OPTIMIZING THE REDUCTION OF TENSORS TO IRREDUCIBLE REPRESENTATIONS

SONG KIM*

Abstract. Tensors are mathematical objects that are used to describe concepts throughout mechanics and material science. When working with tensors, it can be convenient to contract larger reducible tensors with a change of basis matrix into a single irreducible representation without losing information, for example if these tensors are being operated on by composable neural network layers. However, the current method of contracting tensors is slow when contracting larger tensors and has room to be optimized. In this paper, we develop a parallelized divide-and-conquer algorithm for finding a change of basis matrix that contracts a reducible representation to its irreducible components.

Key words. parallel algorithms, tensor products

AMS subject classifications. 68W10

1. Introduction.

In mathematics, tensors of rank n are objects with n indices. In the physical sciences, tensors are also assumed to obey certain transformation rules under specific group symmetries. In addition to their applications in the physical sciences (for example, as representations of moments of inertia or elasticity), tensors have recently been used in the construction of equivariant neural networks.

One common tensor operation is taking the reduced tensor product, or contracting one or more reducible tensor objects into a single irreducible tensor. This process has the potential to be quite slow, especially when reducing larger tensors with more symmetries. So, through this paper, we develop a faster parallelized method to determine this reduced tensor product. We first provide the mathematical background behind the group representations we will be working with. Then, in Section 2 we discuss the algorithms we used to find the change of basis matrix, and in Section 4 we review the performance of each algorithm.

1.1. Groups.

A mathematical group $(G, \otimes)$ is a set G and a binary operation $\otimes = G \times G \to G$ such that

1. $a \otimes (b \otimes c) = (a \otimes b) \otimes c$
2. there exists an identity element $I \in G$ such that $g \otimes I = I \otimes g = g$ for all $g \in G$
3. for every element $g \in G$, there exists an inverse element h such that $g \otimes h = h \otimes g = I$

A particular group that is highly relevant to the physical sciences is the group of $O(3)$, the 3D orthogonal group – more commonly known as the group of 3D rotations and inversion $(x, y, z) \to (-x, -y, -z)$.

1.2. Reducible and irreducible representations.

A group representation D describes the action of a group G on a vector space V:

$$D : G \to \text{linear operation on } V.$$

Colloquially, it is common to refer to both the group representation and the vector space it acts on as the "representation". Reducible representations within $O(3)$ are representations D that can be decomposed into smaller vector spaces W such that D

*Massachusetts Institute of Technology, Cambridge, MA is still a group representation of $O(3)$ under W. The "smallest" representations are called irreducible representations, or "irreps". For the $SO(3)$ group, these representations are the Wigner D matrices.

Within the $O(3)$ group, irreps can be represented as objects that have degree l and parity p, which is either odd (o) or even (e). For example, a scalar could be represented as a $0e$ irrep and a vector could be represented as a $1o$ irrep. Moreover, irreps can be combined to create more complicated irrep objects, for example $100 \times 0e \oplus 50 \times 1o$ (where $\oplus$ is the direct sum). For these larger irrep objects, the multiplicity of a particular "unit" irrep is the number of times it occurs within the larger object. So in the previous example, $0e$ has multiplicity 100 and $1o$ has multiplicity 50.

These irreps satisfy the following properties:

1. Any representation within $O(3)$ can be decomposed into a direct sum of irreps.
2. Any physical quantity, under the action of $O(3)$, transforms with a representation of $O(3)$.
3. The *dimension* of an irrep with degree l is equal to $2l + 1$.

We can use the following operations to define vector spaces built from irreps:

1. Direct sum ($\oplus$), the concatenation of vector spaces composed of irreps:
   In our notation, we add the multiplicities of irreps that are the same.
   $1 \times 0e \oplus 1 \times 0e = 2 \times 0e$
   $1 \times 0e \oplus 1 \times 0o = 1 \times 0e \oplus 1 \times 0o$
2. Tensor product ($\otimes$), the generalization of multiplication:
   Parities are multiplied as follows: $o \otimes o = e$, $o \otimes e = o$, $e \otimes e = e$
   Degrees: all irreps of parity $p_1 \otimes p_2$ and degree between $|l_1 - l_2|$ and $l_1 + l_2$ are represented within the final product.
   $1 \times 1o \otimes 1 \times 1o = 1 \times 0e \oplus 1 \times 1e \oplus 1 \times 2e$

Both irreducible and reducible representations can be described as tensors. If the representation M is the product of two irreps I_1 and I_2, M can be written as the tensor M_{ij}, where index i spans the dimensions of irrep I_1 and j spans the dimensions of irrep I_2. However, as we will discuss below, these two indices can be contracted into a single index over the irrep basis using tensor products.

