

A parallel implementation of tensor multiplication

(CSE260 research project)

Bryan Rasmussen *

1 December 2006

Abstract

This paper describes a series of generic, parallel routines for the multiplication of arbitrary-rank tensors with an arbitrary number of reductions. The routines can generate pieces of the tensors on-the-fly using index transformations from matrices to achieve significant savings in communications and storage. We test the routines on a series of rank-4 tensors generated from 5-column matrices, and we demonstrate that the algorithms can multiply large (17 million element) tensors with relatively low memory and communication requirements. Numerical experiments indicate that the algorithms scale very well in several measurements.

1 Introduction

The goal of this project is to write a parallel routine for multiplying tensors. To measure success of the project, it is also necessary to compute efficiency and scaling information.

Recall that a *tensor* is an extension of the concept of a linear operator to multilinear algebra [11]. Tensors have many applications in disparate areas such as fluid and solid mechanics, general relativity, and quantum mechanics [4, 8], and the motivating application for this project is in computational chemistry [3, 6, 9, 10, 12]. A full treatment of multilinear algebra and continuum mechanics is beyond the scope of this report, but we include a few simple definitions below for reference.

Given two vector spaces, V_m and W_n , over the same field, with dimension m and n respectively, there exists a natural operation called the *tensor product*, written $V_m \otimes W_n$. This is essentially the outer product of all vectors in the two spaces—and thus the dimension of the result is mn —but the product is taken while preserving a notion of order in resulting bases, and this ordering is what distinguishes the resulting space from a simple vector space. The elements of $V_m \otimes W_n$ are known as tensors. We can continue to take more and more products, say, $V_{m_1}^1 \otimes V_{m_2}^2 \otimes \dots \otimes V_{m_k}^k$, each time preserving the ordering. The number of products determines the *rank* of the tensors in the resulting product space.

Recall from elementary linear algebra that for a particular choice of basis vectors on a finite-dimensional space, every linear operator has a matrix representation.

*Los Alamos National Lab; bryanras@lanl.gov

If $A : V_m \rightarrow W_n$ is the operator, and $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_m\}$ is a basis for V_m , then the matrix representation is the familiar,

$$A = (A\mathbf{e}_1 \quad A\mathbf{e}_2 \quad \dots \quad A\mathbf{e}_m). \quad (1)$$

A similar representation is available for tensors. For a given choice of basis vectors, every tensor can be represented as a multi-dimensional box of numbers, where the number of dimensions corresponds to the rank of the tensor. This multi-dimensional box does not fully define the tensor, for it is still necessary to maintain the ordering mentioned above in order to describe how the box changes with changes in the basis. Nevertheless, the representation of tensors as multi-dimensional boxes is convenient, as we shall see below.

The elements of the box may change with a basis in one of two ways: *covariant* or *contravariant* transformations. Any given tensor may contain a mix of contravariant and covariant indices, which are typically represented with combinations of subscripts and superscripts in the notational convention described below. In the scope of this project, however, we assume the existence of a standard basis, so we do not concern ourselves with transformations of the tensor representation (which, by the way, is actually the most powerful aspect of multilinear algebra).

Therefore, for the rest of this report, we will think of tensors only as multi-dimensional boxes of numbers with an organizational structure. For us, a vector will be a rank-1 tensor, a matrix will be a rank-2 tensor, etc.

Notationally, we represent these boxes using indices, so, for example, we write a rank-3 tensor u with dimensions $\{N_i, N_j, N_k\} = \{3, 2, 5\}$ as u_{ijk} , where $i = \{1, 2, 3\}$; $j = \{1, 2\}$; and $k = \{1, 2, 3, 4, 5\}$. Tensors inherit most of the usual operations such as summation and scalar multiplication from the field. Additionally, one can multiply tensors in a variety of ways.

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

$$w_{ijkmn} = u_{ijk}v_{mn}. \quad (2)$$

If an expression contains one or more repeated indices, then it implies the Einstein summation convention, which means that we sum over the repeated (*i.e.*, “dummy”) indices. One repeated index in Eqn. (2) would result in a rank-3 tensor, for example

$$w_{ijm} = u_{ijk}v_{mk} = \sum_{k=1}^{N_k} u_{ijk}v_{mk}, \quad (3)$$

while two repeated indices would result in a rank-1 tensor, for example,

$$w_j = u_{ijk}v_{ik} = \sum_{i=1}^{N_i} \sum_{k=1}^{N_k} u_{ijk}v_{mk}. \quad (4)$$

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

To illustrate the subscript notation, consider column vectors \mathbf{u} , \mathbf{v} , and matrices A , B , all over \mathbb{R}^3 . The subscript notation for these entities is u_i , v_i , a_{ij} , and b_{ij} respectively, where $i, j \in \{1, 2, 3\}$.

