A parallel, object-oriented implementation of tensor multiplication

Bryan Rasmussen*, Scott Baden†

July 24, 2007

Abstract

We describe a set of serial and parallel C++ classes for the multiplication of arbitrary-rank tensors with an arbitrary number of reductions. The classes include multiplication functions and a reasonable set of manipulations and file access functions. They also include functions to generate pieces of the tensors on-the-fly. These functions use index transformations from characteristic matrices and computable "core tensors" to achieve significant savings in communications and storage. We test the routines on a series of rank-4 tensors and demonstrate that the parallel algorithms can multiply large tensors with relatively low memory and communication requirements. Numerical experiments up to 256 processors indicate that they scale very well in several measurements.

1 Introduction

This paper outlines several algorithms for multiplying arbitrary-rank tensors with an arbitrary number of reductions. We implement them through a set of C++ classes, some of which work in serial and others that use MPI for large-scale parallel machines. The classes provide a set of useful, general-purpose utilities for manipulating large tensors. The following two subsections give some background information about tensor manipulation and contrast the current approach with other tensor multiplication software in the literature.

1.1 Mathematical background

Recall that a *tensor* is an extension of the concept of a linear operator to multilinear algebra. The motivating application for the current project is in computational chemistry [4, 12, 17, 18, 21], but tensors have many applications in disparate areas such as fluid and solid mechanics, general relativity, and quantum mechanics [5, 16]. A full treatment of multilinear algebra and continuum mechanics is beyond the scope of this paper, but we include a few definitions and examples below for reference. Many texts on multi-linear algebra are available, for example [19].

*Los Alamos National Lab; `bryanras@lanl.gov`; LA-UR 07-4529
†University of California, San Diego; `baden@cs.ucsd.edu`

The most precise way to define tensors is to use ordered outer products over multiple, generalized vector spaces. We assume the existence of standard bases in this paper, however, so we do not keep track of basis transformations (except in the explicit examples outlined below). In particular, the data structures described below are insensitive to whether indices are covariant are contravariant, so the reader may perceive of tensors only as multi-dimensional boxes of numbers with an organizational structure.

The *rank* of the tensor is the number of dimensions in the box. A rank-1 tensor is indistinguishable from a vector, and a rank-2 tensor is indistinguishable from a matrix.

We represent these boxes using subscript index notation, so, for example, a rank-3 tensor, u, with dimensions 3, 2, and 5 is written u_{ijk}, where $i \in \{1, 2, 3\}$; $j \in \{1, 2\}$; and $k \in \{1, 2, 3, 4, 5\}$. The convention is to use lower-case letters as subscripts and corresponding upper-case letters as the range of that subscript with starting index 1. In this example, we write $\{I, J, K\} = \{3, 2, 5\}$.

Tensors inherit most of the usual operations such as summation and scalar multiplication from the field, which we assume to be the real numbers. The most general operation between two tensors simply multiplies every pair of elements to create a new tensor with a rank equal to the sum of the two original ranks. For example, if u_{ijk} is a rank-3 tensor and v_{mn} is a rank-2 tensor, then multiplying them creates a rank-5, tensor, which we write

$$w_{ijkmn} = u_{ijk} v_{mn}. \tag{1}$$

If an expression contains one or more repeated indices, then it implies the Einstein summation convention, which means that we sum over the repeated "dummy" indices. Two repeated indices in Eqn. (1) would result in a rank-1 tensor, for example,

$$w_j = u_{ijk} v_{ik} = \sum_{i=1}^{N_i} \sum_{k=1}^{N_k} u_{ijk} v_{mk}. \tag{2}$$

This is sometimes known as *tensor contraction* or *reduction*.

Many tensors have a simple representation with respect to one basis, but a more complicated representation with respect to another. It is often much cheaper to store the simpler version, plus some transformation rules, than it is to store the tensor in several forms. Our software contains elements and functions that allow us to calculate a *k-index transformation*. This means that a rank-k tensor, v, has a rank-k "core tensor," g, that is computable from analytic expressions, and a set of k matrices called "characteristic matrices," z^i. The transformation provides any element of the tensor using the formula,

$$v_{i_1 i_2 \ldots i_k} = z^1_{i_1 j_1} z^2_{i_2 j_2} \cdots z^k_{i_k j_k} g_{j_1 j_2 \ldots j_k}. \tag{3}$$

This transformation becomes very important for minimizing storage and communication costs in the parallel algorithm.

Here is where the motivating application becomes apparent. The k-index transformation in computational chemistry represents a change in basis from an atomic-orbital basis to a molecular-orbital basis, the so-called "AO-to-MO" transformation. For our purposes, however, it mostly serves to avoid the storage of large tensors on disk.

1.2 Computational background

The codes presented here are very generic. They can multiply two tensors of any rank with any number of reductions. The only restriction is that if p is the number of reductions, then the contracted indices be the last p indices of both tensors, as in

$$w_{a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n} = u_{a_1 a_2 \ldots a_m c_1 c_2 \ldots c_p} v_{b_1 b_2 \ldots b_n c_1 c_2 \ldots c_p}. \tag{4}$$

At a minimum, we would like to be able to multiply two rank-4 tensors with two reductions, where both tensors potentially have very large dimensions.

The literature of recent decades contains many serial and parallel algorithms for tensor manipulation [10, 15]. Probably the most sophisticated and complex parallel version is the "Tensor Contraction Engine" (TCE) [1, 3, 6, 8, 9, 14]. This is an intensive, collaborative project spread out among several locations, distributed with the NWChem computational chemistry package [11].

The TCE is not a code suite *per se*. Rather, is it is an interpreter that generates optimized `FORTRAN` code based on a user's description of chemical reactions in a high-level language. The actual algorithms used to perform tensor multiplication may change depending on the size and structure of the multiplications, number of processors, available memory, and other factors. Determining an optimal algorithm based on these factors is a very complicated task and is an active area of research [13, 14].

On the other end of the spectrum is a set of `MATLAB` classes described in [2]. These classes institute a different philosophy from the TCE in that they emphasizes ease of use, flexibility, and visualization at the expense of speed. Moreover, the `MATLAB` codes are strictly serial, except for internal use of multiple cores in the commercial software itself. The primary advantage of a commercial product such as `MATLAB` is that it allows for very fast prototyping and instant analysis of the results.

The codes outlined in this paper do not attempt to emulate the TCE's high-level constructs, nor do they run under the `MATLAB` environment. We instead implement all tensor manipulation functions in a series of C++ classes that a user can call directly. The code performs well on a variety of problems and scales in a reasonable way as the tensors grow large.

Our approach is closer to the `MATLAB` example than the TCE, in that our classes are very generic and are not tilted toward one application. Their purpose is to allow for rapid code development without having to go through a front-end, application-specific language. While development in C++ is certainly not as fast as development in `MATLAB`, our classes are definitely an improvement in one sense: they can handle significantly larger tensors than the (serial) `MATLAB` classes because they are inherently parallel with out-of-core capability. The approach outlined in the following sections therefore represent a compromise between computational speed, ease-of-use, flexibility, and simplicity.

2 Approach

Our development is two-fold. First, we consider a serial version and discuss some alternatives and modifications. Second, we discuss ways to parallelize it. Recall that the minimum requirement is the product of two rank-4 tensors with two

reductions each. That is, we want to evaluate Eqn. (4) repeatedly with $m = n = p = 2$.

