions and generate a regular mapping. This approach maintains perfect load-balance and achieves high communication and task granularity by managing each virtualized sub-grid explicitly by a single process.

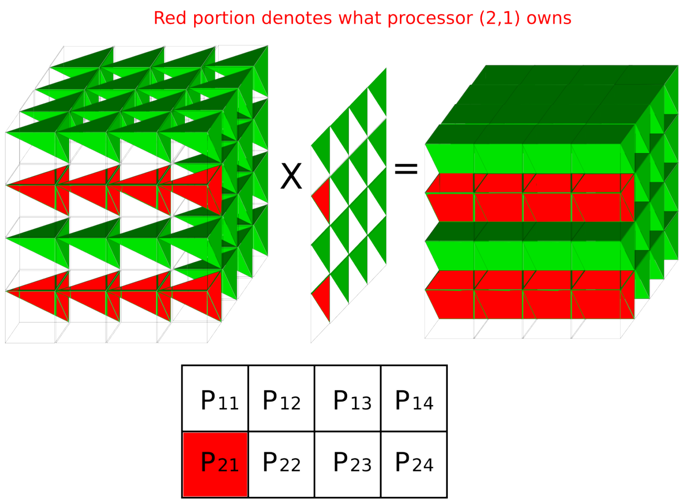


Figure 3: Virtualization as used in CTF to perform contractions.

A tensor mapping decomposed by Cyclops Tensor Framework maintains the following properties

- If two indices i, j belong to a tensor and have symmetry $(i \leq j)$, they must be mapped with the same cyclic phase.
- If a contraction index i belongs to both tensors A and B, the i-index dimension of A and B must be mapped with the same cyclic phase and not along the same physical processor grid dimension.
- If a contraction index i belong to either A and C or B and C the i-index dimension of A and C or B and C must have the exact same mapping.
- Each tensor is distributed or along the entire processor grid.

Obeying the above mapping conditions allows for efficient generation of a symmetry-oblivious contraction algorithm that computes the solution following the redistribution. The generated distributed contraction algorithm simply applies known matrix multiplication distributed algorithms recursively for each index which is contracted over. Since we do not allow dimensions of A and B which are contracted over to be mapped onto the same physical dimension, they must form a 2D decomposition (with one or both dimensions potentially virtualized). Further, since each index of C is mapped onto the same dimension as a corresponding index of A or B, we know that the answer will end up in the right place.

4.2 Mapping heuristics

Currently, Cyclops Tensor Framework assumes the physical topology is a torus and attempts to find the best virtualized mapping for the given torus. No assumptions are made about the dimension of the torus or lengths of any torus dimension. Further, the torus is folded in all possible ways that maintain the original dimensional ordering (this assumption is not very restricting and is made for convenience). If no topology is specified, the processor count is fully factorized $p = p_1 \cdot p_2 \cdot \ldots \cdot p_d$, to form a d-dimensional torus processor grid with edge lengths $p_1, p_2 \ldots p_d$. This fully unfolded processor grid is then folded back-up so as to generate the largest amount of different potential decompositions.

Once a tensor is defined or a tensor contraction is invoked, CTF searches through all topologies and selects the best mapping which satisfies the constraints. 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 a trivial amount of time with respect to the contraction. Once a mapping is decided upon, the tensors are redistributed.

When trying to select a mapping of a tensor or of a contraction of tensors onto a physical processor grid, a greedy algorithm is used. The longest tensor dimension is mapped along the longest dimension of the processor grid. For each such mapping of a dimension, CTF enforces all other constraints by mapping all other symmetric or related dimensions to available physical or virtual dimensions with the same phase. For contractions, dimensions which are contracted over are mapped first and combined together span the entire processor grid (this is necessary to insure A and B are distributed over the whole processor grid). The remaining dimensions (dimensions which are not contracted over and belong to C) are mapped so that C is distributed over the whole processor grid, again insuring no constraints are violated. Enforcing the constraints is done by iterating over all dimensions and increasing virtualization when necessary.

The best mapping can be selected according to many conditions, such as whether a redistribution of data is required, how much memory the distributed algorithm will use, how much virtualization is required and so forth. The current heuristic is to maximize the amount of computational work done in each sub-contraction. This heuristic effectively maximizes the volume of computation in each block, which is a condition that ensures communication optimality.

4.3 Tensor redistribution

Each contraction can place unique restrictions on the mapping of the tensors. Therefore, to satisfy each new set of restrictions the mapping must change and tensor data must be reshuffled among processors according to the new mapping. Since the redistribution can potentially happen between every contraction, an efficient implementation is necessary. However, the data must first be given to CTF by the user application. We detail a scheme for input and output of data by key-value pairs, as well as a much more efficient algorithm for mapping-to-mapping tensor redistribution. Since Coupled Cluster and most other scientific applications are iterative and perform sequences of operations (contractions) on the same data, we assume input and output of data will happen less frequently than contractions.