For example, the irrep $2e$, which has dimension 5, can be represented as

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}$$

for some $a_1, \dots, a_5 \in \mathbb{R}$. The product of irreps $1o \otimes 1o$ can be represented as the 3×3 matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

for some $a_{11}, \dots, a_{33} \in \mathbb{R}$. This tensor product can further be broken down into the direct sum of the three irreps $0e \oplus 1e \oplus 2e$, where the $0e$ irrep represents the trace of the 3×3 matrix, the $1e$ irrep represents the antisymmetric components, and the $2e$ irrep represents the symmetric traceless part.

In the pseudocode in Section 2, single irreps will be referred to by variable name, while irreps with multiplicity greater than 1 will be written as $\mathrm{MulIr}(m, i)$ where m is the multiplicity and i is the irrep itself.

1.3. Clebsch-Gordan coefficients. The Clebsch-Gordan coefficients C are defined such that given tensors $x \in X, y \in Y, z \in Z$ where X, Y, and Z are vector spaces in $\mathbb{R}^s$, there exists a coefficient C such that x and y can be contracted with C to return a single equivalent tensor z. In other words, using Einstein summation notation (Appendix A),

$$x_i y_j C_{ijk} = z_k$$

where

$$\left(D^X_{i'i}(g)\, x_i\right) \left(D^Y_{j'j}(g)\, y_j\right) C_{i'j'k'} = D^Z_{k'k}(g)\, z_k$$

for all operations $g \in G$ and $D^Z(g)$ is the representation of operation g within the group G on vector space Z. The Clebsch-Gordan coefficients for $SO(3)$ are computed from integrals over the basis functions of a given irreducible representation, e.g. the real spherical harmonics, and are tabulated to avoid unnecessary computation.

1.4. Reduced tensor products. Given a single reducible tensor object M, there exists a change of basis Q comprised of multiple Clebsch-Gordan coefficients such that M can be contracted into a single set of irreps z. Or,

$$M_{ij} Q_{ijk} = z_k.$$

2. Algorithms.

Given the symmetries and antisymmetries between different indices of a reducible representation, and the irreps spanning the dimensions of each index, we find a change of basis matrix Q as described above that will contract that tensor into a single irreducible representation.

All versions of our algorithm are comprised of the same three sub-algorithms, which we will refer to as **find_P**, **find_R**, and **find_Q**. We will describe the overall structure of the entire algorithm first, both in its original and improved forms, then discuss each sub-algorithm in more detail in subsection 2.5.

2.1. Inputs and outputs. For a given tensor, its indices could be related by symmetries or antisymmetries. For convenience, we write these (anti)symmetries as "formulas" where permutations of the indices are set "equal" to each other. For example, if the tensor M_{ijk} was symmetric with respect to i and j, we would write that as $ijk = jik$, where indices i and j appear in the same locations between the two permutations. On the other hand, if M was antisymmetric with respect to i and j, that would be written as $ijk = -jik$. Incomplete (anti)symmetric relations are allowed as inputs (for example, $ijk = jki$ has one more equivalent permutation kij), as our algorithm will find the remaining symmetries.

At the beginning of the algorithm, we also define the irreps associated with some or all of the input's indices. These irreps must be consistent with the relations defined by the symmetry formula described above.

2.2. Overall algorithm (original). Our starting algorithm for finding the change of basis matrix was implemented in Python by Geiger et al. [1] and is described as Algorithm 2.1.