It is possible to construct all manner of vector and matrix operations very compactly using subscripts. The inner product of \mathbf{u} and \mathbf{v} is simply $s = u_i v_i$. The outer product is $w_{ij} = u_i v_j$. These derive directly from the definition,

$$s = u_i v_i = \sum_{i=1}^3 u_i v_i = \mathbf{u}^T \mathbf{v}, \text{ and} \quad (5)$$

$$w_{ij} = u_i v_j = \begin{pmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \end{pmatrix} = \mathbf{u} \mathbf{v}^T. \quad (6)$$

Similarly, the notational equivalents of matrix-vector and matrix-matrix multiplication are $v_i = a_{ij} u_j$ and $c_{ik} = a_{ij} b_{jk}$ respectively:

$$v_i = a_{ij} u_j = \sum_{j=1}^3 a_{ij} v_j = \begin{pmatrix} a_{11} v_1 + a_{12} v_2 + a_{13} v_3 \\ a_{21} v_1 + a_{22} v_2 + a_{23} v_3 \\ a_{31} v_1 + a_{32} v_2 + a_{33} v_3 \end{pmatrix} = \mathbf{A} \mathbf{u}, \text{ and} \quad (7)$$

$$c_{ik} = a_{ij} b_{jk} = \sum_{j=1}^3 a_{ij} b_{jk} = \begin{pmatrix} \begin{pmatrix} a_{11} b_{11} + a_{12} b_{21} \\ + a_{13} b_{31} \end{pmatrix} & \begin{pmatrix} a_{11} b_{12} + a_{12} b_{22} \\ + a_{13} b_{32} \end{pmatrix} & \begin{pmatrix} a_{11} b_{13} + a_{12} b_{23} \\ + a_{13} b_{33} \end{pmatrix} \\ \begin{pmatrix} a_{21} b_{11} + a_{22} b_{21} \\ + a_{23} b_{31} \end{pmatrix} & \begin{pmatrix} a_{21} b_{12} + a_{22} b_{22} \\ + a_{23} b_{32} \end{pmatrix} & \begin{pmatrix} a_{21} b_{13} + a_{22} b_{23} \\ + a_{23} b_{33} \end{pmatrix} \\ \begin{pmatrix} a_{31} b_{11} + a_{32} b_{21} \\ + a_{33} b_{31} \end{pmatrix} & \begin{pmatrix} a_{31} b_{12} + a_{32} b_{22} \\ + a_{33} b_{32} \end{pmatrix} & \begin{pmatrix} a_{31} b_{13} + a_{32} b_{23} \\ + a_{33} b_{33} \end{pmatrix} \end{pmatrix} \\ = AB. \quad (8)$$

As an exercise, the reader might convert some other familiar linear algebraic operations such as $A^T B$, AB^T , and $\mathbf{u}^T A \mathbf{v}$, into the subscript form.

Finally, by way of simplification, we assume that every tensor has a very compact storage method called the *k-index transformation*. This means that a rank- k tensor, v , with a maximum dimension of N will be completely represented with an $N \times M$ "characteristic matrix", z (where M is undetermined). The transformation provides any element of the tensor using the formula,

$$v_{i_1 i_2 \dots i_k} = \sum_{j_1 j_2 \dots j_k=1}^M z_{i_1 j_1} z_{i_2 j_2} \dots z_{i_k j_k}. \quad (9)$$

This compact transformation becomes very important for minimizing storage and communication costs in the parallel algorithm. We note that computational cost of forming a single element of the tensor using Eqn. (9) is proportional to M^k , so the number of columns in the characteristic matrix can be a very important factor in performance of a parallel algorithm. For the remainder of this report, we assume relatively small dimensions, typically with $M = 5$.

2 Approach

In the current application, the tensors of interest are of rank-4, potentially with very large dimensions. The specific forms of interest are

$$\begin{aligned} w_{abij} &= u_{abef}v_{efij}, \\ w_{abij} &= u_{ijef}v_{abef}, \quad \text{and} \\ w_{abij} &= u_{afie}v_{bejf}. \end{aligned} \tag{10}$$

We concentrate on the following variation because it is simpler to program, and moreover the others are equivalent after a re-ordering:

$$w_{abij} = u_{abef}v_{ijef}. \tag{11}$$

In fact, both the serial and parallel codes are more generic than Eqn. (11). They actually operate on 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 in¹

$$w_{a_1 a_2 \dots a_m b_1 b_2 \dots b_n} = u_{a_1 a_2 \dots a_m c_1 c_2 \dots c_p} v_{b_1 b_2 \dots b_n c_1 c_2 \dots c_p}. \tag{12}$$

It turns out that code for solving Eqn. (12) is not that much more complicated than code for solving Eqn. (11).

2.1 Serial Algorithm

There are two common ways to write a serial tensor multiplication algorithm with reductions. The optimal strategy is to “unwrap” each tensor into matrix form, so for example, a rank-3 tensor with dimensions 2, 3, and 4 might be represented as a 2×12 matrix with 2×3 blocks. We may then represent various forms of tensor multiplication with block matrix multiplication, to which we may apply optimized functions from the Basic Linear Algebra Subprograms (BLAS) available on the computational platform.

We do not use this strategy here. Instead, we compute the tensor element-by-element in a serial loop, using the algorithm outlined in Fig. 1. (Throughout this report, we borrow the `MATLAB` colon notation to represent pieces of data objects.) Although it is not completely efficient, this algorithm allows us to extend the code to arbitrary-rank, arbitrary-reduction operations much more easily than if we had to unwrap tensors. Also, it allows us to study the internal operations in detail and split up the tensors more easily for parallelization.