2.1 Serial algorithm

There are several common techniques for multiplying tensors with or without reductions. One option is to "unwrap" tensors into large matrices, then treat the tensor operations as an equivalent matrix multiplication. This is a generalization of the different approaches to matrix-matrix multiplication—when multiplying two matrices, one may consider the problem *in toto*, as a series of matrix-vector multiplications, as a series of vector inner products, or, most fundamentally, as a series of scalar operations.

The advantage of reshaping tensors into matrices is that it allows the use of optimized algorithms. For example, we could apply the level 3, 2, and 1 Basic Linear Algebra Subprograms (BLAS) for matrix-matrix, matrix-vector, and vector-vector decompositions respectively. Alternatively, for large tensors in parallel we may use out-of-core versions of standard techniques such as Cannon's algorithm [7]. Most applications to date have explored ways to convert tensor operations to matrix operations [6, 20].

All arithmetic in this paper, however, is confined conceptually to the scalar level. Our code does in fact use C-interface BLAS internally when available, but for every use of those libraries, we provide equivalent scalar loops so that a user may compile and run without optimized libraries. The reader may therefore picture our algorithms as purely scalar loops without losing any understanding of the essential concepts. Emphasizing low-level operations allows us to produce a single code for arbitrary-rank tensors with arbitrary reductions without adding too much complexity. Also, it allows us to study the internal operations in detail and split up the tensors more easily in parallel.

The multiplication function is therefore equivalent to a straight serial loop over all combinations of indices. In the notation of Eqn. (4), the number of arithmetic operations is

$$\# = \begin{cases} 2C1C2\ldots CpA_1A_2\ldots A_mB_1B_2\ldots B_n & p > 0 \\ A_1A_2\ldots A_mB_1B_2\ldots B_n & \text{otherwise} \end{cases} . \tag{5}$$

For example, if all dimensions are approximately equal to N, then a multiplication of two rank-4 tensors with two reductions costs $\mathcal{O}\left(N^6\right)$. The number of operations for the multiplication is fixed, although it is possible to optimize the running time by considering data locality and other machine-dependent factors.

The k-index transformation is trickier. Consider as an example the formation of a rank-4 tensor,

$$v_{mnpq} = g_{hijk} z^1_{mh} z^2_{ni} z^3_{pj} z^4_{qk}. \tag{6}$$

Eqn. (6) is itself a product of four tensors with one reduction each, so it is possible to compute it by calling a product function repeatedly. This is in fact the most efficient method in terms of number of operations. Inverting the order of multiplication and collecting intermediate terms gives

$$v_{mnpq} = z^4_{qk}\left(z^3_{pj}\left(z^2_{ni}\left(z^1_{mh} g_{hijk}\right)\right)\right). \tag{7}$$

Evaluating this expression requires

$$\# = 2MK\left(HIJ + IJN + JNP + NPQ\right) \tag{8}$$

multiplication and division operations. When all dimensions are similar, we write this as $8N^5$ and denote the computational complexity as $\mathcal{O}\left(N^5\right)$.

Remark 2.1 This bookkeeping does not account for the number of operations needed to form the core tensor in the first place. In practice, the core tensor may require on-the-fly computation, or perhaps even disk access. The level of ease or difficulty of creating pieces of the core tensor can greatly affect the effectiveness of our approach.

Remark 2.2 The fact that M—the left-most index in the result—is on the outside of the parentheses in Eqn. (8) will be very important in the parallel version of the algorithm. It may seem at first that the order of evaluation is non-intuitive, but it will allow us to divide the problem for parallelization without increasing the computational complexity beyond that of the final product.

The problem with a naïve product in Eqn. (7) is that H, I, J, and K could be very large, so the intermediate tensors could be too large to store on a single processor. We assume in the next section that any processor can store small pieces of each tensor, but not the whole thing. One way to decrease the storage requirements is to fuse the loops that produce intermediate tensors. Loop fusion is a common element in the TCE, which has generated significant interest in optimizing the levels of fusion for various types of tensor products [3, 14].

By fusing two loops over the product in Eqn. (3), we reduce the intermediate storage requirements by two dimensions, so if a tensor has rank k, then the intermediate tensors have rank $k-2$. A rank-3 tensor only requires the storage of a few vectors, a rank 4 tensor only requires the storage of a few matrices, etc. While it is possible to fuse loops more deeply and and further reduce memory requirements, reducing by two is sufficient for most applications.

Fig. 1 illustrates the concept applied to Eqn. (7). An unfused version requires the storage of rank-4 intermediate tensors, while the fused version requires the storage only of matrices. Both have the same number of operations. This is just an example to illustrate the concept of loop fusion—the actual code is much more general (and much less legible) because it works on tensors of any rank.

The notation of Fig. 1 is a form of compressed-loop pseudocode, where the order of the loop variables in the `for` declaration indicates the order of nesting. Assigning a scalar to a tensor quantity, as in `T1=0`, implicitly assigns the scalar to the whole tensor element-wise.

2.2 Parallelization

As mentioned in Sec. 1.2, we do not convert the tensors internally into matrices. We instead reduce the problem in concept to a string of scalar loops as in Fig. 1. (To reiterate—internally, we collect many of the scalar loops and use BLAS functions on subsets of the problem when possible. The reader loses no understanding, however, by perceiving our algorithms on a scalar level.) What this means in practice is that we break the problem into a set of serial multiplications on individual processors, rather than use an intrinsically parallel algorithm with all its attendant message passing and disk access.

A few key assumptions are necessary to compute Eqn. (4) in parallel. For the rest of this paper, a *row* of a rank-k tensor is a rank-$(k-1)$ tensor formed by fixing the first index of the original. For example, u_{3jk} is a row of the rank-3 tensor, u_{ijk}. Our primary assumptions are the following:

```
T1=0; T2=0; T3=0; v=0;
for m,k,j,i,h to M,K,J,I,H
  T1(m,k,j,i) += z1(m,h)*g(h,i,j,k)
end h,i,j,k,m

for n,m,k,j,i to N,M,K,J,I
  T2(n,m,k,j) += z2(n,i)*T1(m,k,j,i)
end i,j,k,m,n

for p,n,m,k,j to P,N,M,K,J
  T3(p,n,m,k) += z3(p,j)*T2(n,m,k,j)
end i,j,k,m,n

for q,p,n,m,k to Q,P,N,M,K
  v(m,n,p,q) += z4(q,k)*T3(p,n,m,k)
end k,m,n,p,q
```

```
v=0
for m,k to M,K
  T1 = 0
  for j,i,h to J,I,H
    T1(j,i) += z1(m,h)*g(h,i,j,k)
  end h,i,j

  T2=0
  for n,j,i to N,J,I
    T2(n,j) += z2(n,i)*T1(j,i)
  end i,j,n

  T3=0
  for p,n,j to P,N,J
    T3(p,n) += z3(p,j)*T2(n,j)
  end i,j,n

  for q,p,n to N,P,Q
    v(m,n,p,q) += z4(q,k)*T3(p,n)
  end n,p,q
end K,M
```

Figure 1: Loops used to calculate Eqn. (7): unfused (left) and fused (right)

1. All tensors can be formed one row at a time, either from the k-index transformation, as in Eqn. (3), or else by loading a row from disk.
2. The u and v are similar to each other in dimension and size.
3. The dimensions of the first indices of u and v are not small compared to any of the other dimensions.
4. Each processor has enough memory to store one row of u, v, and w simultaneously.

