Tensor Contraction Optimizations

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Abhijit Sringeri Vageeswara

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Master's Examination Committee:

Dr. P. Sadayappan, Advisor

Dr. Atanas Rountev

Abstract

Tensor contractions are frequently encountered computationally expensive operations in the fields of Nuclear Physics, Quantum Chemistry, Fluid dynamics and other areas of computational science. Most often, these operations are executed in a sequence, where the output of one operation is used as input to later operations. Existing parallel solutions are either not communication optimal or do not handle a sequence of contractions optimally. A recently developed RRR framework requires tensors to be distributed in certain specific ways over a certain specific logical views of a processor grid for optimal communication. Redistribution of tensors over changing logical processor grids between successive contractions in a sequence enables it to be executed in a communication efficient manner in the RRR framework.

While these redistributions involve no computation and are just communication operations, they can take a significant amount of the runtime of the contraction operation. In this thesis, an efficient way to redistribute tensors onto changing logical views of multi-dimensional processor grids is presented. A cost model to select the best distribution and grid scheme is implemented and is tested on a sequence of CCSD tensor contraction equations and the results are presented.

A report on some work to optimize the CCSD(T) method of the NWChem software suite is also presented.

Dedication

I dedicate my work to my family and everyone and everything that sparked my interest in computer science.

Acknowledgments

I would like to thank my advisor Dr. P. Sadayappan for giving me the opportunity to be a part of his wonderful research group and for constantly encouraging me to pursue something interesting. I would also like to thank everyone who is a part of his research group; especially Samyam Rajbhandari and John Eisenlohr with whom I had the pleasure of working closely with. I would also like to thank Sriram Krishnamoorthy from PNNL for providing me an opportunity to spend a wonderful summer at PNNL performing research in my field.

I would also like to thank all my family and friends without whose constant support and encouragement, I wouldn’t have remained sane through all of this.

Vita

2006 August – 2010 July	B.E. Information Science, Visveswariah Technological University
2010 August – 2013 July	Solution Engineer, Oracle Corporation
2014 January – 2015 August	Graduate Research Associate, Department of Computer Science, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering

Specialization: High Performance Computing

Table of Contents

Abstract ii

Dedication iv

Acknowledgments v

Vita vi

List of Code Snippets and Algorithms ix

List of Figures x

Chapter 1: Introduction 1

Chapter 2: Background 3

2.1 Tensors 3

2.2 Tensor Contraction 4

2.3 Tensor Distribution 5

2.3.1 Distribution and Serialization 6

2.3.2 No Redundant Mapping 6

2.3.3 Replication 7

2.4 Processor Grids 8

2.5 RRR Framework 9

2.5.1 Distribution of Contraction and External Indices 9

2.5.2 Reduction, Recursive Broadcast and Rotation (RRR) 10

2.5.3 Recursive Broadcast with DDO 11

2.5.4 Reduction with DDA and SD 11

2.5.5 Rotation with DC 12

2.5.6 Cost function based on RRR 12

2.6 Need for Tensor Redistribution 13

Chapter 3: Redistribution of Tensors 15

3.1 Communication 15

3.2 Replication and its effect on communication 25

3.6.1 Replication before redistribution 25

3.6.2 Replication after redistribution 27

Chapter 4: Experiments 30

Chapter 5: NWChem ccsd(t) Optimization 33

Chapter 6: Conclusion and Future Work 37

References 39

Appendix A: List of contractions 40

List of Code Snippets and Algorithms

Listing 1: Code showing simple tensor contraction.. 5

Listing 2: Algorithm showing how senders are determined and data is sent.................... 20

Listing 3: Algorithm showing how send buffers are formed... 21

Listing 4: Algorithm showing how receivers are identified and receive operation is carried out 23

Listing 5: Algorithm showing how number of tiles to receive from different processors are calculated 24

Listing 6: Algorithm showing how replication after redistribution is handled................ 28

Listing 7: One of the loops in ccsd(t) with openmp pragma.. 35

List of Figures

Figure 1: Tensor with block addresses.. 17

Figure 2: Processor ranks and addresses before and after redistribution.......................... 18

Figure 3: Number of tiles in processors before and after redistribution........................... 22

Figure 4: A 2-d <4,4> processor grid with processor ids 26

Figure 5: Execution times with/without redistribution... 31

Figure 6: Tensor contraction equations in ccsd(t).. 34

Figure 7: Execution times of ccsd(t) method for Uracil molecule................................... 36

Chapter 1: Introduction

Tensor contractions are frequently encountered computationally expensive operations in the fields of Nuclear Physics, Quantum Chemistry, Fluid dynamics and other areas of computational science [1]. NWChem is a software suite that makes heavy use of tensor contractions implemented using the Global Array Toolkit to perform these contractions in parallel [2]. However, optimization of inter-processor communication is not addressed. The recently developed RRR framework for tensor contractions provides a solution for this [3]. It uses MPI collective operations to communicate during a distributed tensor contraction. It optimizes the communication operations so as to perform the tensor contraction with a given data distribution on a given logical view of a multidimensional processor grid in the most efficient way possible. When the data is distributed ideally on an ideal processor grid, it performs the most communication optimal contraction.