The “index corresponding to i,j” lines in Fig. 1 mask a great deal of work. If not handled carefully, these lines could consume more computation than the element multiplications and summations themselves. We try to be as efficient as possible by storing cumulative products and minimizing the modulo and division operations involved in converting a single-scalar number into a list of indices. To illustrate, let x be a vector containing cumulative products of dimensions starting at the outer index and moving inward. If the dimensions of a rank-4 tensor are $\{3, 6, 4, 2\}$, then the corresponding vector is $x = (48 \ 8 \ 2 \ 1)$. Fig. 2 demonstrates the use of this vector to update indices continuously with a minimum of computation.

¹This is a different convention from the interim progress report.

```

// Define some constants.
N = product of dimensions of C
R = product of dimensions of repeated indices
rankA = rank A
rankB = rank B

// Rank of C will be Apart+Bpart.
Apart = rankA-r
Bpart = rankB-r

// Loop over all elements in C.
for ii = 0:N-1

    set idx = index of C corresponding to ii.

    // Get parts of the index of A and B.
    Aindex(0:(Apart-1)) = idx(0:(Apart-1))
    Bindex(0:(Bpart-1)) = idx(Apart:end)

    // Loop over all combinations of repeated indices.
    val = 0
    for jj = 0:R-1

        set idx2 = repeated indices corresponding to jj.

        // Fill out the rest of Aindex and Bindex
        Aindex(Apart:end) = idx2
        Bindex(Bpart:end) = idx2

        val = val + A(Aindex)*B(Bindex)

    end

    C(idx) = val

end

```

Figure 1: Serial algorithm for multiplying $C = AB$ with r reductions

```

// Initialization
dims = list of dimensions
x = cumulative product of dims (out-to-in)
initialize idx to -1 in all elements

// Loop (inner or outer)
for ii=0:product(idx)-1

    // Index computation
    kk = length(idx)-1
    while (kk > 0) AND ( (ii mod x(kk)) == 0)

        idx(kk) = (idx(kk)+1) mod dims(kk)
        kk = kk-1

    end

    ...

end

```

Figure 2: Index computation in a loop

The serial algorithm is written in C++. The code defines a new class called `bryTensor`, which stores the data and characteristic matrix in double-precision format in single arrays. It only allocates memory when necessary, such as when loading data from a file.

In addition to the usual functions for access, resizing, input/output, etc., the class contains a function, `formit`, that forms the entire tensor from a characteristic matrix and another function, `prod`, that multiplies two tensors with reductions in the form of Eqn. (12). The function `formit` accepts an offset as an argument, so we may form only certain “rows”, *i.e.* sections with a fixed first index.

For example, consider a tensor a_{ijk} with dimensions $i = 1, 2, 3, 4, 5$, $j = 1, 2, 3$, and $k = 1, 2, 3, 4$. To access only the third and fourth rows, perform the following steps:

1. Instantiate a tensor, b_{mnp} , where $m = 1, 2$, $n = 1, 2, 3$, and $p = 1, 2, 3, 4$.
2. Load the characteristic matrix of a into b .
3. 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.

The multiplication function, `prod`, is a straightforward serial loop, as discussed above. Its arguments are two references to (not necessarily distinct) tensors and a number of reductions. It overwrites the current object with the product of the tensors and allocates memory if necessary.

To test the code, we generate several sets of tensors and characteristic matrices of various sizes using random numbers. We test all the functions of the class, including file loads, multiplication, and formation from characteristic matrices. The results of the code agree with two other sources: 1) `MATLAB` routines written specifically for testing purposes, and 2) a publicly-available `MATLAB` tensor package [1].

2.2 Parallelization

The literature contains several attempts at parallelization of tensor multiplication since the late 1980s [5, 7]. One very sophisticated and difficult approach is to generate optimal code automatically for each problem at the time of solution [2]. This is clearly a complicated undertaking and is beyond the scope of the current project.

We seek instead to develop a strategy that trades optimality for robustness. A few key assumptions simplify the multiplication in Eqn. (12):

1. All tensors can be formed from the k -index transformation, as in Eqn. (9).
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.

With these assumptions in hand, the problem becomes one of properly distributing the rows of u and v among the processors. We choose distribute the rows of u first, then distribute the rows of v .

To wit, let N_u and N_v be the number of rows of u and v respectively, and let P be the number of processors. Then the distribution to processor n breaks down into two cases:

- $P < N_u$:

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

If processor n receives I rows of u , starting at row i (zero-indexed), then i and I are given by

$$i = n \lfloor N_u/P \rfloor + \min \{n, (N_u \bmod P)\}, \text{ and} \quad (13)$$

$$I = \lfloor N_u/P \rfloor + \begin{cases} 1 & n < (N_u \bmod P) \\ 0 & \text{otherwise} \end{cases} \quad (14)$$

For example, u has 10 rows divided among 4 processors, then processor 0 operates on rows $\{0, 1, 2\}$, while processor 2 operates on rows $\{6, 7\}$. Remember that all processors have to operate on all of v . When the number of processors grows larger than N_u , the distribution becomes more complicated.