The task is to distribute the rows of u and v (from Eqn. (4)) among the processors in an efficient way. We consider two techniques: *simultaneous distribution* and *scheduling*. The former is useful only for very small tensors or when interprocessor communication is impossible for long lengths of time. The latter is much more appropriate for large tensors on most modern clusters.

Both are implemented in the C++ classes described in Sec. 3, and both use the same basic concept of splitting the problem into a set of row-row multiplications. The difference between the two parallelization techniques is in the way we allocate combinations of rows of u and v.

Before discussing specifics, we should ask whether the overall approach is a reasonable. Let A_1 and B_1 be the number of rows of u and v respectively in Eqn. (4), and let P be the number of processors. Multiplying u and v requires the multiplication of A_1B_1 individual rows. In the allocation of the workload, we must be careful not to ask for too many redundant formations of u and v.

Consider, for example, the multiplication of two rank-4 tensors with two reductions. Let all the dimensions in the system be roughly the same, and call that

dimension N. On a single processor with sufficient memory, the multiplication of u and v requires the following tasks:

- Form all of u and v: $16N^5$. Multiply them: $2N^6$.

The total flop count is $2N^6 + 16N^5$. Now consider the same operation on N^2 processors. Each processor forms one row of u and one row of v using the 4-index transformation and then multiplies them. This requires redundant formations:

- Form each row of u and each row of v N times: $16N^6$. Multiply all rows: $2N^6$.

The total flop count is $18N^6$. With fewer processors, the operations count will be smaller only if the allocation is rational in avoiding extra formations. (Now it is clear why moving the row index on the outside of the parentheses of Eqn. (8) is critical to the success of the current approach: Forming a single row costs $8N^4$, whereas it would cost $8N^5$ if we evaluated Eqn. (7) in another order.)

Our approach is therefore reasonable if done properly. It adds some extra calculations but the computational complexity is still of the same order. We note, however, that the utility of breaking up the problem into smaller and smaller pieces is limited due to the increasing number of redundant formations. With very large problems, it would be necessary to interleave disk access operations with formations rather than using exclusive on-the-fly formation.

The following two sections discuss two methods for allocating the workload.

2.3 Simultaneous distribution

In this technique, processors work in isolation from other processors. Each processor computes its section of the workload based on its identifier, the size of the multiplication, and the total number of processors in the system. The algorithm for determining the work that processor n performs switches on two cases:

- $P < A_1$:

In this case, each processor has to multiply one or more rows of u by *all* rows of v. If $P = A_1/2$, then each processor receives two rows of u; if $P = A_1/3$, then each receives three rows, etc. The difficulty is when P is not an integer division of A_1, and there are left-over rows to distribute.

Let $\lfloor \cdot \rfloor$ denote the floor operation. If processor n receives I rows of u, starting at row i (zero-indexed), then i and I are given by

$$i = n \lfloor A_1/P \rfloor + \min \{n, (A_1 \bmod P)\}, \text{ and} \tag{9}$$

$$I = \lfloor A_1/P \rfloor + \begin{cases} 1 & n < (A_1 \bmod P) \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

When the number of processors grows larger than A_1, the distribution becomes more complicated.

- $P \geq A_1$:

In this case, each processor works on exactly one row of u, and all the processors assigned to a given row then split up v among themselves. Processor n is assigned to the following row, i, of u:

$$i = \begin{cases} n/(d+1) & n < m(d+1) \\ m + \dfrac{n - m(d+1)}{d} & n \geq m(d+1) \end{cases} \tag{11}$$

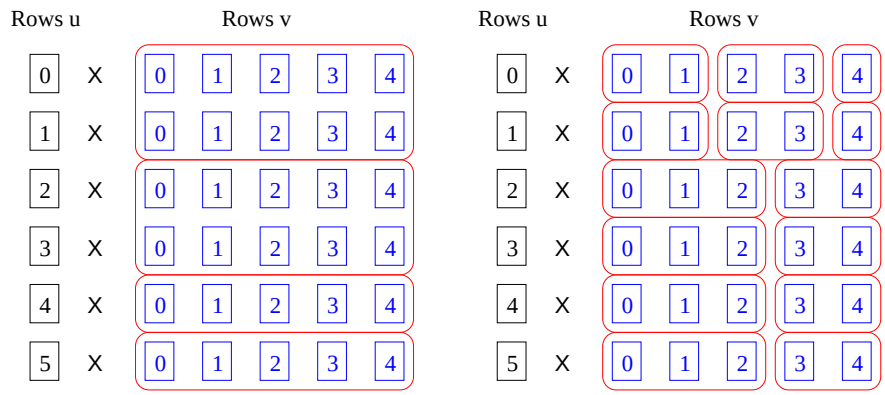

Figure 2: Distribution for $A_1 = 6$, $B_1 = 5$, with $P = 4$ (left) and $P = 14$ (right)

where $m = (P \bmod A_1)$, and $d = \lfloor P/A_1 \rfloor$.

To determine what piece of v processor n receives, define two quantities: q and Q. These are similar to n and P, except that they are local on a single row of u. Specifically, Q is the number of processors on current row,

$$Q = \begin{cases} d+1 & n < m(d+1) \\ d & n \geq m(d+1) \end{cases} . \tag{12}$$

while q is the rank of processor n among the processors assigned to the same processor:

$$q = \begin{cases} n \bmod (d+1) & n < m(d+1) \\ [n - m(d+1)] \bmod d & n \geq m(d+1) \end{cases} \tag{13}$$

In a formulation similar to Eqns. (9) and (10) above, let j be the first row of v that processor n operates on, and let J be the number of rows of v. The formulas are

$$j = q \lfloor B_1/Q \rfloor + \min \{q, (B_1 \bmod Q)\} \text{, and} \tag{14}$$

$$J = \lfloor B_1/Q \rfloor + \begin{cases} 1 & q < (B_1 \bmod Q) \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

For a visual depiction of the distribution, consider the plots in Fig. 2. These show the numbered rows of u and v in columns and rows respectively. The rounded rectangles represent the assignment area for a single processor. In the numbering scheme described by Eqns. (9-15), the processor numbers start at zero in the top-left and increase left-to-right, top-to-bottom.

In summary, this technique has several advantages and disadvantages. The primary advantages are that it incurs very little redundancy and has essentially no communication. On the downside, the technique is difficult to adapt to machines with different processor speeds or non-exclusive nodes, and it works best only if the number of processors either divides or is divided by the number of rows of u (i.e., $P = \{\ldots, A_1/3, A_1/2, A_1, 2A_1, 3A_1 \ldots\}$).

A better way to parallelize the algorithm on machines with inter-process communication is to schedule pieces of the multiplication dynamically. This is more

robust, but it runs the risk of inducing more redundant formations than would otherwise be necessary. As a partial remedy, we use a heuristic to assign new rows to processors depending on their previous states.

2.4 Scheduling

The basic mechanism for scheduling is simple: each time a processor requests more work, it receives from the scheduler some row i of u and some row j of v: (i, j). The processor then forms the ith row of u (if necessary) and the jth row of v (if necessary) either using the k-index transformation or loading from disk, and then multiplies the two rows. The scheduler should assign row pairs so as to avoid unnecessary formations.