The solution works as long as there is only one contraction to be performed and we use the logical view of the processor grid and the data distribution on the grid. The problem is that tensor contraction operations are most often run in a sequence where the output of one operation is used as input to the next. This means we have little to no control over how tensors are distributed for subsequent contractions in the sequence and means RRR won't be able to perform the most efficient contraction. Redistribution of tensors over changing logical views of the processor grid between successive contractions in the sequence solves this problem. It allows us to change the way the tensor data is distributed and the logical view of the grid it is distributed on, between each contraction in the sequence. This allows RRR to perform each contraction in the sequence in an optimal manner.

In this thesis, an efficient way to redistribute tensors onto changing multi-dimensional processor grids is proposed. The redistribution is a multistep process in which the replication of tensor data before and after the process determines which steps are to be taken. In cases where there is no replication on either end, the data is accumulated and a point to point communication operation is used to redistribute the data. In cases where there is data replication either before or after the redistribution, an aggregation or broadcast operation is used to take advantage of data replication to reduce communication costs.

Chapter 2: Background

This section describes in brief the various terms and concepts like Tensors, Tensor Contractions, Tensor Distribution, and then briefly explains how the RRR framework works and how redistributing a tensor onto different processor grids can have a positive impact on overall performance of a sequence of tensor contractions.

2.1 Tensors

Tensors like scalars and vectors are mathematical objects used to describe physical properties. In fact tensors are merely a generalization of scalars and vectors; a scalar is a zero dimensional tensor, and a vector is a one dimensional tensor. The dimension of a tensor is defined by the number of directions (and hence the dimensionality of the array) required to describe it.

From a computational and storage perspective, Tensors are just higher dimensional arrays. Just as a matrix M[A x B] is essentially a two dimensional rectangular array with A rows and B columns, tensors are just a higher order generalization of this concept. Whereas a matrix is always defined by two indices, a Tensor is described by as many indices as its dimension.

For representing matrix M of size A x B as a tensor, we need to use two indices representing its dimensions as M[i; j]. Here there are A elements along index i and B elements along index j. Although an order is decided for the indices of a tensor for the purpose of storage, the order does not hold any significance in the physical interpretation of the tensor. A 4D tensor N[i; j; k; l] has four indices namely i, j, k and l and the tensor can also be represented as N[k; i; l; j] or any other permutation of the indices. However, the storage of the tensor only follows one of all the possible permutations.

2.2 Tensor Contraction

Tensor contractions are higher dimensional analogues of matrix-matrix products. Consider the multiplication of A [M _K] and B [K _N] to generate an output matrix C [M_N]. Here, an element of C at the intersection of ith row and jth column which is identified as C[i][j] (Note: C[i][j] is used to denote an element where I and j mean values of the respective indices, while C[i; j] is used to denote the tensor where i and j indicate the indices of the tensor) is a dot product of ith row of A and jth column of B. This matrix multiplication also represents a contraction of tensors A[i; k] and B[k; j] that contracts the index k and yields an output tensor C[i; j]. From the indices of the tensors, it can be observed that index i of A and the index j of B are retained in C after the contraction. These are called as \external indices" of the input tensors. However, the index k of both input tensors is contracted and thus does not appear in the output tensor. It is termed as a “contracting index" of the input tensors.

In higher dimensional tensor contractions, there can be more than one contracting indices in each input tensor. For example, contraction of two tensors: A[a; b; k; l] and

B[l; k; c; d] will yield an output tensor C[a; b; c; d] such that the contraction indices k and l are contracted, while the indices a, b, c and d as external indices are retained in C.

The code snippet below shows the contraction between two tensors A[a;b;c;d] and B[d;l] to form the output tensor C[a;b;c;l]. Here the indices a,b,c of tensor A and index l of tensor B are external indices since they appear on the output tensor and index d of tensor A and tensor B is the contracting index.

```
for(int a=0; a<n; a++)
    for(int b=0; b<n; b++)
        for(int c=0; c<n; c++)
            for(int d=0; d<n; d++)
                for(int l=0; l<n; l++)
                    C[a,b,c,l] += A[a,b,c,d] * B[d,l];
```

Listing 1: Code showing simple tensor contraction

2.3 Tensor Distribution

Contraction of tensors can be parallelized on a cluster of processor nodes with distributed memory. The processor grid considered for this study is a multi-dimensional torus. To perform parallel computation of the contraction, the tensor needs to be distributed onto the processor grid. A tensor can be stored in a processor grid of the same or different dimensionality as that of the tensor. Usually, tensor indices are mapped to the grid dimensions in order to distribute the tensor. This mapping is referred to as “index-dimension mapping". It is represented by a vector of the same dimensionality as the tensor where the value at i^{th} dimension represents the physical grid dimension the i^{th} index of tensor is mapped to. There are certain constraints and rules to distributing a tensor onto a grid which are discussed below [3].