Let M be the input tensor and X be the full space of our tensor product. Given M's symmetry formula, we first generate the complete set F of equivalent permutations by composing and/or inverting all existing permutations. Then, we use the sub-algorithms **find_P** and **find_R** to find matrices P and R. P (the "permutation basis") is a basis for the subspace of X where the action of permutation is stable (invariant, potentially with a sign change). R (the "rotation basis") is a basis of X such that the representations of rotations are block diagonal matrices.

Finally, we find a change of basis matrix Q from both P and R using the sub-algorithm **find_Q**. Q is a basis of the same subspace of X as P with the properties of both P and R.

In terms of efficiency, this algorithm is slowest at the **find_Q** segment. Here, we find the eigenvalue decomposition of several matrices, which is slow and memory-expensive. In addition, **find_R** involves creating a large matrix (R itself) and is also memory-expensive.

Algorithm 2.1 Find the change of basis (original)

```
Input: Symmetry formula $F$, dictionary $D_i$ mapping indices to corresponding irreps
Output: Change of basis matrix $Q$ for contracting representations of the given form into irreps
  $F :=$ the full set of equivalent index permutations from the given (anti)symmetric relation
  $P :=$ permutation basis, found using find_P
  $R :=$ rotation basis, found using find_R
  $Q :=$ change of basis matrix between $P$ and $R$, found using find_Q
  return $Q$
```

2.3. Overall algorithm (divide and conquer). One of our improvements to this algorithm is to rework it as a divide-and-conquer algorithm. In this algorithm, we begin the same way we did in Algorithm 2.1, by finding F and P. We can then split this problem into two sub-problems by dividing the input indices into two subsets and defining each subset's symmetry relations. We then solve each sub-problem, which results in two smaller change of basis matrices Q_1 and Q_2. In practice, we iterate through all possible two-way splits of the indices and use the split that will result in the smallest Q_1 and Q_2. Q_1 and Q_2 are then combined into the overall rotation basis R through **find_R**, and P and this new R are passed into **find_Q** to obtain the final change of basis matrix for the entire larger problem. The outline of this approach is shown as Algorithm 2.2.

This divide and conquer approach is faster than the original algorithm because much of the work is done using smaller matrices. On the other hand, because of the multiple sub-problems that are created and merged throughout the algorithm, the divide and conquer algorithm also uses more memory than the original algorithm.

2.4. Parallelized algorithm. The other modification we made to the algorithm is to parallelize it. Starting with Algorithm 2.2, we implemented multithreading on the sub-algorithms **find_R** and **find_Q**, as both sub-algorithms involve performing operations independently on a series of irreps. Moreover, each iteration of **find_Q**'s main loop still involves finding the eigenvalue decomposition of several matrices, which is the slowest single operation in the entire algorithm.

The main issues with parallelization for both **find_R** and **find_Q** were that both algorithms required writing to parts of a shared list of unspecified length, and in the case of **find_Q**, maintaining the same order between two separate lists. These issues were resolved using locks, which means that this list-writing could theoretically behave serially.

2.5. Sub-algorithms. Now, we will describe the sub-formulas **find_P**, **find_R**, and **find_Q** in greater detail.

Algorithm 2.2 Find the change of basis (divide and conquer)

```
Input: Symmetry formula $F$, dictionary $D_i$ mapping indices to corresponding irreps
Output: Change of basis matrix $Q$ for contracting representations of the given form into irreps
  $F$ := the full set of equivalent index permutations from the given (anti)symmetric relation
  $P$ := full permutation basis, found using find_P
  Find the best way to split the given indices into two sub-problems
  $Q_1$ := Change of basis matrix from first sub-problem, found recursively
  $Q_2$ := Change of basis matrix from second sub-problem, also found recursively
  $R$ := Full rotation basis, found by combining both sub-problems (find_R)
  $Q$ := Full change of basis matrix, found using find_Q
  return $Q$
```

find_P (Algorithm 2.3). Our first sub-algorithm is **find_P**, which finds the permutation basis as defined in subsection 2.2.