The exact means through which we perform the scheduling may change—what is important is the procedure used to assign pieces of the task. For now, assume that one processor performs scheduling exclusively, while all others perform the formation and multiplication. Future versions of the code may allow the scheduler to do computational tasks as well, or they may even dispense with a root node and instead let each processor schedule itself according to global data. Such a change would be transparent to the user.

Let P be the number of processors doing actual computation. Let $\mathbf{s}$ be a table of length P that lists an approximate speed for each processor relative to others. For example, if there are five processors, and the first three are twice as fast as the others, then the table could be any positive multiple of $\mathbf{s} = \begin{pmatrix} 2 & 2 & 2 & 1 & 1 \end{pmatrix}$. Denote the kth element of $\mathbf{s}$ as s_k.

Two other tables, $\mathcal{U}$ and $\mathcal{V}$, of length A_1 and B_1 respectively, track the amount of work left to do on each row of u and v. They essentially describe how "crowded" the rows of u and v are, where a processor's contribution to the crowd depends on its speed.

The following two paragraphs describe the construction of $\mathcal{U}$. The construction of $\mathcal{V}$ is analogous.

Each row of u corresponds initially to B_1 rows of v: $(i, 1)$, $(i, 2)$, ... , (i, B_1). We let m_i be the number of unassigned rows of v in row i of u, so $m_i = B_1$ initially for each i, and we decrement it every time a processor receives a pair of the form $(i, *)$. Let $\mathcal{I}_i$ be a set containing IDs for all processors that are currently working on a pair of the form $(i, *)$. ($\mathcal{I}_i$ could be empty.)

Define the ith element of $\mathcal{U}$ to be

$$\mathcal{U}_i = \frac{m_i}{1 + \sum_{k \in \mathcal{I}_i} s_k}. \tag{16}$$

Initially, all elements of $\mathcal{U}$ are equal to B_1. The value of $\mathcal{U}_i$ falls as processors are assigned rows i of u, until eventually $\mathcal{U}$ is all zeros. The higher the value of $\mathcal{U}_i$, the less crowded that particular row of u, and the more attractive that row is for fresh assignment.

Consider a processor that just finished (i_0, j_0) and is requesting more work from the scheduler. The scheduler will look at the remaining work available and send back a new row-pair, say (i, j), to the processor for formation/multiplication.

Our rule for assigning a new row-pair is

1. If $U_{i_0} > 0$, then we may stay on this row of u. Perform the following:
   - Let j be such that $\mathcal{V}_j = \max \{\mathcal{V}_j \in \mathcal{V} \mid \text{pair } (i_0, j) \text{ is available}\}$.

- Assign the processor to the pair (i_0, j).
- Update $\mathcal{U}_{i_0}$, $\mathcal{V}_{j_0}$ and $\mathcal{V}_j$.

2. Otherwise, if row j_0 is available on some other row i of u, then
   - Let i be such that $\mathcal{U}_i = \max\{\mathcal{U}_i \in \mathcal{U} \mid \text{ pair } (i, j_0) \text{ is available}\}$.
   - Assign the processor to the pair (i, j_0).
   - Update $\mathcal{U}_{i_0}$, $\mathcal{U}_i$, and $\mathcal{V}_{j_0}$.
3. Otherwise,
   - Let i be such that $\mathcal{U}_i = \max \mathcal{U}$
   - Let j be such that $\mathcal{V}_j = \max\{\mathcal{V}_j \in \mathcal{V} \mid \text{pair } (i, j) \text{ is available}\}$.
   - Assign the processor to the pair (i, j) .
   - Update $\mathcal{U}_{i_0}$, $\mathcal{V}_{j_0}$, $\mathcal{U}_i$, and $\mathcal{V}_j$,

In other words, we try to prevent the processor from having to form a new row of u. If that is unavoidable, we assign it to a new u but try to prevent it from having to form a new row of v. If that is unavoidable, we assign it to new row of u and v. Whenever we have to assign a new row, we choose one that is the "least crowded" according to the $\mathcal{U}$ and $\mathcal{V}$.

Two special cases apply. If this is the first request from a particular processor, then obviously (i_0, j_0) makes no sense, so skip to step 3. If all combinations are exhausted (i.e., $\mathcal{U} \equiv 0$), then return an exit code.

We test this procedure using a stochastic simulation written in `MATLAB`. The simulation models a parallel computer as a structure array of "processors". Each processor has a speed—the mean return time it takes for the simulated processor to multiply two rows—and a standard deviation.

The simulation runs over all A_1B_1 row combinations. Each time through the loop, it determines which processor will return next. It assigns a new row combination to that processor using our distribution procedure, and then it computes a new return time by drawing on a normal distribution with the processor's mean return time and standard deviation. During this process, we track total time and total number of formations, both of which we may then compare to ideal computation cases.

To illustrate, we consider a simulated machine with two different types of processors. We assume that half the processors have speed 1 and the other half have speed 1/3. The mean time it takes a processor to multiply two rows is the reciprocal of its speed, and the standard deviation is 10% of that. We model formations of individual rows via the k-index formation for two rank-4 tensors, so it takes 4 times as long to form a single row as it does to multiply two rows.

In order to quantify the results in an intelligible way, we compare to an "ideal" distribution scheme. The ideal number of formations occurs when no processor ever has to form two rows after the initial assignment: $A_1B_1 + P$ formations. Determining an ideal computation time is more complicated due to discretization effects. We do not calculate the absolute, best possible running time; rather we estimate it based on the following derivation.

Let f be the cost of formation relative to the cost of multiplication so the cost of forming two rows and multiplying them is $1 + 2f$. Consider a setup like the one described above, where the system has P processors, some of which have speed 1, and some of which have speed $1/c$. Let χ be the fraction of processors with baseline speed 1. Each processor receives an initial assignment of two rows, which leaves

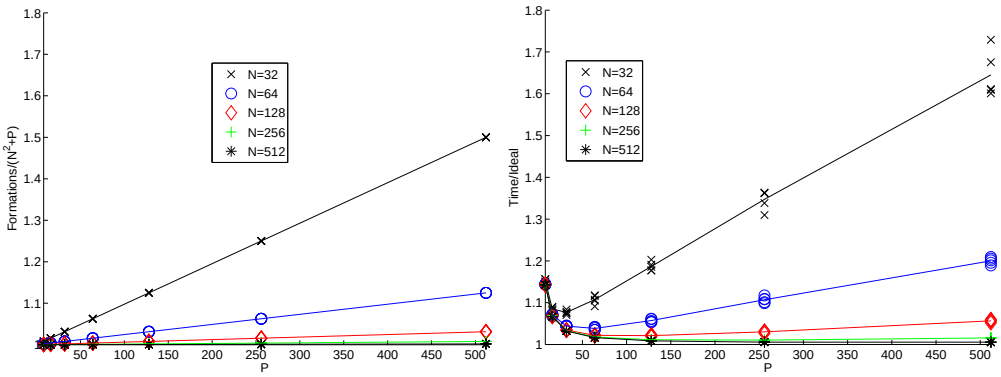

Figure 3: Simulated performance of the scheduling scheme: Number of formations vs P (left) and Time vs P (right)

$A_1B_1 - P$ row combinations remaining. Let α be the fraction of those remaining combinations assigned to the baseline processors.