2.3.1 Distribution and Serialization

Each dimension of the tensor must be either distributed along some dimension of the torus or serialized. An index i of the tensor is said to be distributed along dimension p of the grid if each node along p holds a range of index values along i. An index i of the tensor is said to be serialized if every node in the grid along the dimension p holds all the index values along i.

2.3.2 No Redundant Mapping

Redundant mapping here refers to mapping two different dimensions of the tensor onto the same dimension of the processor grid. This isn't allowed. Each set of elements along any dimension of a tensor can be viewed as a Cartesian product between each index of a tensor. Say we have a tensor of two dimensions A[i;j], it isn't possible to have the entire set of product set of elements if both the tensor dimensions are mapped to the same processor dimension. If both i and j indices are mapped along the same processor dimension and are distributed in the same way, the only elements that now make sense are those where i=j. It is not possible to store any other elements since each node along the dimension cannot have different values for i and j.

2.3.3 Replication

From the perspective of the processor grid, if a tensor dimension is mapped onto a processor grid dimension, then that means each processor along that dimension holds a range of values of the tensor. On the other hand, if no dimension of the tensor is mapped to a grid dimension, then it means all the processors along that dimension hold all of the tensor data. In other words, tensor data is replicated along the processors in that dimension.

2.3.4 Examples of Tensor Distribution

Let us go through a few example with their index dimension mapping provided to get familiar with the concept.

1. All dimension distributed with no replication: Consider the example where a d-dimensional tensor is distributed along d different dimensions of a d-dimensional torus grid. Here, there is a one to one mapping and there is no replication or serialization. Consider tensor T[a;b;c;d] distributed along a 4-dimensional processor grid. There are a total of 4*3*2*1 total ways of distributing this with given constraints. Consider one such example where in index a is mapped to dimension 0, index b along dimension 1, index c along dimension 2, and index d along dimension 3 of the grid. The index dimension mapping in this case is <0,1,2,3>.
2. Some dimensions distributed and some serialized with no replication: Consider the example where in a 4-dimensional tensor T[a;b;c;d] is distributed on a 3-

dimensional grid with three of the tensor dimensions distributed along three dimensions of the processor grid. The fourth dimension of the tensor is serialized. Let's say indices a,b,c are distributed along dimensions 0,1,2 and index d of the tensor is serialized. The index dimension mapping would be <0,1,2,serial>.

3. Some dimensions distributed and some serialized with replication: This case is similar to the previous one except that there are dimensions of the processor grid that isn't mapped to any dimension of the tensor. Consider a 4-dimensional tensor T[a;b;c;d] distributed on a 4-dimensional grid with indices a,b,c distributed along dimensions 0,1,2 with index d being serialized. The index dimension mapping would be <0,1,2,serial>. Here, dimension 3 of the grid has no mapping and hence it doesn't appear in the index dimension map and the tensor data is replicated along that dimension.

2.4 Processor Grids

It is often convenient to have a logical organization of all the processors involved in the distributed tensor contraction. It helps in easy distribution of data, formation of broadcast groups during computation. The grid considered in this study is a torus. A torus shaped grid has nodes linked in series in each dimension with a link between the two extreme nodes. In a d-dimensional torus, every node has two neighbor nodes in each dimension and total 2 x d neighbors. Each processor in the grid can be identified by its d-dimensional coordinates. Thus for a 3D grid, a processor P[i;j;k] is the ith processor in

0th dimension, jth in the 1st dimension and kth in the 2nd dimension, where dimension indexing starts at 0.

Torus networks are especially suited to tensor contractions because it allows for efficiently shifting data along a dimension or rotating the data along a dimension which are frequently used in tensor contraction algorithms.

Now, a given number of processors can be logically grouped to form many different torus networks. Let us consider an example of sixty four processors. The 64 processors can be construed as many different torus grids. Let us consider a few of them.

1. 2-dimensional grid with <8,8> layout. Here, there are only two dimensions and there are 8 processors along dimension 0 and 8 along dimension 1.
2. 3-dimensional grid with <4,4,4> processors. Here there are three dimensions with four processors along each dimension.
3. 4-dimensional grid with <2,2,4,4> layout. Here there are four dimensions with two along dimensions 0 and 1, and four along dimensions 2 and 3.

2.5 RRR Framework

This section briefly describes the communication operations used in RRR framework and how tensor distribution has an effect on which operations are chosen.