To support general and simple data entry, CTF allows the user to write tensor data bulk-synchronously into the tensor object using key-value pairs. This allows the user to write data from any distribution that was previously defined, and to do so with any desired granularity (all data at once or by chunks). Redistribution happens by calculating the cyclic phase of each key to determine which processor it belongs on. Once counts are assembled the data is redistributed via all-to-all communication. After this single redistribution phase, each process should receive all data belonging to its sub-tensors, and can simply bin by virtual block then sort it locally to get it into the right order. This key-value binning scheme is essentially as expensive as a parallel sorting algorithm.

When transitioning between distributions, which we expect to happen much more frequently than between the application and user, we can take advantage of existing knowledge about the distribution. To be more precise, such a redistribution must manipulate the local tensor data, which is laid out in a a grid of virtual blocks, each of which is a symmetric tensor of the same size. The dimensions and edge-lengths of the virtual and physical processor grid can change between mappings and therefore must be accounted for during redistribution. CTF supports such redistributions generally using the following algorithm

1. Iterate over local tensor data and compute the virtual sub-tensor destination of each element.

2. Consider the assignment of virtual sub-tensors to physical processes to compute how many elements are received by each process into each of its virtual buckets.

3. Iterate over sub-tensor data and the virtual grid (in inverse hierarchical order with respect to the data) so as to preserve global element order, and move each element to the correct send buffer.

4. Exchange the data among processes with all-to-all communication.

5. Iterate over the new sub-tensor data and the new virtual grid (again in inverse hierarchical order with respect to data), preserving global element order, and reading out of received buffers (which are accordingly ordered).

The above redistribution algorithm is complex due to the necessity of maintaining global element order within the buffers. By maintaining global element ordering, we eliminate the need for keys, since this ordering is preserved in the old and the new mapping.

After threading the redistribution algorithm with OpenMP, we found the integer logic running time to be negligible with respect to contraction time. Threading the redistribution kernel required iterating over different parts of the sub-tensor when writing data to and from buffers. Overall, the redistribution kernels are the most complex part of Cyclops Tensor Framework implementation, because they must explicitly and efficiently deal with the representation of the symmetric sub-tensors as well as the virtualization. However, after the redistribution, the parallel contraction kernel only takes into account the virtualized grid, while the sequential contraction kernels need to only consider the sub-tensor, creating a vitally important software engineering separation.

5. PARALLEL PERFORMANCE

Cyclops Tensor Framework is still under development but much of the infrastructure is in-place and functional. The framework can already decompose tensors of dimension at least up to 8 (though no dimensional limit exists in the code) with partial or full symmetries of at least up to 4 indices. However, no optimized sequential symmetric contraction kernels have been coupled with CTF yet. Further, some optimizations, such as adaptive replication are not yet in-place. We give scalability results for the model contraction problem, using matrix multiplication as the sequential kernel but employing high-dimensional of parallel decompositions of symmetric tensors of dimension 4,6, and 8. The framework achieves highly efficient weak-scaling on multiple supercomputer architectures demonstrating the viability of our approach.

5.1 Implementation

Much of the logic and kernels contained by CTF has been described in the previous sections. Here, we summarize and describe the overall work-flow of the contraction library. At a high-level CTF has the following execution mechanism,

1. Describe or automatically detect the physical network topology.

2. Define distributed tensor objects with symmetries and map the tensors.

3. Write data to tensors bulk-synchronously via (global index, value) pairs.

4. Invoke contraction/summation/scale/trace operation on tensor(s).

5. In parallel, determine best mapping for the operation and define operational kernel.

6. Redistribute tensors to the selected decomposition and topology.

7. Run the contraction kernel on the tensors, computing the solution.

8. When requested, read data out via (global index, value) pairs.

Tensors are treated as first-class objects and direct access to data is hidden from the user. The data must be written and read bulk-synchronously by global index. This data entry mechanism is necessary to be able to remap and abstract away the data decomposition from the user. While the initial redistribution is somewhat slower than a mapped redistribution, the assumption is that the data will be read and written infrequently and contractions will be done in sequences.

The implementation uses no external libraries except for MPI [10], BLAS, and OpenMP. We used vendor provided optimized BLAS implementations (IBM ESSL and Cray LibSci) on all architectures for benchmarking. All code is tightly integrated and written in C/C++. Computationally expensive routines are threaded and/or parallelized with MPI. Performance profiling is done by hand and with TAU [19].

5.2 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/P and Cray XE6.

We benchmarked our implementations on a Blue Gene/P (BG/P) [11] machine located at Argonne National Laboratory (Intrepid). BG/P is an interesting target platform because it uses few cores per node (four 850 MHz PowerPC processors) and relies heavily on its interconnect (a bidirectional 3D torus with 375 MB/sec of achievable bandwidth per link). Therefore, interprocessor communication and topology-awareness are key considerations on this architecture.