Disregarding discretization effects, the ideal running time should occur when the baseline processors and the off-speed processors both finish at the same time. That is to say, α must satisfy,

$$(1+2f) + \alpha\,(1+f)\,\frac{(A_1B_1 - P)}{\chi P} = c\,(1+2f) + (1-\alpha)\,\frac{c\,(1+f)\,(A_1B_1 - P)}{P\,(1-\chi)}. \quad (17)$$

Solving for α and substituting into the left-hand side gives the following estimate of "ideal" running time:

$$T_{\text{ideal}} = (1+2f) + \frac{(1+2f)\,(c-1)\,(1-\chi) + c\,(1+f)\,(A_1B_1 - P)\,/P}{1+\chi\,(c-1)}. \quad (18)$$

In our configuration, $c = 3$, $f = 4$, and $\chi = 1/2$.

By any measure, the simulation indicates that our scheduling procedure is robust and avoids most unnecessary row formations. The curves in Fig. 3 illustrate the performance of the distribution procedure. To generate Fig. 3, we choose $N = A_1 = B_1 \in \{32,\ 64,\ 128,\ 256,\ 512\}$, $P \in \{8,\ 16,\ 32,\ 64,\ 128,\ 256,\ 512\}$ and run the simulation five times for each combination of P and A_1. The lines on the plot pass through the average of all five points at each P.

The number of formations and the running time both deviate most from the ideal when the problem size is small relative to the number of processors. Results are also more volatile in those cases. Randomness and discretization probably account for this, as such effects are more significant with smaller problems than with larger problems.

3 Implementation

We implement our algorithms in a set of C++ classes. The following sections describe pieces of the classes and illustrate ways to use them to perform some common tasks. The discussion is split into descriptions of serial and parallel code, although this distinction is sometimes artificial; many of the same basic concepts apply in both cases.

Both the serial and parallel classes contain a significant number of utility functions. They can load and store tensors or characteristic matrices in text or binary formats with a variety of choices of notation. (For example, they can print in a binary, compact text, or a more human-readable text form.) The parallel class `bTensorPar` contains broadcast functions for the whole tensor and for the characteristic matrices. There are access functions for elements such as rank, dimensionality, number of elements, MPI data, formation flags, etc. There are operator overloads for element-wise addition, subtraction, multiplication, and division. The programming interface for the classes may grow more complex as the classes mature, but most changes should be additions, not deprecations.

3.1 Serial code description

The serial code is in a C++ class called `bTensor` and a child of that class called `bTensorCharm`. The parent class contains all the usual functions for loading, storing, and printing tensors, as well as a product function, `prod`, that accepts references to two tensors and overwrites the current tensor with their product. It also defines all the usual access functions and arithmetic operator overloads.

The child class, `bTensorCharm`, is an extension of the parent. It contains all the functions and data of the parent class and in addition stores characteristic matrices as private sub-tensors. The core tensor is a user-defined pointer to a function that accepts an index vector as input and returns a double corresponding to that index. Thus, the `bTensorCharm` class does not actually store elements of a core tensor, although the user can mimic storage by using static data in the core tensor function pointer. The class also contains public functions for loading and printing the characteristic matrices, as well as for forming the tensor from the k-index transformation.

The formation function `formit` accepts an offset as an argument, so we may form only certain "rows", i.e. sections with a fixed first index. Consider a tensor a_{ijk} with dimensions $\{I, J, K\} = \{5, 3, 4\}$. To access only the third and fourth rows, perform the following steps:

1. Instantiate a tensor, b_{mnp}, where $\{M, N, P\} = \{2, 3, 4\}$.
2. Load the characteristic matrices of a into b.
3. Set the function pointer in b to define the core tensor.
4. Call `formit` with an offset of 2.

The $2 \times 3 \times 4$ tensor b now contains the third and fourth rows of the $5 \times 3 \times 4$ tensor a. This ability to peel off individual rows of a tensor becomes important in the parallel version. Sec. 6 illustrates a sample usage of the `bTensorCharm` class without offset.

3.2 Parallel code description

The parallel code is an extension of the serial code. We define a new class called `bTensorPar` that inherits most of the data and functions from `bTensorCharm`. It also contains data to represent the MPI state (process ID, communicator, etc.), and functions for broadcasting, scattering, and collecting either the whole tensor or the characteristic matrix.

The general procedure for multiplying two tensors is the following:

1. Get MPI information.
2. Instantiate tensors of the proper size.
3. If applicable, load the characteristic matrices and broadcast them.
4. Call the function `parProd` with desired parameters.

Most of the work is in the `parProd` function. On the root node, this kicks off a scheduling algorithm. On all other nodes, it queries the scheduler for rows of u and v to multiply, then it forms, loads, or otherwise creates rows, multiplies them, and stores the result according to options passed in.

This procedure is the scheduling method from Sec. 2.4. See Sec. 7 for an example of working C++ code that implements this algorithm.

The programmer also has access to the scheduler directly through the public functions `getPart()` and `setPart()`. `getPart()` provides four integer values to the programmer. The first two are the rows of u and v to form respectively. The next two give information about whether those rows have already appeared in the scheduler. For example, the third (resp. fourth) element is either 0, 1, or 2. Those are codes for the following:

0 The scheduler has not assigned this row of u (resp. v) to any other processor before.

1 The scheduler has assigned this row of u (resp. v) to another processor, but that processor has not yet returned for another row pair.

2 The scheduler has assigned this row of u (resp. v) to another processor, and that processor has completed its assignment and returned for more.

The programmer may use this information to optimize the parallel computation using formation procedures other than file loading of k-index transformations.

Instead of scheduling, we could use the all-at-once function and create large pieces of the tensors at once instead of row-by-row. This is implemented through the function `splitMult()`, which, when called, gives each processor a list of row-row combinations using the formulas in Sec. 2.3. The practicality of such an approach is limited by memory.

4 Results

We test the parallel code on the `flash` platform at Los Alamos National Laboratory. The segment used for the tests has 300 nodes on a Myrinet LANAI interconnect. Each node comprises two single-core, AMD Opteron processors running at 2.0 GHz, with 8 GB of memory shared asymmetrically between the processors. All processors have 1 MB of cache. This system offers dedicated nodes, but it also

enforces strict wall-clock limits that prevent us from evaluating completely out-of-core tensors (e.g., $512 \times 512 \times 512 \times 512$ and larger, with 256 processors and with memory as listed).

We experiment with the multiplication of two rank-4 tensors with two reductions. The dimensions are always identical across indices: $N \times N \times N \times N$, where N varies from 16 to 128. The number of processors varies from 16 to 256. For the remainder of this section, we assume that any two tensors being multiplied have identical dimensions. The characteristic matrices of the tensors are $N \times N$, uniformly-distributed random numbers in the interval $[-1.05, 1.05]$. The core tensors are functions that return a 1.0.

The growth rate of the k-index transformation for a single row is $8N^4$, while the tensors grow only at a rate of N^4, so for every platform there exists a point when disk access is faster than on-the-fly formation. For completeness, experiments that combine disk access and one-the-fly formation have been conducted. In those experiments, the processors do one of three things upon the reception of a new row assignment:

- If no other processor has received this row, form the row and save the result to a binary file so that other processors can use it later.
- If another processor has received this row but has not returned, form the row but do not save the result to disk.
- If another processor has received this row and has returned, load the row from a binary file.