2.5.1 Distribution of Contraction and External Indices

Let k be a contraction index then k_A and k_B are corresponding contraction indices in input tensors A and B. k_A and k_B can each be either serialized on every node of the torus or distributed along some dimension of the torus. All different possible distributions of k_A and k_B are: Distributed, Distributed - Aligned (DDA), Distributed, Distributed - Orthogonal (DDO), Serialized, Distributed (SD) or vice versa, Serialized, Serialized (SS). DDA refers to the case when both k_A and k_B are distributed along the same dimension of the torus and DDO refers to distribution along separate dimensions.

Let e_A be an external index in A. e_A can be either serialized on every node of the torus or distributed along some dimensions. All different possible mappings of e_A are: Distributed-Conflicting (DC), Distributed-Exclusive (DE), Serialized (S). e_A is Distributed-Exclusive if no external index of B is distributed along the same dimension of torus as e_A, otherwise it is Distributed-Conflicting (DC).

2.5.2 Reduction, Recursive Broadcast and Rotation (RRR)

Let p_x and p_y be two dimensions of p-dimensional torus and n_x and n_y be the number of processors along p_x and p_y. Let P(x; y) represent a general node whose coordinates are x along p_x and y along p_y.

Distribution of a contracting iterator K along dimension p_x of the torus implies that the mapping produces partial results along p_x. The output tensor has to be combined along p_x to obtain final result. This communication operation for combining partial results of output tensor along a dimension p_x is referred to as Reduction.

If an iterator I is serialized, but the corresponding index i_A in an input tensor A is distributed along a dimension p_x, then A has to be broadcasted along p_x. If I is a contracting iterator and there are multiple such serialized iterators with distributed indices, we will call the broadcast operation on A as Recursive Broadcast.

Similarly if I is an external iterator, then we call the broadcast operation as Rotation. Rotation simply means that A is rotated along the dimensions, resulting in an efficient pipelined all-to-all broadcast.

2.5.3 Recursive Broadcast with DDO

Recursive Broadcast corresponds to serialization of the contracting iterator. Let contraction index k_A and k_B be distributed along processor dimension p_x and p_y respectively. Let $k^m{}_A$ and $k^m{}_B$ be range of values of k_A and k_B held at processor P(m; y) and P(x;m) respectively. Notice that except for the diagonal processors P(m;m) the k_A and k_B data are not aligned. In order to contract k^m, $k^m{}_A$ and $k^m{}_B$ initially held at P(m; y) and P(x;m) need to be held by all nodes. To do this P(m; y) can broadcast $k^m{}_A$ along p_x to P(*; y) and P(x;m) can broadcast $k^m{}_B$ along p_y to P(x; *). Now every processor P(x; y) holds $k^m{}_A$ and $k^m{}_B$ and a local contraction k^m can be performed. This is done for each m so that entire contraction index k is contracted locally on each node.

2.5.4 Reduction with DDA and SD

Reduction corresponds to distribution of the contracting iterator. Let contraction index k_A and k_B be distributed along processor dimension p_x. Let $k^m{}_A$ and $k^m{}_B$ be the range of values of k_A and k_B held at processor P(m; y). Notice that $k^m{}_A$ and $k^m{}_B$ are perfectly aligned on each node, i.e. at node P(m; y), km can be contracted since it has both $k^m{}_A$ and

k^m_B . After the local contraction each node along p_x i.e nodes P(*; y) will hold partial result which can be summed using a Reduction. Notice that Reduction can also be used for contraction indices that are Serialized-Distributed. Without loss of generality if k_A is serialized and k_B is distributed along p_x, then the nodes P(m; y) that hold k^m_B also hold k^m_A since k_A is serialized.

2.5.5 Rotation with DC

Rotation corresponds to serialization of external iterator. Let external indices e_A and f_B be distributed along p_x of the torus. Hence, e_A and f_B are conflicting. Let e^m_A and f^m_B be ranges of e_A and f_B held at some processor P(m; y). Under such distribution there is no node which holds e^m_A and f^m_B where $m \neq n$. In other words a full cartesian product between e_A and f_B is not formed. Since e_A and f_B are external indices that appear in output tensor C, these indices cannot be aligned and a full cartesian product needs to be formed between them. This can be done by rotating either A or B along p_x. Without loss of generality, during each step of rotation of B a processor P(m; y) will recieve f^{m-1}_B from P(m-1; y) and will send f^m_B to processor P(m + 1; y). Hence, after n_x steps of rotation, f_B will be completely serialized on each node.

2.5.6 Cost function based on RRR

Given a processor grid, an index dimension mapping, and a suitable tensor distribution, the scheme described above can be shown to use optimal communication [3], with the total cost given by the following equation.

Total cost = num_rotation * (cost(recursive_broadcast + local_compute)+ cost(reduction) + ts + tw * m)

2.6 Need for Tensor Redistribution

We have seen how a single tensor can be distributed in many different ways onto a single processor grid and we have seen how a fixed number of processors can be logically organized into many different torus grids. We see from the previous section how both of those aspects have an impact on which operations are performed and ultimately on performance. So, for the most efficient tensor contraction operation, the tensors involved must be distributed in a certain specific way on a certain specific processor grid. This is not much of an issue if there was only one tensor contraction to be performed. But most often, tensor contractions are performed in sequence meaning outputs from a contraction operation are used as inputs in subsequent contraction operations in the sequence.