Our second experimental platform is 'Hopper', which is a Cray XE6 supercomputer, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. This machine is located at the NERSC supercomputing facility. Each node can be viewed as a four-chip compute configuration due to NUMA domains. Each of these four chips have six superscalar, out-of-order cores running at 2.1 GHz with private 64 KB L1 and 512 KB L2 caches. Nodes are connected through Cray's 'Gemini' network, which has a 3D torus topology. Each Gemini chip, which is shared by two Hopper nodes, is capable of 9.8 GB/s bandwidth. However, the NERSC Cray scheduler does not allocate contiguous partitions, so topology-aware mapping onto a torus cannot currently be performed.

5.3 Results

All benchmark scaling results are collected for the model tensor contraction problem with square tensors of dimension 2,4, and 8, using matrix multiplication as the sequential subcontraction kernel. Testing of the parallel decomposition has been done for all types of contractions using a sequential kernel that unpacks each symmetric sub-tensor. Full results for tensor contraction sequences and actual Coupled Cluster methods will require integrated sequential symmetric contraction kernels. However, the study detailed here tests the full complexity of the parallel component, which has no awareness that it is working on a model contraction rather than any given Coupled Cluster contraction (an 8D model contraction has as high dimension and as much symmetry as any CC contraction done in methods up to CCSDTQ). Further, we defined the contractions so as to force each tensor to be redistributed between every contraction, benchmarking the worst case scenario for our redistribution kernels.

Figure 4(a) demonstrates weak scaling performance of CTF on a Blue Gene/P supercomputer. The scalability is good, but a significant amount of extra computation is done for 8D tensors (less so for 6D and very little for 4D). The amount of padding done for an 8D tensor, quadruples the computational cost in the worst observed case. This overhead is correlated with the fact that each sub-tensor gets to be very small and the padded/surface area dominates. However, this padding does not grow with processor count and stays bounded. The performance dip for 8D tensors at 512 nodes is due to increased virtualization going up from 256 nodes (256 nodes can be treated as an 8D hypercube). On 512 nodes, the mapping must have more virtualization and therefore more padding. However, this increase levels off and the virtualization necessary actually decreases from 512 nodes onward (it should continue decreasing until reaching $2 \cdot 4^8 = 131,072$ nodes). In this preliminary benchmark, topology-aware mapping is not done onto the torus dimensions. In any case, it is difficult to map 6D and 8D tensors to a 3D physical processor grid. A 5D or 6D processor grid such as the ones employed by the newer BG/Q and the K-computer, respectively, will be much more fit for mapping of these tensors.

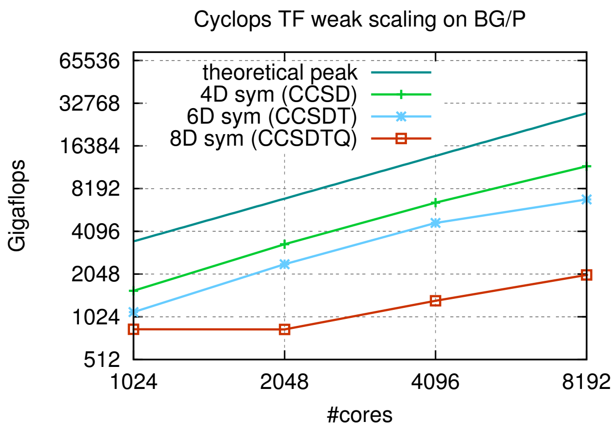
Weak scaling on the Cray XE6 architecture (Figure 4(b)) has similar characteristics as the scaling on BG/P. In fact, the padding overhead for 6D and 8D tensors is significantly decreased, since each process (4 per node, 6 cores per process) handles much larger granularity (sub-tensor size) than BG/P nodes can, due to the difference in on-node memory size. Despite the lack of precise topology-aware mapping, communication overhead is small at the given node counts. For both BG/P and XE6, most of the execution time is spent inside sequential matrix multiplication, which demonstrates that the parallel efficiency is very high ($\geq 50\%$).

We do not detail strong scaling performance, since memory replication is not yet fully implemented inside CTF. Given 2.5D-like replication schemes within CTF the strong scalability will significantly improve. When running the same size problem on more nodes, more memory becomes available, allowing for more replication and reduced communication. A theoretical analysis and a detailed performance study of strong scaling done in this fashion is given in [20].