For a given tensor M, we start with a sequence of indices f_0, a complete list of index permutations F, and a list D containing the dimension of each index's corresponding irrep. We create a set B_{full} containing all possible combinations of dimensions at each index, one for each degree of freedom if M had no symmetry constraints. We create another set B containing all equivalent permutations (and signs for symmetry/antisymmetry) of each element in B_{full}. Then, we find the change of basis matrix between B_{full} and B by concatenating the permutations in B; this matrix is P.

find_R (Algorithm 2.4). Our next sub-algorithm is **find_R**, which finds the rotation basis R of our starting tensor by combining the results of two sub-problems.

find_R takes as input the change of basis matrices Q_1, Q_2 and output irreps I_1, I_2 from the two sub-problems. (**find_R** in the not-divide and conquer algorithm is the equivalent of taking Q_2 to be the identity matrix and I_2 to be a single $0e$ irrep.) It builds R up in parts: for each pair of irreps $[\mathrm{MulIr}(m_1, i_1), \mathrm{MulIr}(m_2, i_2)]$ from I_1 and I_2 respectively, we take the segments of Q_1 and Q_2 respectively that correspond to each irrep; call these segments S_1 and S_2. In particular, S_1 contains a block of $m_1 \cdot \dim(i_1)$ elements from Q_1, and similarly for S_2. Then, for each unit irrep i_{out} within the tensor product $i_1 \otimes i_2$, we contract S_1 and S_2 with the Clebsch-Gordan coefficient of i_1, i_2, i_{out} as $(S_1)_{mia}(S_2)_{njb}C_{ijk}$ and add the result to R.

In practice, R as returned by our implementation of this algorithm is a dictionary, with each contraction being stored in a list associated with its corresponding i_{out} irrep. This is for easier access in **find_Q** and to save space, as otherwise R would be a large block diagonal matrix.

In the parallelized version, the algorithm is split onto multiple threads along each irrep within I_1 (so line 2 in Algorithm 2.4). Because the locations of each S_1 segment depend on the dimensions of each preceding irrep in I_1, we facilitate creating these segments by determining the bounds of each segment before running the parallelized iterations.

find_Q (Algorithm 2.5). Our last sub-algorithm is **find_Q**, which computes one possibility for Q given permutation basis P and rotation basis R.

We assume R is specified in dictionary form, as described above. For each irrep

Algorithm 2.3 find_P

```
Input: indices $f_0$, list of index permutations $F$, list $D$ of the dimensions of each index's irrep
Output: Permutation basis $P$
  $B_{full}$ := Cartesian product of $[1 : d$ for $d$ in $D]$ {full basis}
  $B$ := empty set {new basis}
  for $x$ in $B_{full}$ do
    $x_s := \{(s, [x[i]$ for $i$ in $p])$ for $(s, p)$ in $F\}$ {signed $x$}
    if $(-1, x)$ not in $x_s$ then
      Add both $x_s$ and $x_s$ with the opposite signs to $B$
    end if
  end for
  Sort $B$
  $d_{sym}$ := length of $B$
  $P$ := zeros($d_{sym}$, length of $B_{full}$)
  for $x$ in $B$ do
    $x_2$ := element of $x$ that contains the most symmetric (as opposed to antisymmetric) relations
    for $(s, e)$ in $x_2$ do
      $j := 0$
      for $(k, d)$ in zip$(e, D)$ do
        $j = j \cdot d$
        $j = j + (k - 1)$
      end for
      $P[i, j + 1] = s/\sqrt{\text{length}(x_2)}$
    end for
  end for
  Reshape $P$ into dimensions $(P, d_{sym}, D...)$
  return $P$
```

i represented in R, we take one slice R_i of the submatrix of R associated with i. We then construct a block matrix $A = \begin{bmatrix} R_i R_i^T & -R_i P^T \\ (-R_i P^T)^T & P P^T \end{bmatrix}$ and find the eigenvalues and associated eigenvectors of A. If A has at least one negative eigenvalue, we create matrix X containing the eigenvectors associated with each negative eigenvalue and create a basis by orthonormalizing XX^T. We multiply each vector in this basis by a correction factor to obtain the final change of basis matrix Q.

When parallelizing **find_Q**, we create a thread for each irrep i in I_R.

3. Implementation. We implemented three versions of the algorithm in Julia 1.7: the original algorithm, the divide-and-conquer algorithm, and the parallelized version of the divide-and-conquer algorithm.