- $P \geq N_u$:

In this case, each processor is assigned to 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 following row (i) of u :

$$i = \begin{cases} n/(d+1) & n < m(d+1) \\ m + \frac{n - m(d+1)}{d} & n \geq m(d+1) \end{cases} \quad (15)$$

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

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,

$$Q = \begin{cases} d+1 & n < m(d+1) \\ d & n \geq m(d+1) \end{cases} \quad (16)$$

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

$$q = \begin{cases} n \bmod (d+1) & n < m(d+1) \\ [n - m(d+1)] \bmod d & n \geq m(d+1) \end{cases} \quad (17)$$

In a formulation similar to Eqns. (13) and (14) 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

$$j = q \lfloor N_v/Q \rfloor + \min \{q, (N_v \bmod Q)\}, \text{ and} \quad (18)$$

$$J = \lfloor N_v/Q \rfloor + \begin{cases} 1 & q < (N_v \bmod Q) \\ 0 & \text{otherwise} \end{cases} \quad (19)$$

For a visual depiction of the distribution, consider the plots in Fig. 3. 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. (13-19), the processor numbers start at zero in the top-left and increase left-to-right, top-to-bottom. If it takes one unit of computational work to multiply one row of u by one row of v , then the total amount of work represented in Fig. 3 is $N_u N_v = 30$.

When $P = 4$, two processors are assigned one row of u each, and two are assigned two rows of u each. All processors must multiply their rows by all of v . The smallest workload on any given processor is 5, and the largest is 10. An ideal distribution would give a maximum workload of 8.

When $P = 14$, two rows of u get three processors each, and four rows get two processors each. The processors assigned to each row of u split up the workload either $\{2, 2, 1\}$ or $\{3, 2\}$. The smallest workload on any given processor in this situation is 1, and the largest is 3. Even with an ideal distribution, though, there would still be at least one processor with a workload of 3.

Our distribution scheme is not perfect, and although it works well in practice, there are situations, in which we must exercise caution. Consider, for example, $P = N_u = 1000$. Each processor operates on one row of u and one row of v . If we increase P to 1999, then almost all rows of u are assigned two processors instead of one. There will nevertheless be one extra row of u that is assigned to only *one* processor. That single processor will dominate the computation time. In other words, we will have nearly doubled the number of processors with no significant improvement in performance.

We may avoid such bottlenecks by choosing P wisely. If $P \geq N_u$, then P should be an integer multiple of N_u ($N_u, 2N_u, 3N_u, \dots$). If $P < N_u$, then P should be an integer fraction of N_u ($N_u/2, N_u/3, N_u/4, \dots$). It does not help to choose P between these values.

2.3 Parallel code

The code for the parallel algorithm is an extension of the serial code in C++. We define a new class called `bryTensorMPI` that inherits most of the data and functions from the previous class. 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 function that performs the calculations in Eqns. (13-19) actually stands apart from the class. We expect that an end-user would call the function, then instantiate objects based on the data that the function returns. It may be worthwhile to place it under the `bryTensorMPI` class in the future.

The parallel test program has two modes: memory-efficient, and process-efficient. The reason for the two options goes back to our assumption that each processor need only store one row each of u , v , and w . If we adhere strictly to this dictum, then if $P < N_u$, a processor may have to form all the rows of v as many times as it has rows of u . For example, if a processor is assigned 4 rows of u , then it will have to form each row of v close to 4 times. (We may reduce the number of formations slightly by sweeping backwards and forwards in the nested loops shown below.)

Setting the program to process-efficient mode causes each processor to generate its entire piece of u , v , and w simultaneously. This mode saves extra formations at the cost of increased memory requirements. As such, it may be of limited utility as

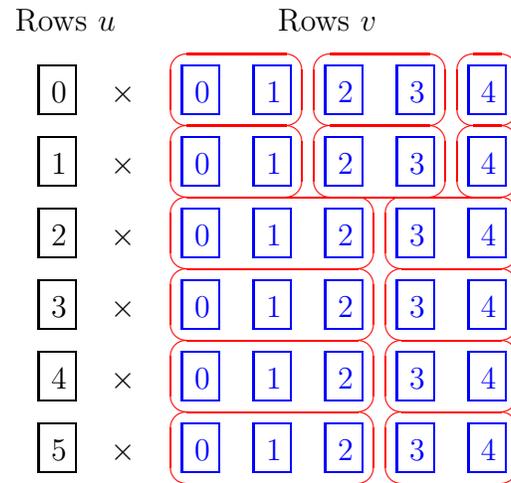
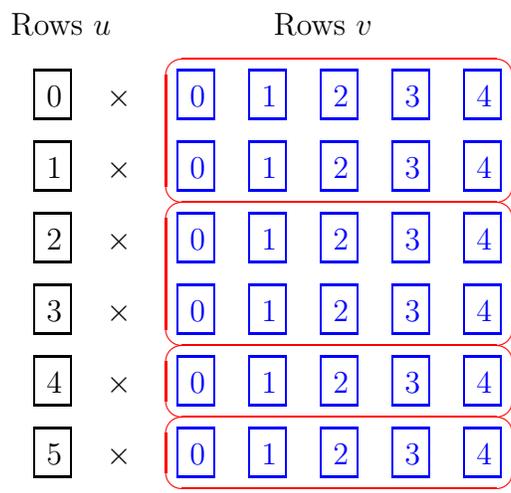


Figure 3: Distribution for $N_u = 6$, $N_v = 5$, with $P = 4$ (top) and $P = 14$ (bottom)

the size of the tensors grows. It does lead to some very interesting scaling results, however—see Sec. 3.