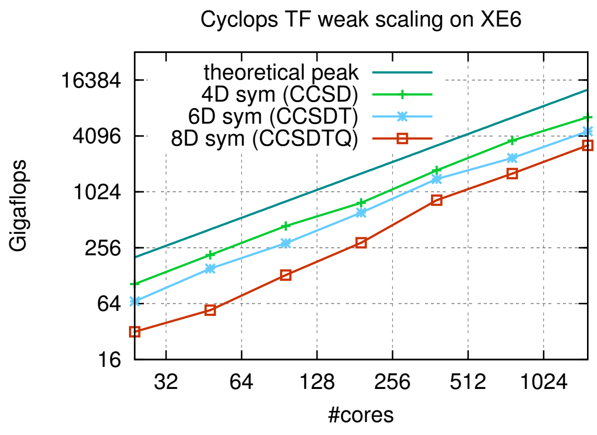
6. FUTURE WORK

Cyclops Tensor Framework is currently in development with several major components existing and some missing. In particular, there are no integrated efficient symmetric sequential contraction routines. Currently only unoptimized routines are employed for verification and optimized matrix multiplication is employed for benchmarking of certain contractions. Since CTF preserves any tensor symmetry, it allows for any sequential contraction kernel to be seamlessly parallelized, by using it as a function pointer and calling it many times on sub-tensors. We will explore general and auto-generated routines for sequential tensor contractions. Further, we will couple the sequential contraction design with the virtual decomposition by exploiting the blocking implicitly provided by virtualization.

Adaptive replication and memory awareness will be integrated into CTF. Part of the mapping process will involve



(a) IBM BG/P



(b) Cray XE6

Figure 4: Preliminary weak scaling results of CTF

calculating the necessary buffer space needed to perform a contraction and selecting the algorithm which performs the least amount of communication without surpassing the memory limits. This methodology has been employed in 2.5D algorithms for linear algebra and shown to significantly improve strong scaling as well as allow more flexibility in the decomposition.

Different types of sparsity in tensors will also be considered in Cyclops Tensor Framework. Tensors with banded sparsity structure can be decomposed cyclically so as to preserve band structure in the same way CTF preserves symmetry. Completely unstructured tensors can also be decomposed cyclically, though the decomposition would need to perform load balancing in the mapping and execution logic.

Cyclops Tensor Framework will also be integrated with a higher-level tensor manipulation framework as well as Coupled Cluster contraction generation methods. Currently, CTF performs contractions on tensors in a packed symmetric layout. However, multiple packed contractions are required to compute the full symmetric contraction and the higher-level Coupled Cluster equations can be decomposed into contractions in different ways. We will integrate CTF vertically to build a scalable implementation of Coupled Cluster 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 Coupled Cluster method (excitations past quadruples have a negligible contribution). In this report we've given preliminary results demonstrating the scalability of CTF decomposition of 4 to 8-dimensional tensors on massively parallel supercomputers. We will implement further optimizations and infrastructure that will improve strong scalability and allow for efficient execution of Coupled Cluster methods rather than just selected tensor contractions.

Acknowledgments

The first author was supported by a Krell Department of Energy Computational Science Graduate Fellowship, grant number DE-FG02-97ER25308. This research was supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). It was also supported by U.S. Department of Energy grants numbered DE-SC0003959, DE-SC0004938, DE-FC02-06-ER25786 and DE-SC0001845. This research used resources of the Argonne Leadership Computing Facility 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, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

7. 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.*, 39:575–582, September 1995.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3 – 28, 1990.

[3] E. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, H. Krishnan, C. chung Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, and E. Sibiryakov. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. 2006.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in linear algebra. *SIAM J. Mat. Anal. Appl.*, 32(3), 2011.

[5] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. 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(2):276 –292, feb. 2005.

[6] 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.

[7] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Bozeman, MT, USA, 1969.

[8] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[9] X. Gao, S. Krishnamoorthy, S. Sahoo, C.-C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Efficient search-space pruning for integrated fusion and tiling transformations. In *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin / Heidelberg, 2006.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[11] IBM Journal of Research and Development staff. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52:199–220, January 2008.

[12] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017 – 1026, 2004.

[13] H. 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, pages 326–333, New York, NY, USA, 1981. ACM.

[14] S. L. Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Comput.*, 19:1235–1257, November 1993.

[15] L. V. Kale and S. 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, pages 91–108, New York, NY, USA, 1993. ACM.

[16] R. A. Kendall, E. Apra, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. Straatsma, T. L. Windus, and A. T. Wong. High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Physics Communications*, 128(1-2):260 – 283, 2000.

[17] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996. 10.1007/BF00130708.

[18] J. Poulson, B. Maker, J. R. Hammond, N. A. Romero, and R. van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. in press.

[19] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[20] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Supercomputing, Seattle, WA, USA*, Nov 2011.

[21] E. Solomonik and J. Demmel. Communication-optimal 2.5D matrix multiplication and LU factorization algorithms. In *Lecture Notes in Computer Science, Euro-Par, Bordeaux, France*, Aug 2011.

[22] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.