4. Results. Our three algorithms were timed and compared using different versions of the bispectrum problem, where all possible symmetries of three indices are represented ($ijk = jik = jki$). The bispectrum, or bispectral density, is a statistic used to search for nonlinear interactions. In terms of tensors, where the standard power spectrum is the dot product of two representations, the bispectrum is the tensor product, then the dot product of two representation. Although in the true bispectrum problem we would restrict the output to contain only scalars, in this instance

Algorithm 2.4 find_R

```
Input: Input irreps $I_1$ and $I_2$, input $Q_1$ and $Q_2$
Output: Output irreps $I$, rotation basis $R$
  Define $R := \{\}$, $I :=$ empty irreps, $k_1 := 1$
  for MulIr$(m_1, i_1)$ in $I_1$ do
    $S_1 := Q_1[k_1 : k_1 + m_1 \cdot \dim(i_1) - 1]$
    Reshape $S_1$ into dimensions $(m_1, \dim(i_1), :)$
    $k_1 = k_1 + \dim(i_1) \cdot m_1$
    $k2 := 1$
    for MulIr$(m_2, i_2)$ in $I_2$ do
      $S_2 := Q_2[k_2 : k_2 + m_2 \cdot \dim(i_2) - 1]$
      Reshape $S_2$ into dimensions $(m_2, \dim(i_2), :)$
      $k_2 = k_2 + \dim(i_2) * m_2$
      for $i_{out}$ in $i_1 \otimes i_2$ do
        $I = I \oplus \text{MulIr}(m_1 m_2, i_{out})$
        $C :=$ Clebsch-Gordan coefficient of $i_1, i_2, i_{out}$
        Append $K_{mnkab} = (S_1)_{mia}(S_2)_{njb}C_{ijk}$ to $R[i_{out}]$
      end for
    end for
  end for
  $R :=$ matrix formed by concatenating $R$'s elements
  return $I, R$
```

we will return the full result of contracting the input tensor. Because of all of the symmetries involved, the computations involved in this contraction are particularly time-consuming, making this a good test case.

For each algorithm (original, divide-and-conquer, parallelized), we timed each iteration of the bispectrum problem using 1, 2, 4, or 8 threads, across 1, 2, 4 or 6 CPUs, on the same or different nodes. The irrep corresponding to each index was taken to be $0e$, $0e \oplus 1o$, $0e \oplus 1o \oplus 2e$, or $0e \oplus 1o \oplus 2e \oplus 3o$; these cases will be referred as $l = 0$ through $l = 3$ after their irrep of largest degree. We ran these tests on the MIT Supercloud, which uses Intel Xeon Platinum 8260 processors.

We compared the three algorithms' performances (see Figure 1) when run on 1 CPU across 1, 2, 4, and 8 threads. In all but the $l = 0$ case, the parallelized version of the divide and conquer algorithm was faster than the serial divide and conquer algorithm, with the parallelized algorithm being almost twice as fast at higher thread counts. The reason this did not hold for $l = 0$ could be that this was the smallest test case and there were too few computations to offset the cost of creating and handling threads. Also, the original (serial) algorithm was the slowest of the three algorithms for $l = 2$ and $l = 3$ (about 3 times slower than either of the other algorithms), but not for the two smaller cases. Again, in the smaller cases splitting the original bispectrum problem into two smaller problems could be less efficient because splitting and recombining the sub-problems could be more costly than the computations needed to solve each sub-problem.

We also sought to examine how well our parallelized algorithm scaled across more resources (Figure 2). In particular, we compared the algorithm's performance when run on multiple nodes, 1 CPU per node. Here, we found that using 1 node was generally slower than using more than 1, and that using 1 thread was generally slower than