These experiments have shown that when disk access is available, the time to compute the product varies widely and is generally slower than forming the rows as needed. With larger tensors, however, the tests consistently run faster with disk access. We can, for example, multiply two $N = 256$ tensors using file access in about 8000 seconds, whereas on-the-fly formation runs past wall-clock time limits. (The wall-clock limit is four hours on the test platform.)

Thus, while it might be faster in many situations to load an already-computed row from disk rather than generate it anew from characteristic matrices, the performance boost from file loading depends on the speed of the file system. For the rest of the tests in this section, we form individual rows of the tensors on-the-fly using the k-index transformation and do not save pre-computed rows for future reference. The results below concentrate analysis on the algorithm itself while attempting to eliminate other system-dependent properties.

Fig. 4 shows computation time as a function of N. It contains four parts: time spent on formation, time spent on multiplication, total time, and fraction of total time spent on multiplication. Each line represents a constant number of processors.

The reported times are the *maximum* times that it takes for any one processor to complete its piece of the task, and the total time is the sum of the two. The actual computation time might be slightly higher or lower depending on other, non-timed operations and general distribution of the workload.

Consider, for example, a run where one processor spends 5 seconds forming rows but only 2 seconds multiplying them, and another processor spends 4 seconds forming rows and 3 seconds multiplying them. Then the reported values of time-to-form, time-to-multiply, and total time satisfy respectively $T_{\text{form}} \geq 5$, $T_{\text{mult}} \geq 3$, and $T_{\text{tot}} \geq 8$, although the total computation time may be no longer than 7 seconds. Effects like these should be insignificant for large tensors.

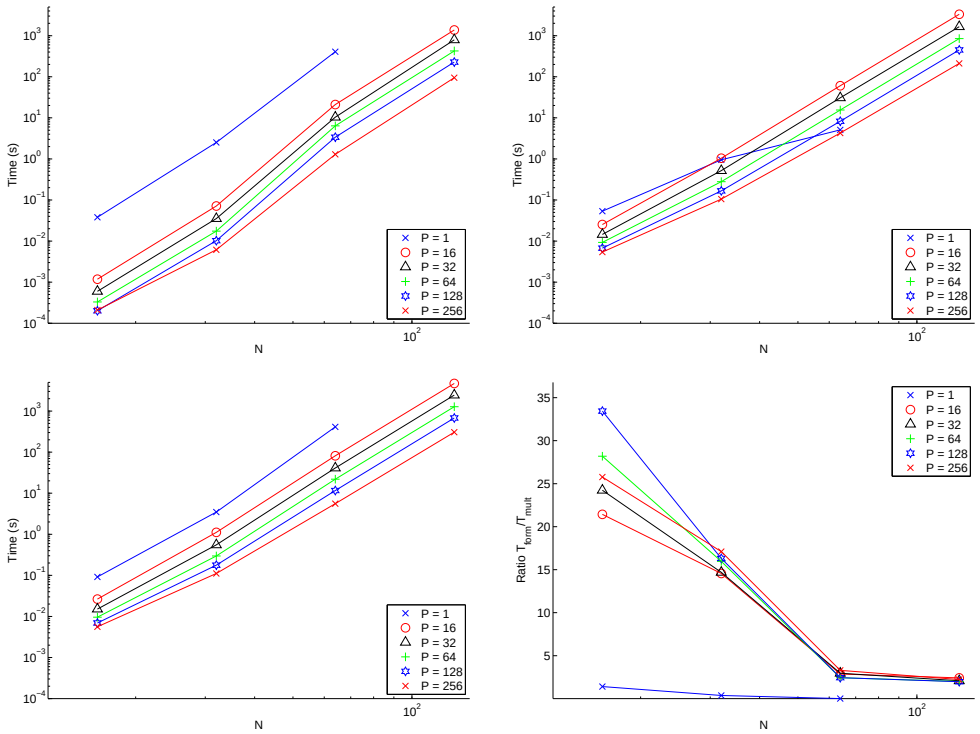

Figure 4: Computation times and ratio vs. N for different numbers of processors; left to right from top left: T_{mult}, T_{form}, T_{total}, and $T_{\text{form}}/T_{\text{mult}}$

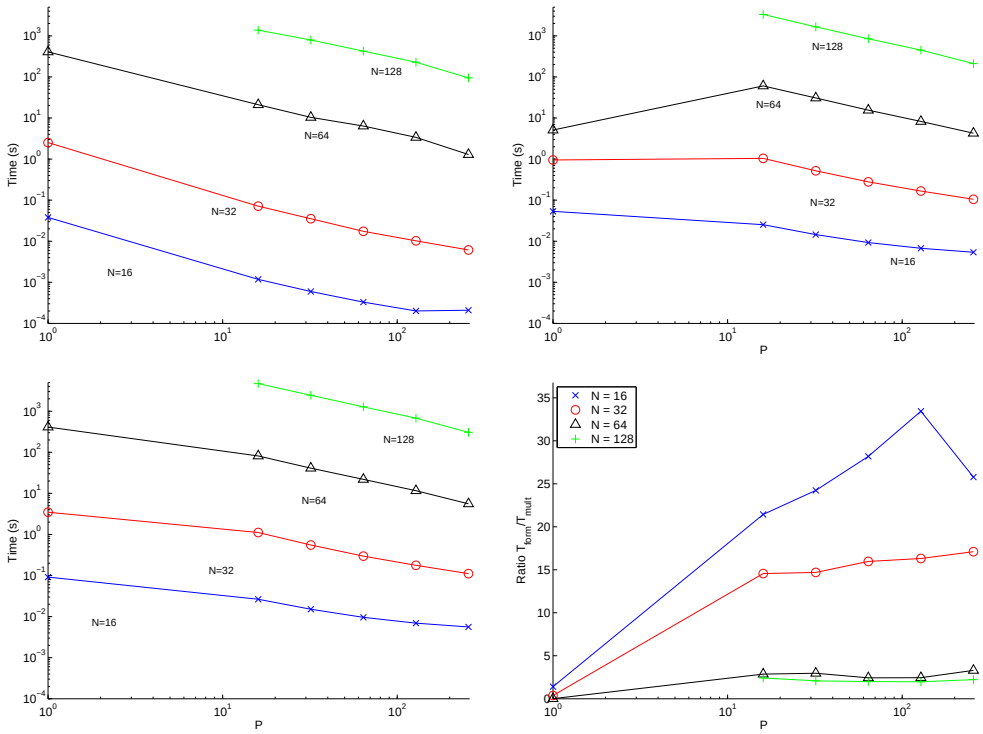

Figure 5: Computation times and ratio vs. P for different N; left to right from top left: T_{mult}, T_{form}, T_{total}, and $T_{\text{form}}/T_{\text{mult}}$

Fig. 5 shows vertical slices of Fig. 4. Each curve represents a different value of N and decreases according to an increasing number of processors.

4.1 Scaling results

We use several methods to measure scaling efficiency. Customarily, the term "efficiency" refers to speedup relative to a serial algorithm. We use the serial tensor classes to produce serial timings by having them form all of u and v from characteristic matrices, then multiply them and place the result in w. Other, less generic, algorithms may perform slightly better than the current serial code, but the current ones are reasonably fast and provide a good baseline for scaling calculations.

Unfortunately, memory and run-time limits on the parallel machine preclude serial timings for $N > 64$. We still compute scalings for $N = 128$, but they are relative to smaller parallel runs, not serial runs. Three tables below outline some basic properties of how well the algorithms scale. All times are total times (sum of multiplication and formation times), and we denote by T_P the time required to complete the operation using P processors.

