

# Tensor Multiplication on Parallel Computers

Bryan Rasmussen, HPC-4

**T**ensor multiplication is a ubiquitous task in many scientific and engineering applications. While there has been a great deal of research on efficient numerical methods for matrix multiplication on parallel machines, the general multiplication of arbitrary-rank tensors with an arbitrary number of contractions has seen less progress.

We consider a new set of C++ classes that compute, store, and multiply tensors on both serial and parallel platforms. The original, motivating application is in computational chemistry, for example [1], although the work should apply to many disparate areas such as fluid mechanics, general relativity, and quantum mechanics [2].

For the purposes of this project, a “tensor” is simply a multidimensional box of numbers, stored in a logical order. A rank-0 tensor is a scalar, a rank-1 tensor is a vector, etc. The code performs tensor multiplications of the form

$$w_{a_1 a_2 \dots a_m b_1 b_2 \dots b_n} = u_{a_1 a_2 \dots a_m c_1 c_2 \dots c_p} v_{b_1 b_2 \dots b_n c_1 c_2 \dots c_p}, \quad (1)$$

where  $m$ ,  $n$ ,  $p$  and their associated dimensions are completely arbitrary. The multiplication in Equation (1) assumes the Einstein summation convention which is to say that we sum over repeated indices. (For example, the inner product of two vectors is  $u_i v_i$ .)

We also assume that every tensor has a very compact storage method called the  $k$ -index transformation, also known as the Tucker transformation. This transformation says that a rank- $k$  tensor,  $v$ , has a “core tensor,”  $A$ , and characteristic matrices  $z^1, z^2, \dots, z^k$  such that any element of the  $v$  has the form

$$v_{i_1 i_2 \dots i_k} = z^1_{i_1 j_1} z^2_{i_2 j_2} \dots z^k_{i_k j_k} A_{j_1 j_2 \dots j_k}. \quad (2)$$

This transformation becomes very important for minimizing storage and communication costs in the parallel algorithm. The main savings comes from the fact that the core tensor,  $A$ , is usually known *a priori* from straightforward analytic expressions.

To create a parallel algorithm, we assign each processor a different piece of the three tensors in Equation (1). We assume that each processor has enough memory to store one row of  $u$ ,  $v$ , and  $w$  simultaneously, where a row is the piece of the tensor corresponding to a single, fixed index in the first position.

Each processor then creates only its portions of  $u$  and  $v$  one row at a time and multiplies them together to form rows of  $w$ . This approach has several advantages, namely that it requires very little message passing, and it minimizes the number of redundant formations and calculations.

One disadvantage of our approach is that it requires each processor to work on a large piece of the problem for a long time, thus increasing the probability that a single processor failure will sabotage the computation. Another limitation is that we must increase the number of processors in potentially large step-increments in order to take advantage of larger clusters. (The ideal number of processors is an integer multiple or divisor of the number of rows in  $u$ .)

A typical solution to these difficulties is to subdivide the problem and then use a scheduler to balance the workload. Unfortunately, small divisions of labor necessitate redundant calculations, so there is a limit to how we can chop up the problem. At present, the code does include a scheduler with simple rules on how to divvy the pieces of the tensors, but that

capability remains untested. Despite its simplicity (or perhaps because of it), the all-at-once method seems to be very robust, efficient, and scalable.

Consider the following results on the DataStar IBM Power4 computer at the San Diego Supercomputing Center. We experiment with the multiplication of two rank-4 tensors of equal dimensions. The tensors descend from  $k$ -index transformations where the core tensors consist of all ones, and the characteristic matrices all have five columns of uniform random numbers. The theoretical problem size therefore increases with the square of the number of rows,  $N_u$ . Figure 1 shows computation time as a function of  $N_u$ . Each line represents a constant number of processors.

The tensor classes are still under active development, particularly with regard to the  $k$ -index transformation. Through judicious storage of partial calculations, it is possible to reduce the complexity of Equation (2) to  $N^{k+1}$ , where  $N$  is the largest dimension of any of the three tensors. The naïve way of doing this often leads to temporary storage requirements that exceed the memory available to any current processor, and indeed to any processor likely to be produced in the next 20 years.

In theory, we can circumvent this restriction by fusing nests of loops—indeed, there is significant progress in this area—but optimal loop fusion is a complicated process for arbitrary-rank tensors. Other research groups have developed algorithms for generating code automatically to suit specific requirements and platforms [3]. We aim instead for a more general approach that may not be optimal but still allows for increased productivity for practical problems.

Our near-term goal is an efficient, parallel, arbitrary-dimensional algorithm that integrates the  $k$ -index transformation with tensor multiplication to enable a wide variety of new calculations with only a

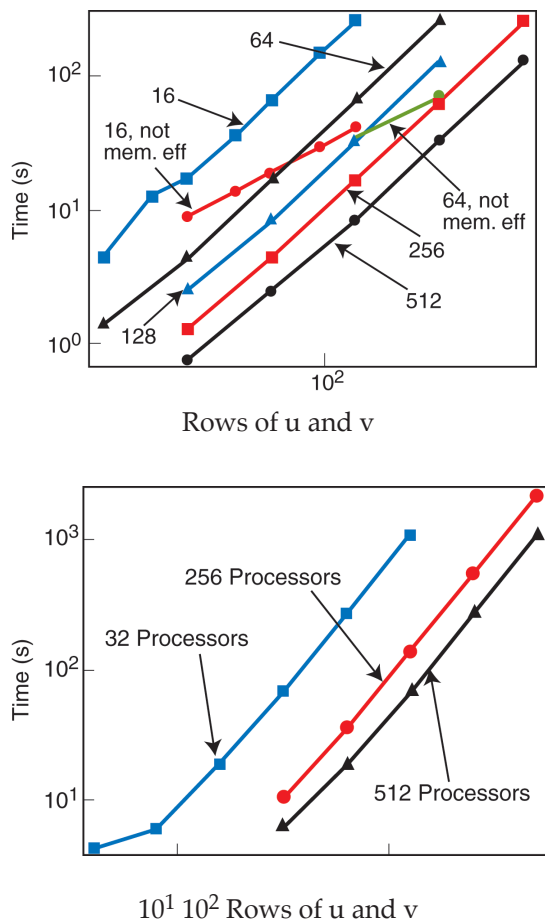
simple set of C++ classes. Preliminary results indicate that this is possible in the near future.

**For more information contact Bryan Rasmussen at [bryanras@lanl.gov](mailto:bryanras@lanl.gov).**

[1] 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," *Chem. Phys. Lett.*, **194**, 84 (1992).  
 [2] A. J. McConnell, *Applications of Tensor Analysis*, (Dover, New York, 1957).  
 [3] G. Baumgartner, et al., "Synthesis of high-performance parallel programs for a class of *Ab Initio* quantum chemistry models," *Proceedings of the IEEE*, **93**(2), 276 (February 2005).

**Funding Acknowledgements**

This research was supported by the NNSA tri-Lab Advanced Simulation and Computing Program.



**Fig. 1.** 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). Scaling appears to be nearly perfect over the ranges studied. The lines labeled "not mem. eff" refer to a capability of the code to trade computation time for memory. This mode will not be practical for most applications due to memory constraints.