Now, what might be the most efficient distribution and the most efficient grid for the first contraction may not be the same for subsequent contractions. But we already have the output tensor distributed a certain way on a certain grid. This is where a redistribution operation is used. It takes the tensor, a new processor grid and a new index mapping as input and the redistributes the tensor onto the new grid according to the index mapping specified. This way, we can perform each tensor contraction in the sequence in the most efficient way available. Even though the redistribution operation adds extra time from communication operations, its presence reduces the total contraction time by a margin enough to justify a need for them.

Consider a simple case presented where the below two contractions need to be done in sequence.

1. A[i;j;k;l] = B[i;j;m;n] x C[m;n;k;l]
2. X[i;j] = A[i;j;k;l] x Y[k;l]

The first contraction involves all 4-dimensional tensor and the second involves two 2-dimensional tensors and the output of the first contraction. Performing both the above contractions on a single grid either 2-d, 3-d or 4-d is not efficient. It leads to either redundant computation when done on a 3-d or a 4-d grid because the data to be computed will be replicated or extra communication during contractions on a 2-d grid. The most optimal way to perform the above sequence of contractions is to perform the first one on a 4-dimensional grid and then redistribute the output tensor A onto a 2-dimnsional grid to perform the next contraction. This way we will be performing both the contractions in the most optimal way with the drawback that we are adding extra communication times due to redistribution. But, generally even with the extra communication operations due to redistributions there is gain in overall performance.

Chapter 3: Redistribution of Tensors

The redistribution algorithm takes the tensor to be redistributed, the new processor grid, the new tensor index mapping as input and redistributes the tensor according to the new index mapping onto the new processor grid. Tensor redistribution is an important operation that may need to be carried out on all tensors between each successive contraction in a sequence. The frequency with which they need to be carried out necessitates the development of an efficient algorithm that can handle it.

The algorithm uses the current processor grid and index mapping as well as the new ones to determine where data needs to be moved to in order for the tensor distribution to change. It uses point to point communication whenever there is a unique source and destination of data. If there are multiple sources or destinations due to replication before or after redistribution, optimizations are applied to ensure all processors are used during communication and total communication time is reduced.

3.1 Communication

Redistribution is a communication operation that involves changing the data distribution of a tensor meaning tensor data that stored in a processor before and after redistribution are different. So, the first step of redistribution involves identifying which processors act as senders and which act as receivers for the redistribution process. Before understanding how that is achieved, knowledge how tensor and grid is addressed is important.

Tile Address

Each block of the tensor data stored in the processor grid has a global address which is a vector with the same dimension as the tensor. It specifies where that block of tensor belongs to in the entire tensor. This address is used during the actual contraction process to identify which tensor block to use during communication and computation. It is important to note that this address does not change during a redistribution as only the local address assigned to a tensor block within a processor is changed. So, these addresses should also be communicated along with the actual tensor data when redistribution occurs.

Let us consider an example of a 3-dimensional tensor with 3 blocks along each dimension. This would mean a total of 27 blocks with address ranging from (0,0,0) to (2,2,2) as shown in figure 1.