Tab. 1 is "classical" efficiency: the time required to complete the problem using

Table 1: Efficiency relative to serial

N	$T_1/16t_{16}$	$T_1/32T_{32}$	$T_1/64T_{64}$	$T_1/128T_{128}$	$T_1/256T_{256}$
16	0.22	0.19	0.15	0.10	0.06
32	0.20	0.20	0.18	0.15	0.12
64	0.32	0.31	0.29	0.28	0.29

Table 2: Efficiency relative to $P = 16$

N	$T_{16}/2T_{32}$	$T_{16}/4T_{64}$	$T_{16}/8T_{128}$	$T_{16}/16T_{256}$
16	0.88	0.69	0.48	0.30
32	1.00	0.94	0.78	0.62
64	0.99	0.93	0.88	0.92
128	0.96	0.92	0.87	0.96

one processor, divided by the time it takes to complete using P processors times P. In other words, efficiency, E is defined as $E = T_1/PT_P$. We do not have a serial timing available for $N = 128$, so this is a shortened table.

Tab. 2 illustrates the same concept, except that the baseline is the $P = 16$ case instead of the serial algorithm. This is a better way to measure efficiency because the serial algorithm is impractical for large tensors due to memory constraints. We should therefore compute efficiency relative to algorithms that will actually run on the tensors of interest. Recall, too, that the memory requirements for the parallel algorithm are independent of the number of processors available.

Tab. 3 shows how the run time changes with *each* step in the number of processors. Increasing the number of processors from P_1 to a larger number P_2 gives the step value $P_1T_{P_1}/P_2T_{P_2}$. We call this "pointwise" efficiency. It displays the evolution of the efficiency of the algorithm as P grows.

4.2 Issues in analysis

The key point to recognize in Tab. 1 is that there is a significant penalty in going from serial to parallel initially, so the parallel algorithm *with scheduling* is not suited to small tensors where $N < 64$. For large tensors, however, the efficiency does not degrade beyond the initial drop.

Table 3: Pointwise (step-to-step) efficiency

N	$T_1/16T_{16}$	$T_{16}/2T_{32}$	$T_{32}/2T_{64}$	$T_{64}/2T_{128}$	$T_{128}/2T_{256}$
16	0.22	0.88	0.79	0.69	0.62
32	0.20	1.00	0.93	0.84	0.80
64	0.32	0.99	0.94	0.94	1.05
128	–	0.96	0.96	0.94	1.11

The reader should keep two facts in mind: 1) It is difficult to obtain serial timings for large tensors due to memory requirements, and therefore, 2) comparison to a serial algorithm is not completely valid, since the real advantage of the parallel algorithm is in how much memory it saves. For example, a 512^4 multiplication problem in IEEE double precision would require 1.5 TB of memory to form and multiply in serial on a single processor. To handle this multiplication on a 512-processor machine using the parallel algorithm, each processor would only need access to 2 GB—not a small requirement, but not unreasonable.

The exact reason for the performance drop in the transition from serial to parallel is apparent in the two plots in the right of Fig. 4. The formation time is lower in the serial algorithm because it can form all of u and v at once without any redundant formations. The serial algorithm forms exactly $2N$ rows, whereas the parallel algorithm must form at least $N^2 + P$ if all processors are to be involved.

Redundant formations are of course the inescapable toll for moving out of core. We determined in Sec. 2.2, however, that the complexity of formation in a parallel algorithm is of the same order as the complexity of multiplication, namely $\mathcal{O}\left(N^6\right)$ for rank-4 tensors. Once the initial performance drop is paid for, the slopes of time vs. N do not bend upwards for sufficiently large tensors.

To ensure that the scheduler is indeed behaving as expected and is not driving the complexity higher, we record the number of row formations actually performed during each multiplication and divide them by $N^2 + P$. Fig. 6 shows that the number of formations never grows more than a couple of percent over the theoretical parallel minimum.

The run with 256 processors and $N = 16$ required fewer than $N^2 + P$ formations. Row formation for that experiment was so fast that 81 of the processors never received row-pair assignments. The job was already over before those processors had time to initialize their MPI neighborhoods and call the scheduler for an assignment. A barrier would have prevented this and ensured that every processor performed at least two row formations, but then the total clock time to complete the problem would have been longer. Alternately, we could have used the "simultaneous distribution" method instead of scheduling. (Using 256 processors on such a small problem really is overkill anyway.)

5 Discussion and future work

The tensor multiplication classes described in this paper seem to be very robust, efficient, and scalable. Using 256 processors of a machine that is not state-of-the-art, we can form two 2.7-million-element rank-4 tensors on-the-fly from four 16000-element core tensors and multiply them with two reductions—all in about 5 minutes with very little communication overhead. Simulations and practical experiments indicate that scaling results are good over the ranges of interest, and that the load balancing system is every efficient. The code has a straightforward C++ object-oriented interface.

We consider several areas for further investigation:

Core tensor At present, the "core tensor" for the k-index transformation is a function pointer that accepts an integer index and returns a value corresponding to that index. We use a dummy function for timing studies, but in several practical problems the generation of the core tensor may constitute the majority of the total work.

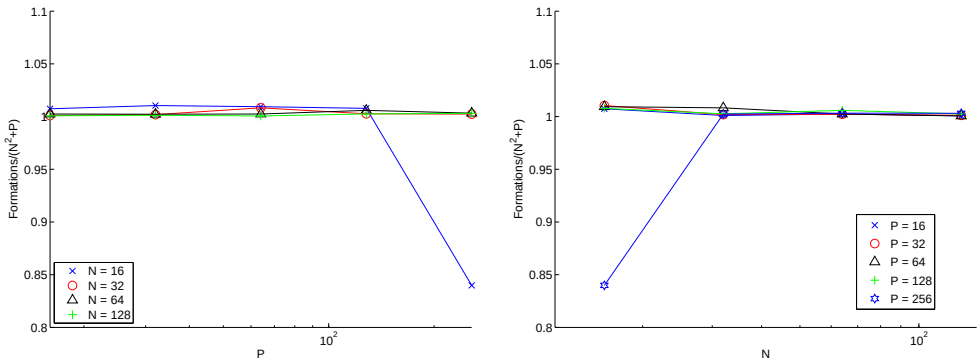

Figure 6: Total number of formations divided by $N^2 + P$ plotted vs. P (left) and N (right)

Users of the classes can always implement pre-computed core-tensors by setting the function pointer to reference static data, but this might be slow, depending on the application. As practicality demands, it may be worthwhile in the future to overload the formation function with different, application-dependent interfaces.

Strings of products Many applications require the multiplication of long strings of tensors with different numbers of reductions along the way [1]. One future goal is to provide a multiple-product public function. This would probably take the form of a more generic version of the k-index transformation, placed in the public interface to the `bTensorPar` class.

File I/O We provide several print and read functions for both binary and text files. In the timing studies, we discard row-row-multiplications as we compute them, but practical problems will inevitably include a significant amount of file access. The fraction of time spent on file access will depend on the application and the file system itself.

Other transformations There are other common tensor decompositions besides the k-index transformation. We would probably add these as different classes, as in [2].