(Note that if $P \geq N_u$, then each processor is assigned only one row of u , so using memory-efficient mode does not penalize the run times. Numerical experiments have confirmed this observation.)

The parallel code test program only uses the broadcast capability of the class. The general procedure is the following:

1. Get MPI information.
2. Call the function to determine the rows of u and v that this processor will operate on. Store these in vectors \mathbf{x} and \mathbf{y} respectively.
3. Instantiate tensors of the proper size.
 - If in memory-efficient mode, this is one row each of u , v , and w .
 - Otherwise, it is the entire piece of u and v upon which this processor operates.
4. Load the characteristic matrices for u and v .
 - If this is the root processor, load from a file and broadcast to other processors.
 - Otherwise receive the broadcast.
5. Form and multiply pieces of u and v from their respective characteristic matrices.

- If in memory-efficient mode, use the following algorithm:

```

for ii=0:length(x)-1
    form row u( x(ii) ,:,: , ... ,:)
    for jj=0:length(y)-1
        form row v(y(jj),: , ... ,:)
        multiply u(x(ii),: , ... ,:)*v(y(jj),: , ... ,:)
    end
end

```

- Otherwise, form entire pieces of u and v assigned to this processor and multiply. That is,

```

form u(x(:),: , ... ,:)
form v(y(:),: , ... ,:)
multiply u(x(:),: , ... ,:)*v(y(:),: , ... ,:)

```

Note, too, that we promptly discard each piece of the tensor after we calculate it. In a real application, it would be necessary to do something with the result. Exactly what depends on the application of course, but almost certainly it would involve large amounts of communication, memory, and possibly disk access. Therefore, in order for the tensor multiplication code in this paper to be useful in practice, it must eventually contain efficient communication and storage routines that are tailored to individual applications. The timings reported in Sec. 3 might eventually constitute a relatively small part of the overall computational effort.

3 Results

We test the parallel code on the **DataStar** platform at the San Diego Supercomputer Center. We experiment with the multiplication of two rank-4 tensors with

Table 1: Weak scaling studies

P	N_u	Off-dim.	Time (s)
128	256	16	133.30
512	512	16	132.99
16	16	16	4.51
64	32	16	4.45
256	64	16	4.48
4	32	16	67.33
16	64	16	67.41
64	128	16	67.02
256	256	16	67.64

dimensions $N_u \times 16 \times 16 \times 16$, or $N_u \times 32 \times 32 \times 32$, where N_u varies from 16 to 512. (Hereafter, we assume that any two tensors being multiplied have identical dimensions, so we just write N_u for the number of rows.) The dimensions of the indices other than the first index (*i.e.*, 16 and 32 above) are known as the “off” dimensions in the discussions of this section. In all experiments, the characteristic matrices of the tensors are $N_u \times 5$, uniformly-distributed random numbers in the interval $[-1.05, 1.05]$. A complete, sorted list of parallel runs is in Sec. 5.

We vary the number of processors from 4 to 512 in order to study scaling effects. Fig. 4 shows computation time as a function of N_u for both choices of off-dimension. Each line represents a constant number of processors. All of the lines are generated using memory-efficient mode except for two.

Fig. 5 shows vertical slices of the Fig. 4. Each curve represents a different value of N_u and decreases according to an increasing number of processors. Note the sawtooth pattern in the $N_u = 32$ curve of the top plot. This is a demonstration of the effect—mentioned Section 2.3—that it does not pay to increase P unless we increase it in specific quantities.

In general, the method appears to scale very well. We use two methods to measure scaling. In the first, we try to obtain “weak” scaling results by increasing the number of processors with the size of the problem. This can be difficult to do properly because the size of the problem increases with N_u^2 , and thus we quickly run out of processors. Tab. 3 shows three sections of data in which the number of processors increases with the square of the number of rows. All runs are in memory-efficient mode.

The second method for calculating scaling information is to fit the curves of Fig. 4 in the least-squares sense with a power law, $T = \alpha y^\beta$, where T is time as plotted on the vertical axis. For a perfect scaling, we expect the exponent to be approximately 2. Tab. 2 contains fits and residuals for all curves. The curves in memory-efficient mode (including all data when $P \geq N_u$) have exponents that are actually smaller than 2. This is probably due to a shrinking contribution from the characteristic matrix load and broadcast in the timing total.

Process-efficient runs, on the other hand, exhibit some surprising behavior in Fig. 4 and Tab. 2. Timings actually grow sub-linearly with the workload, to the point where 64 processors running in non-memory-efficient mode are almost as fast as 128 processors in memory-efficient mode when $N_u = 128$!