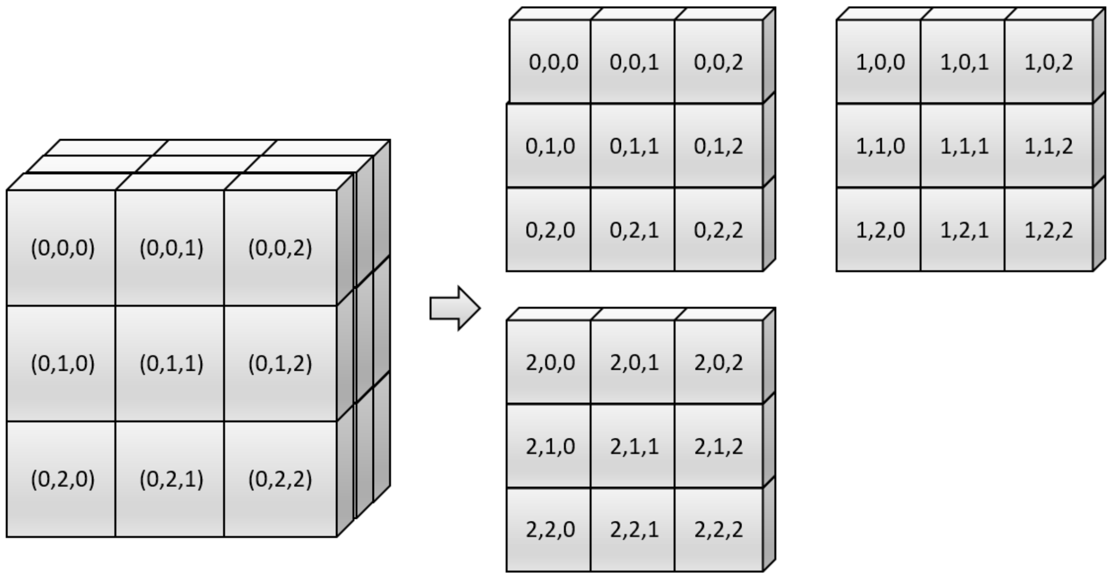

Figure 1: Tensor with block addresses

The index value in the tensor address changes as we move along a dimension. It's important to note that the fastest changing dimension is the one whose index value changes first and is the dimension whose values are stored contiguously inside the block.

Processor Address

Each processor in the grid is also assigned an address depending on its position in the processor grid. This is separate from the rank that is assigned by the MPI process. This address tells us the position of the processor in the virtual grid. This address is represented by a vector with the same dimension as the processor grid. It works the same was as a tensor global address but it's important to note that the processor address changes during a redistribution. This means that the processor with the same rank will have a different address before and after a redistribution if the grid changes. The processor address is always a function of the rank and the grid layout and it's the layout that changes during redistribution.

Consider an example where an 8 processor grid with a <4,4> grid layout with processor addresses ranging from (0,0) to (3,3) is changed to a grid with <2,2,2> grid layout with processor address ranging from (0,0,0) to (1,1,1). Figure 2 shows the grids before and after redistribution with processor ranks and addresses of each processor.

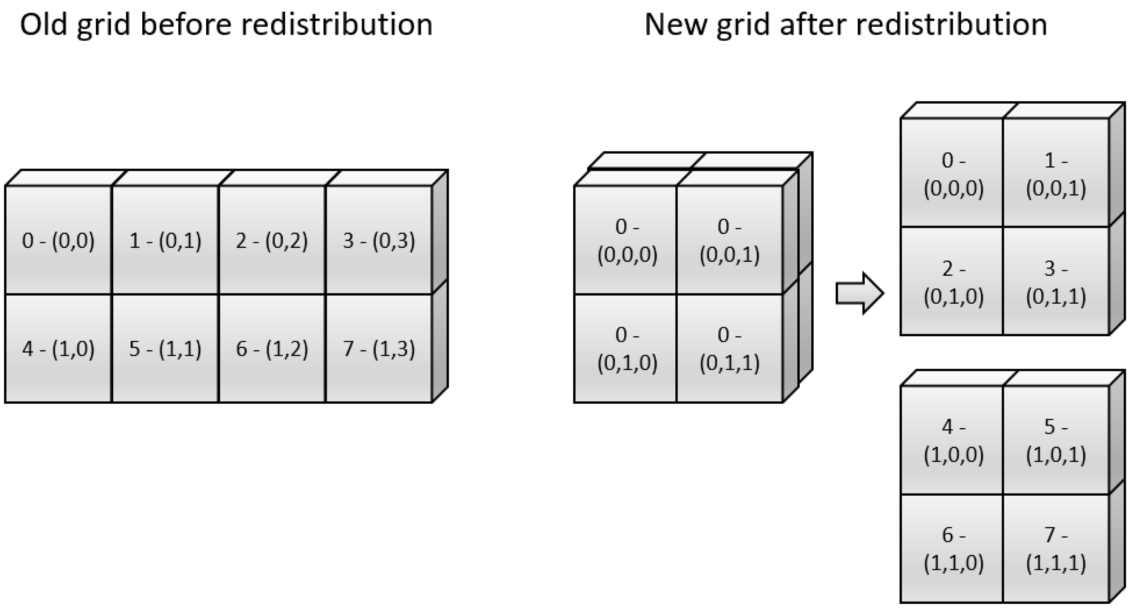

Figure 2: Processor ranks and addresses before and after redistribution

Address changes during redistribution

The position of a particular block of tensor in terms of which processor it belongs to is a function its tile address, the index mapping and processor address. When a redistribution occurs, the index mapping and the processor address are both changed. This means that blocks of tensor that are presently in a processor will need to be moved to a new processor.

Identifying Senders

We need to determine where the data that is stored in the current processor will need to be stored after redistribution. This involves calculating the new processor address for each tensor tile in a processor. The new processor address of a tile along an index is calculated by a modulo operation on the tile address along that index with the number of processors that index is mapped to in the new processor. Calculating this for all indices of a tensor will give the address of the processor that tile needs to be sent to.