Algorithm 2.5 find_Q

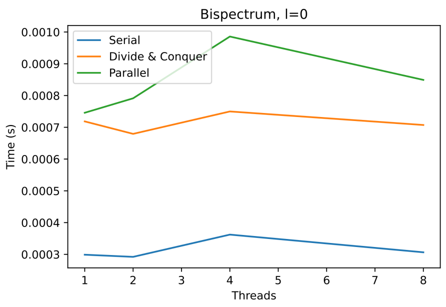
```
Input: Permutation basis $P$, rotation basis $R$, list $I_R$ of irreps represented in $R$
Output: Change of basis matrix $Q$
  $Q := []$, $I_{out} := []$
  for irrep $i$ in $I_R$ do
    $m :=$ length($R[i]$)
    $B_{o3} :=$ all elements of $R[i]$ combined into one matrix
    Reshape $B_{o3}$ to have shape $(m, \dim(i), :)$
    $R_i := B_{o3}[:, 1, :]$
    $A := \begin{bmatrix} R_i R_i^T & -R_i P^T \\ (-R_i P^T)^T & P P^T \end{bmatrix}$
    $X_- :=$ eigenvectors corresponding to negative eigenvalues of $A$ – find the eigen-
    value decomposition of $A$
    if $X_-$ is not empty then
      $X := X_-[1 : m, :]$
      $X' :=$ orthonormalized version of $XX^T$
    else
      $X' := [0]$
    end if
    for $x$ in $X'$ do
      $C_i := x_u (B_{o3})_{ui...}$
      cor $:= \dim(i)/|C|$ {correction factor}
      $C :=$ cor $\cdot$ $C$
      Add $C$ to $Q$
      Add $i$ to $I_{out}$
    end for
  end for
  Combine $Q$ into a single matrix
  return $I_{out}, Q$
```

using more than 1 thread. (The exception here is once again the $l = 0$ case.) However, for both sets of parameters, we also found that adding the second node/thread caused most of the improvements in speed, then adding subsequent nodes or threads led to much less improvement in times. In fact, when considering the number of threads, the impact of adding additional threads on the run time tapers off after 4 threads.

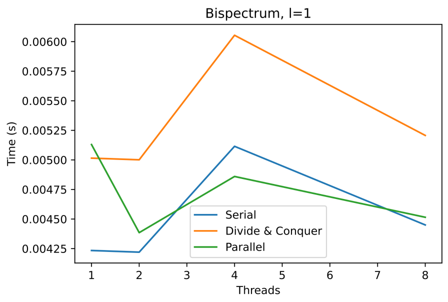
5. Conclusion and next steps.

In summary, we developed a parallelized divide and conquer algorithm for finding the change of basis matrix between a reducible representation and its irrep decomposition. This algorithm ran faster than the otherwise equivalent serial divide-and-conquer algorithm on all but the smallest inputs, and much better than the original non-parallelized, non-divide and conquer algorithm for the same task (again, with the exception of smaller inputs). Its speed improves upon adding more threads or computing power, but these improvements taper off rather quickly.

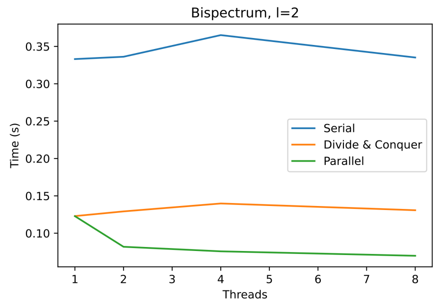
One improvement we could make to our parallelized algorithm is in its memory use. For larger tensor reduction problems, the parallelized segments of both **find_R** and **find_Q** will involve iterating over many irreps and therefore creating many threads, which in itself is expensive. In addition, these threads will all have to write to a central data structure. As an example, we tested all three versions of



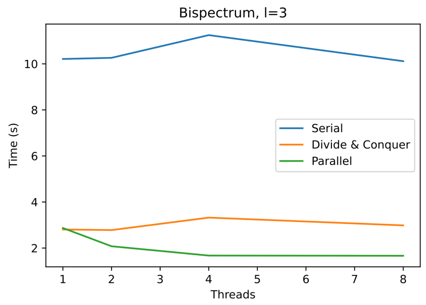
(a)



(b)



(c)



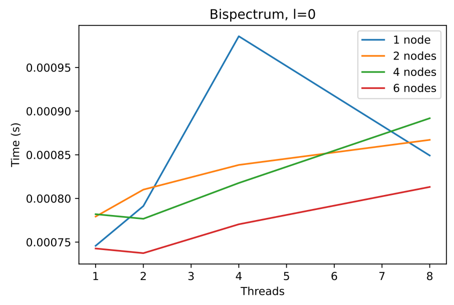
(d)