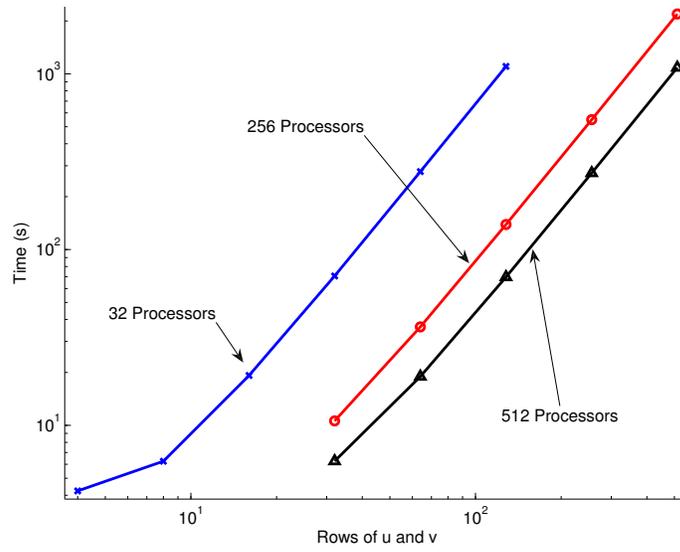
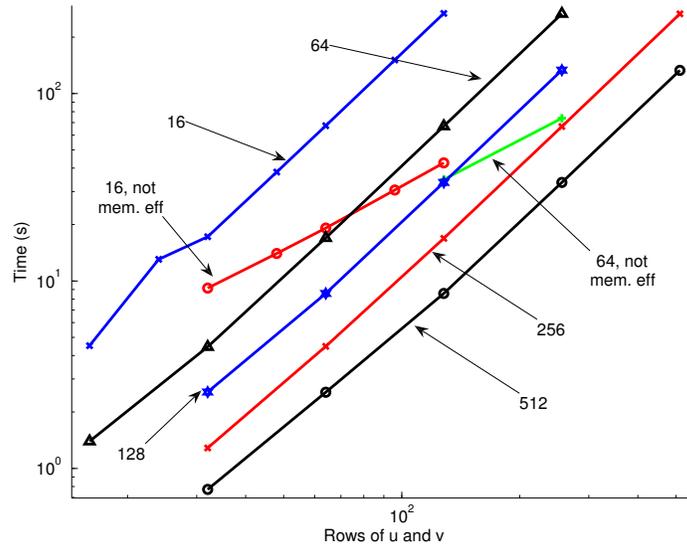


Figure 4: Computation time vs. N_u for $N_u \times 16 \times 16 \times 16$ (top) and $N_u \times 32 \times 32 \times 32$ (bottom) tensors

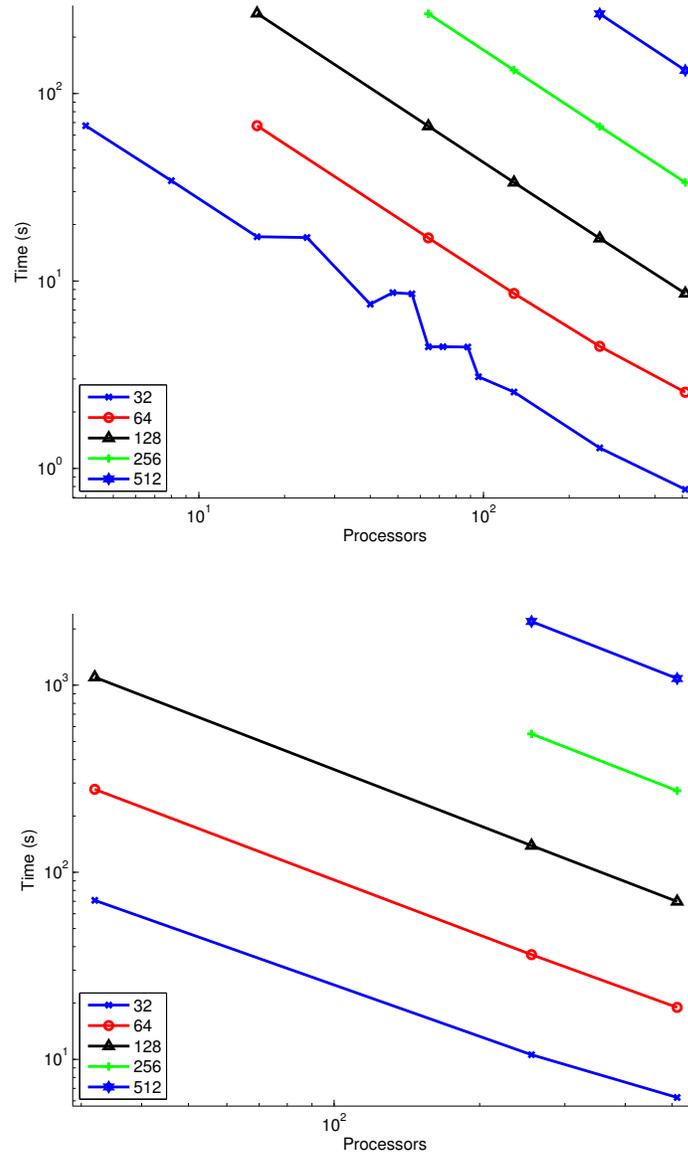


Figure 5: Computation time vs. number of processors for $N_u \times 16 \times 16 \times 16$ (top) and $N_u \times 32 \times 32 \times 32$ (bottom) tensors