Now we know where each block of tensor will end up in after redistribution is done. If the new processor calculated is the same as the old one, no action needs to be taken. All the other blocks need to be sent to their new processor. However, all the blocks that need to go to a given processor is not contiguous in memory. This would result in a large number of send operations essentially one for each block that needs to be communicated resulting in a huge overhead. To overcome this, number of blocks that are to be sent to each of the processor is counted and two buffers are created for each of the processor the size of which will be proportional to the number of blocks that are to be sent to that processor. Two buffers are needed for each processor because one will hold the global address of the block and the other is for the actual data. Now, the data to be moved and their global address is copied onto the corresponding buffers of their corresponding processor. Now, all the data that is to be communicated to a processor is present in that processor's two buffers. These buffers can now be sent out to the relevant processor. So, a non-blocking send operation is used. The whole process is summarized in the algorithm show in Listing 2 and Listing 3.

```
repl_dims: list of replicated dimensions
dims: number of dimensions of tensor to be redistributed
new_proc_addr: processor address of a tile after redistribution
new_idx_map: index dimension mapping after redistribution
tile_address: Array with global tensor tile address for all tiles
tensor_data: Array with tensor data for all tiles
new_grid: Grid layout after redistribution


grid_redistribute_send():

        //Calculate the new processor address for all tiles in a tensor
        for i in range(0,num_tiles)
                for(j in range(0,dims))
                        new_proc_addr[i][new_idx_map[j]]=tile_address[i][j] %
                                                            new_grid[new_idx_map[i]];

        //Initiate sent flag for all tiles
        for i in range(0,num_tiles)
                sent[i] = false;

        for i in range(0,num_tiles)
                if sent[i] == false
                        num_blocks = get_blocks_for_proc(i, new_proc_addr, sent,
                                        data_buf, address_buf);

                        if(num_blocks > 0)
                                //Check if sender and reciever are the same
                                if(get_global_rank(old_grid,old_proc_addr[i]) ==
                                  get_global_rank(new_grid,new_proc_addr[i]))
                                        local_data.add(data_buf);
                                        local_tile_address.add(address_buf);
                                        local_num_tiles.add(num_blocks);
                                //else if they are different
                                else
                                        reciever=get_replicated_proc_ranks(new_proc_addr)
                                        Isend(num_blocks,address_buf,reciever)
                                        Isend(num_blocks, data_buf,reciever)
```
Listing 2: Algorithm showing how senders are determined and data is sent

```
get_blocks_for_proc(index, ranks, sent, blocks, block_addrs):

        count=0;

        //Count number of tiles to send to a processor
        for i in range(0,num_tiles)
                if ranks[i] == ranks[index]
                count++;

        //Initiate buffers
        blocks = new double[count * get_block_size()];
        block_addrs = new int[count * dims];

        //Pointers to the two buffers
        double* out_blocks = blocks;
        int* out_block_addrs = block_addrs;

        for i in range(0,num_tiles)
                if ranks[i] == ranks[index]
                        // Copy block
                        memcpy(out_blocks, tensor_tile[i], get_block_size() *
                                                                    sizeof(double));
                        out_blocks += get_block_size();
                        // Copy address
                        memcpy(out_block_addrs, tile_address[i], dims * sizeof(int));
                        out_block_addrs += dims;
                        sent[i] = true;

        return count;
```
Listing 3: Algorithm showing how send buffers are formed

Number of tiles in a processor

The number of blocks of tensor that each processor holds is not the same across a grid. It is a function of both the index dimension mapping and the processor address. Consider the example of a 2-d tensor A[i;j] with 4 blocks along i and 4 blocks along j. If it's distributed on a 8 processor 2-d grid with grid layout <2,4> and an index dimension mapping of <0,1> meaning index i is distributed along dimension 0 and index j is distributed along index 1. Now, each processor has a total of two blocks. If that tensor needs to be distributed onto a 3-dimensional grid with grid layout <2,2,2> with an index dimension mapping of <0,1>, then the data is replicated along dimension 2 of the grid. This means that each processor now has four blocks as shown in figure 3.