Fig. 1: Comparison of our three implementations of the change-of-basis algorithm. Each instance of the bispectrum problem here was run on 1 CPU across 1, 2, 4, and 8 threads. Generally speaking, it appears that with the exception of the smallest case ($l = 0$), the parallelized version of the algorithm was faster than the serial divide and conquer algorithm. The original algorithm was fastest in the $l = 0$ case (a) and similar to the parallelized algorithm in the $l = 1$ case (b), but quickly became much slower than either divide-and-conquer algorithm starting at $l = 2$.
Here, "Serial" refers to the original algorithm, "Divide & Conquer" refers to the serial divide and conquer algorithm, and "Parallel" refers to the parallelized divide and conquer algorithm.

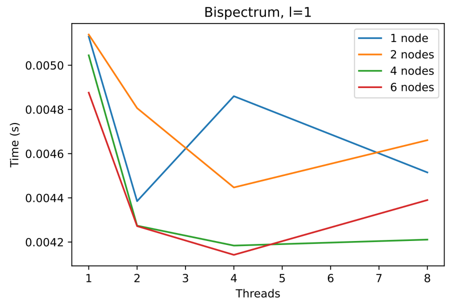
the algorithm on the $l = 4$ bispectrum problem; it was too large for the parallelized algorithm to solve, but was executable using both of the other two algorithms. On the other hand, the $l = 5$ bispectrum problem was too large for any of the three algorithms to solve.

Appendix A. Einstein summation. Einstein summation is a notation convention used to simplify summations of tensors by removing the usual summation symbol ($\sum$) and instead using the tensors' indices. The main rules for Einstein summation are

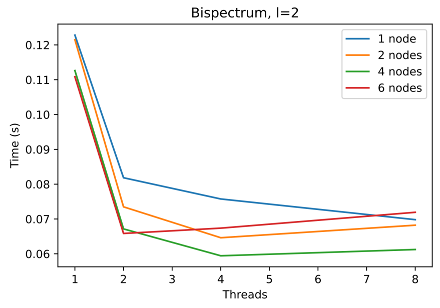
1. Indices that are repeated are summed over.
2. Within each tensor or product of tensors present, any given index can appear at most twice.



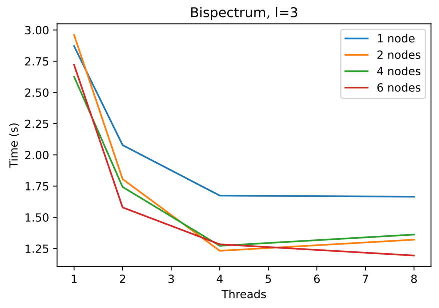
(a)



(b)



(c)



(d)

Fig. 2: Comparison of the parallelized version of our algorithm across 1, 2, 4, and 6 nodes and CPUs (1 CPU per node). Across all test cases, it appears that having 1 node/CPU is consistently slower than having more than one, and that having multiple threads is faster than having just one. However, it also seems that each instance of having more than one node takes a fairly similar amount of time, and that each test case with more than one thread is fairly similar after the initial drop between 1 and 2 threads.

So, for the equation $z_k = x_i y_j C_{ijk}$, the tensors on the right-hand side are summed over the indices i and j to obtain a result with the single index k.

Acknowledgments. I would like to thank Prof. Tess Smidt and Dr. Mario Geiger for their guidance and expertise. All computing for this project was performed on MIT Supercloud.

Source code. The Julia implementations of all the algorithms are located at https://github.com/kikipet/ReducedTensorProduct.jl.

REFERENCES

[1] M. Geiger, T. Smidt, A. M., B. K. Miller, W. Boomsma, B. Dice, K. Lapchevskyi, M. Weiler, M. Tyszkiewicz, S. Batzner, M. Uhrin, J. Frellsen, N. Jung, S. Sanborn, J. Rackers, and M. Bailey, *Euclidean neural networks: e3nn*, 2020, https://doi.org/10.5281/zenodo.5292912, https://doi.org/10.5281/zenodo.5292912.