Table 2: Power-law fits of time vs. N_u curves

P	Off-dim.	Mem.-eff.?	Exponent	Residual
32	32	yes	1.9851	15.5
256	32	yes	1.9948	8.1
512	32	yes	1.9877	7.1
16	16	yes	1.9766	11.3
16	16	no	1.1336	0.2
64	16	yes	1.9894	0.2
64	16	no	1.0857	0.0
128	16	yes	1.9848	0.2
256	16	yes	1.9939	0.2
512	16	yes	1.9815	0.2

The explanation for this sudden reduction is two-fold. First, we note that it is not surprising that the process-efficient calculations are faster because they require less formation from characteristic matrices. For example, if N_u is a multiple of P , say $N_u = tP$, then memory-efficient mode requires us to form v a total of t times, compared to once in process-efficient mode.

By itself, however, the reduced formation requirements are not enough to account for sub-linear growth in process-efficient mode. That phenomenon results from the relative differences between sizes of the pieces of tensors being multiplied. Recall that when $P < N_u$, each processor must operate on one or more rows of u , but *all* of v . Therefore, the full formation of v costs much more computation time than does the formation of a small piece of u . When we double N_u and N_v , we increase the total size of the problem by a factor of four, but we only increase the cost of forming v —and this is the term that dominates the computation—by a factor of two. Thus, the total cost in process-efficient mode only slightly more than doubles with a doubling of N_u and N_v .

Eventually, as N_u grows large, the number of processors will become small compared to N_u , and the scaling exponent will increase back to 2. Long before that happens, though, any modern system would run out of memory and process-efficient mode would be unavailable anyway. We conclude that while the sub-linear growth is an interesting feature of process-efficient mode, memory constraints would probably prevent its use for large, practical problems.

4 Discussion and future work

Despite its simplicity, the tensor multiplication code seems to be very robust, efficient, and scalable. Using 512 processors, we can multiply two 17-million-element rank-4 tensors with two 32-element reductions, all in less than 20 minutes. Scaling results are essentially perfect over the ranges considered, and the code has a relatively simple object-oriented interface.

Practical and fundamental experiments indicate that, despite the need for caution, the load balancing system is actually very good. It seems at first glance (especially given the sub-linear growth in process-efficient mode) that breaking up

both u and v so that each processor gets roughly the same size of each might be a better strategy. In the end, though, tensors u and v must be split into *contiguous* pieces to be distributed among processors. If we were to break up u and v simultaneously instead of sequentially, then memory-efficient mode would require many extra formations compared to the current setup, and formation—not multiplication—is the primary sink for computational resources. While it would be possible to improve process-efficient mode with different splittings, that mode is probably not useful for large tensors.

Some other improvements are immediately apparent, however, and one obvious place is in the k -index transformation. The reader has probably noticed that Eqn. (9) implies a large amount of symmetry. Indeed, any permutation of the indices of a tensor derived from Eqn. (9) will give the same value, as long as the values of the indices are within the allowable dimensions. The current code does not take advantage of this fact. If it did, we could potentially save large amount of both memory and computation time. For example, an $n \times n$ symmetric matrix only requires about half the storage space as a non-symmetric one, while a rank-4 tensor with index-permutation symmetry can reduce storage by a factor of 8.

Another place to improve the code is in the low-level multiplication itself. Sec. 2 outlines a technique for “unwrapping” the tensors and using matrix operations. This is a standard trick that could easily aid the serial and parallel codes, if done properly.

Finally, as mentioned at the end of Sec. 2.3, the multiplication *per se* may not be the limiting factor in an application that requires multiplication of large tensors. In order to use the code in a practical problem, it will be necessary to attach many more functions for handling post-processing and storage of the tensors being multiplied.

5 Appendix: Output of parallel runs

The following table contains the reduced output of the parallel runs on **DataStar**. The form of the tensor multiplication in each case is $w_{ijmn} = u_{ijef}v_{mnef}$. The “Load” time is the time required to load and broadcast the characteristic matrices. The “Comp.” time is the time required to do the computation.