In conclusion, the arbitrary-rank, arbitrary-reduction tensor classes presented in this paper have the potential for use in a wide array of scientific computing disciplines. They are works-in-progress, but already they show promise in both serial and parallel computations.

6 Appendix: Example use of the `bTensorCharm` class

The following code is an example of how to use the **`bTensorCharm`** class to load characteristic matrices, form tensors, and multiply them. All file names and dimensions are hard-coded to enhance readability.

```
#include "bTensorCharm.h"

// Dummy function for core tensor---all ones.
double ones(int *idx) { return 1.0; }

main()
{
   // Define dimensions of tensors.
   int dvecA[4] = {10,12,5,7};
   int dvecB[4] = {4,8,5,7};
   int dvecC[4] = {10,12,4,8};

   bTensorCharm A(4,dvecA);   // Instantiate.
   bTensorCharm B(4,dvecB);
   bTensorCharm C(4,dvecC);

   // Load characteristic matrices from files.
   A.setCharm("inputs/charm10-28.dat",0);
   A.setCharm("inputs/charm12-63.dat",1);
   A.setCharm("inputs/charm5-9.dat",2);
   A.setCharm("inputs/charm7-13.dat",3);
   B.setCharm("inputs/charm4-13.dat",0);
   B.setCharm("inputs/charm8-11.dat",1);
   B.setCharm("inputs/charm5-12.dat",2);
   B.setCharm("inputs/charm7-17.dat",3);

   // Set the core tensors with function "ones".
   A.setCore(ones);
   B.setCore(ones);

   // Form and multiply with two reductions.
   A.formit();
   B.formit();
   C.prod(A,B,2);

   // Print to file in nice format.
   C.print("outputs/test2.out",2);
}
```

7 Appendix: Example use of the bTensorPar class

The following code is an example of how to use the `bTensorPar` class to multiply two rank-4 tensors with two reductions. Most of the code is dedicated to file loading, MPI initialization, and storage of the output. The actual multiplication is in the function, `parProd()`, which calls the functions `getPart()`, and `setPart()` to schedule.

The code performs basically the same tasks as the serial program, but in parallel. The dimensions are different, too—the tensors are both $16 \times 17 \times 18 \times 19$ with the same dimensions in the columns of the characteristic matrices. The procedure is as follows:

All Initialize MPI data and instantiate a single row of `A` and `B`. Also instantiate `C` to hold the result of a single row-row multiplication.

Root Load the characteristic matrices and broadcast them.

Non-root Receive the broadcast.

All Call `parProd()`, with appropriate options.

All Close MPI.

At present, the root node performs only scheduling, so the tensors instantiated on that node are unused.

```
#include "bTensorPar.h"

// Dummy function for core tensor---all ones.
double ones(int *idx) { return 1.0; }

main(int argc, char **argv)
{
   int procs, myid;      // MPI data.
   const int root = 0;   // Root id.
   int dimA[4]={1,17,18,19};   // One row of each.
   int dimB[4]={1,17,18,19};
   int dimC[4]={1,17,1,17};

   // Initialize MPI data.
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   MPI_Comm_size(MPI_COMM_WORLD,&procs);

   // Instantiate single rows of A, B and C.
   bTensorPar A(4, dimA, MPI_COMM_WORLD);
   bTensorPar B(4, dimB, MPI_COMM_WORLD);
   bTensorPar C(4, dimC, MPI_COMM_WORLD);
   A.setCore(ones);
   B.setCore(ones);
```

```
   // Root loads and broadcasts characteristic matrices.
   if (myid==root){
      A.setCharm("inputs/charmA0016-0016.dat",0);
      A.setCharm("inputs/charmA0017-0017.dat",1);
      A.setCharm("inputs/charmA0018-0018.dat",2);
      A.setCharm("inputs/charmA0019-0019.dat",3);
      B.setCharm("inputs/charmB0016-0016.dat",0);
      B.setCharm("inputs/charmB0017-0017.dat",1);
      B.setCharm("inputs/charmB0018-0018.dat",2);
      B.setCharm("inputs/charmB0019-0019.dat",3);
   }
   A.broadCharm(root);
   B.broadCharm(root);

   // Multiply with strict formation. Print result out to file.
   char CHead [] ="outputs/res", CTail [] = ".dat";
   C.parProd(A,B,rowsA,rowsB,NULL,NULL,CHead,CTail,2);

   MPI_Finalize();
}
```

References

[1] A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, A. Hartono, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[2] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, December 2006.

[3] 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.-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(2):276–292, February 2005.

[4] M. Challacombe, E. Schwegler, and J. Almlöf. Fast assemby of the Coulomb matrix: A quantum chemical tree code. *Journal of Chemical Physics*, 104(12):4685–4697, 1996.

[5] T. J. Chung. *Applied Continuum Mechanics*. Cambridge Universiity Press, New York, 1996.

[6] X. Gao, S. K. Sahoo, Q. Lu, G. Baumgartner, C.-C. Lam, J. Ramanujam, and P. Sadayappan. Performance modeling and optimization of parallel out-of-core tensor contractions. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 266–276, New York, June 2005. ACM Press.

[7] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[8] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. E. Bernholdt, S. Hirata, C.-C. Lam, R. M. Pitzer, J. Ramanujam, and P. Sadayappan. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *Computational Science – ICCS 2005*, Lecture Notes in Computer Science, pages 155–164, Berlin/Heidelberg, May 2005. Springer.

[9] S. Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[10] R. W. Johnson, C.-H. Huang, and J. R. Johnson. Multilinear algebra and parallel programming. *The Journal of Supercomputing*, 5(2–3):189–217, 1991.

[11] 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. P. 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):260–283, 2000.

[12] R. Kobayashi and A. P. Rendell. A direct coupled cluster algorithm for massively parallel computers. *Chemical Physics Letters*, 265:1–11, 1997.

[13] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C.-C. Lam, P. Sadayappan, J. Ramanujam, D. E. Bernholdt, and V. Choppella. Data locality optimization for synthesis of efficient out-of-core algorithms. In *High-Performance Computing – HiPC 2003*, Lecture Notes in Computer Science, pages 406–417, Berlin/Heidelberg, February 2003. Springer.

[14] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C.-C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. *Journal of Parallel and Distributed Computing*, 66(5):659–673, 2006.

[15] B. Kumar, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming*, 4(4):275–289, 1995.

[16] A. J. McConnell. *Applications of Tensor Analysis*. Dover, New York, 1957.

[17] P. Piecuch, S. A. Kucharski, K. Kowalski, and M. Musial. Efficient computer implementation of the renormalized coupled-cluster methods: The R-CCSD[T], R-CCSD(T), CR-CCSD[T)], and CR-CCSD(T) approaches. *Computer Physics Communications*, 149:71–96, 2002.

[18] A. P. Rendell, T. J. Lee, and R. Lindh. Quantum chemistry on parallel computer architectures: Coupled-cluster theory applied to the bending potential of fulminic acid. *Chemical Physics Letters*, 194:84–94, 1992.

[19] J. R. Ruíz-Tolosa and E. Castillo. *From Vectors to Tensors*. Springer, Berlin, 2005.

[20] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. John A Montgomery. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993.

[21] G. E. Scuseria. The equilibrium structures of giant fullerenes: Faceted or spherical shape? An ab initio hartree-fock study of icosahedral C_{240} and C_{540}. *Chemical Physics Letters*, 243:193–198, 1995.