P	i	j	m	n	e	f	Mem. eff.?	Load (s^{-2})	Comp. (s)	Total (s)
4	32	16	32	16	16	16	y	0.1	67.3	67.3
8	32	16	32	16	16	16	y	0.1	34.3	34.3
16	16	16	16	16	16	16	y	0.0	4.5	4.5
16	16	32	32	16	16	16	y	0.0	9.2	9.2
16	16	48	48	16	16	16	y	0.1	14.3	14.3
16	16	64	64	16	16	16	y	0.1	19.1	19.1
16	16	96	96	16	16	16	y	0.1	30.4	30.4
16	16	128	128	16	16	16	y	0.2	42.6	42.6
16	24	16	24	16	16	16	y	0.0	13.0	13.0
16	32	16	32	16	16	16	n	0.0	9.2	9.2
16	32	16	32	16	16	16	y	0.0	17.2	17.2
16	48	16	48	16	16	16	n	0.1	14.0	14.0

16	48	16	48	16	16	16	y	0.1	38.1	38.1
16	64	16	64	16	16	16	n	0.1	19.1	19.1
16	64	16	64	16	16	16	y	0.1	67.4	67.4
16	96	16	96	16	16	16	n	0.1	30.6	30.6
16	96	16	96	16	16	16	y	0.1	151.0	151.1
16	128	16	128	16	16	16	n	0.3	42.6	42.7
16	128	16	128	16	16	16	y	0.1	267.6	267.6
24	32	16	32	16	16	16	y	0.0	17.0	17.0
32	4	32	4	32	32	32	y	0.0	4.2	4.2
32	8	32	8	32	32	32	y	0.0	6.2	6.2
32	16	32	16	32	32	32	y	0.0	19.2	19.2
32	32	32	32	32	32	32	y	0.0	70.8	70.8
32	64	32	64	32	32	32	n	0.1	149.3	149.3
32	64	32	64	32	32	32	y	0.1	277.7	277.7
32	128	32	128	32	32	32	y	0.1	1102.2	1102.2
40	32	16	32	16	16	16	y	0.1	7.5	7.5
48	32	16	32	16	16	16	y	0.9	8.6	8.7
56	32	16	32	16	16	16	y	0.1	8.5	8.5
64	16	16	16	16	16	16	y	0.0	1.4	1.4
64	32	16	32	16	16	16	y	0.0	4.5	4.5
64	64	16	64	16	16	16	y	0.1	17.0	17.0
64	128	16	128	16	16	16	n	0.3	34.7	34.8
64	128	16	128	16	16	16	y	0.9	66.9	67.0
64	256	16	256	16	16	16	n	0.2	73.8	73.8
64	256	16	256	16	16	16	y	1.0	266.4	266.5
72	32	16	32	16	16	16	y	1.1	4.3	4.5
88	32	16	32	16	16	16	y	0.1	4.4	4.4
96	32	16	32	16	16	16	y	0.1	3.1	3.1
128	32	16	32	16	16	16	y	0.1	2.6	2.6
128	64	16	64	16	16	16	y	0.3	8.5	8.6
128	128	16	128	16	16	16	y	0.3	33.5	33.6
128	256	16	256	16	16	16	n	0.2	68.9	69.0
128	256	16	256	16	16	16	y	0.5	133.3	133.3
256	32	32	32	32	32	32	y	0.4	10.5	10.6
256	32	16	32	16	16	16	y	0.1	1.3	1.3
256	64	32	64	32	32	32	y	0.4	36.3	36.3
256	64	16	64	16	16	16	y	0.1	4.5	4.5
256	128	16	128	16	16	16	y	0.1	16.9	16.9
256	128	32	128	32	32	32	y	0.4	138.9	138.9
256	256	16	256	16	16	16	y	0.5	66.6	66.6
256	256	32	256	32	32	32	y	0.3	549.0	549.0
256	512	16	512	16	16	16	n	0.4	137.6	137.6
256	512	32	512	32	32	32	n	0.5	1166.5	1166.5
256	512	16	512	16	16	16	y	0.7	265.9	266.0
256	512	32	512	32	32	32	y	0.5	2191.8	2191.8
512	32	16	32	16	16	16	y	0.1	0.8	0.8
512	32	32	32	32	32	32	y	0.2	6.2	6.3
512	64	16	64	16	16	16	y	0.1	2.5	2.5
512	64	32	64	32	32	32	y	0.3	18.9	18.9
512	128	16	128	16	16	16	y	0.1	8.6	8.6

512	128	32	128	32	32	32	y	0.3	69.8	69.8
512	256	16	256	16	16	16	y	1.2	33.4	33.6
512	256	32	256	32	32	32	y	0.5	272.9	272.9
512	512	16	512	16	16	16	y	0.7	132.9	133.0
512	512	32	512	32	32	32	y	0.7	1085.7	1085.7

References

- [1] B. W. Bader and T. G. Kolda. Algorithm xxx: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4), December 2006.
- [2] 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.
- [3] M. Challacombe, E. Schwegler, and J. Almlöf. Fast assembly of the Coulomb matrix: A quantum chemical tree code. *Journal of Chemical Physics*, 104(12):4685–4697, 1996.
- [4] T. J. Chung. *Applied Continuum Mechanics*. Cambridge University Press, New York, 1996.
- [5] 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.
- [6] R. Kobayashi and A. P. Rendell. A direct coupled cluster algorithm for massively parallel computers. *Chemical Physics Letters*, 265:1–11, 1997.
- [7] 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.
- [8] A. J. McConnell. *Applications of Tensor Analysis*. Dover, New York, 1957.
- [9] 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.
- [10] 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.
- [11] J. R. Ruíz-Tolosa and E. Castillo. *From Vectors to Tensors*. Springer-Verlag, Berlin, 2005.
- [12] 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.