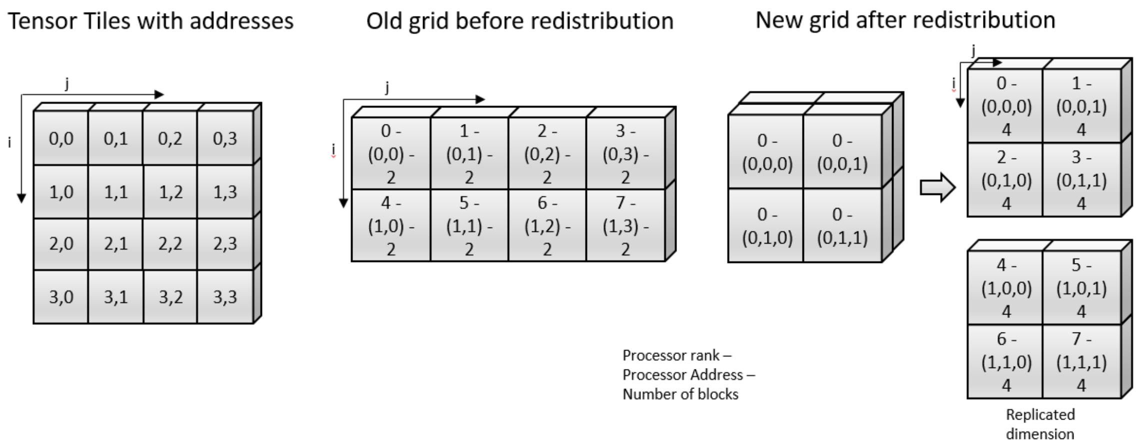

Figure 3: Number of tiles in processors before and after redistribution

So, clearly the number of blocks in a processor depends on the grid layout and index dimension mapping. Since both of these will be changing during redistribution, the number of blocks that a processor holds after redistribution needs to be calculated before receives can be posted. This is done by counting the number of tiles with the new processor address starting from the fastest moving dimension of the tensor address and moving onto next fastest and so on till all of the dimensions are covered.

```
new_tile_num: number of tiles in current processor after redistribution
old_proc_addr: array to hold which processors data was in before redistribution
tile_address: array holding global address of all tensor blocks
old_idx_map: index dimension mapping before redistribution
old_pgrid: grid layout before redistribution
recv_data: data structure to hold data received from different processors


grid_redistribute_recv(/*out*/recv_data):

        //Calculate number of blocks this processor will hold after redistribution
        new_tile_num=get_tile_num(new_idx_map, new_grid, new_proc_addr);

        //Calculate the address of those blocks before redistribution
        for i in range(0,new_tile_num)
                for(j=0 to dims-1)
                        old_proc_addr[old_idx_map[j]] = tile_address[i][j] %
                                                                    old_pgrid[old_idx_map[j]];

        // Find how many blocks will be received from which processor
        num_procs = get_num_recv_blocks(old_proc_addr, new_tile_num, map);

        for i in range(0,num_procs)
                int num_tiles = map[i];

                if num_tiles > 0

                     //Check if sender is same as reciever
                     if(get_global_rank(old_grid,old_proc_addr[i]) ==
                       get_global_rank(new_grid,new_proc_addr[i]))

                         recv_data.push(local_data,local_tile_address,local_num_tiles);

                     else
                        //Initiate recieve buffer
                        rd.blocks = new double[num_tiles * get_block_size()];
                        rd.block_addrs = new int[num_tiles * dims];
                        rd.num_blocks = num_tiles;
                        sender=get_replicated_proc_ranks(old_proc_addr);
                        Irecv(num_tiles,rd.block_addrs,sender);
                        Irecv(num_tiles,rd.blocks,sender);
```
Listing 4: Algorithm showing how receivers are identified and receive operation is carried out

```
get_num_recv_blocks(old_proc_addr, new_tile_num, map):

        // Find total number of processors
        num_procs = 1;

        for i in range (0,old_grid_dims)
             num_procs *= old_pgrid[i];

        // Declare and initialize map of size of number of processors
        map = new int[num_procs];
        for i in range (0,num_procs)
                map[i] = 0;

        for i in range(0,new_tile_num)
                map[ranks[i]]++;

        return num_procs;
```
Listing 5: Algorithm showing how number of tiles to receive from different processors are calculated

Identifying Receivers

Now we have the number of blocks each processor will hold after redistribution. Now we need to find out which processors they were stored in before the redistribution. This is done the same way using the old processor grid and index dimension mapping to calculate addresses of blocks that will end up in current processor after redistribution.

We have the total number of blocks of data and their old processor addresses. We then determine how much blocks of data is to be received from each individual processor and create two buffers to store the received addresses and data. Then non-blocking receives are posted. The whole process is summarized in Listing 4 and Listing 5.

Rebuilding Local Index

Since we used, non-blocking communication, all sends are posted together and then all receives. After the exchange of data is complete, we have the new tensor blocks and their global address. But the data and address are in chunks in the form of buffers. They are copied onto the correct data structure in the tensor. A local index for the blocks is also needed; it is rebuilt using the global address of the blocks. After this, the tensor is now ready for further operations.

3.2 Replication and its effect on communication

Data replication implies that the same data is present in multiple processors. This can be taken advantage of to optimize communication. Since, the same data is either already present in multiple processors or needs to go to multiple processors, we can cut down on the communication cost further. Based on that, there are two scenarios present. The first step in both scenarios involves finding the replicated dimensions which is simply a matter of finding the grid dimensions not present in the index dimension mapping and keeping a count of it.

3.6.1 Replication before redistribution

In this case, there are dimensions of processors to which no tensor dimension is mapped to before redistribution. This means there is replicated data along those dimensions. Determining these replicated dimensions is a simple function of the old processor grid and the old index dimension mapping. So, instead of posting sends and receives from the same processor, whenever data is needed from a processor, all processors along the replicated dimension of that processor is logically treated as the same processor and a function is used to spread this to all the processors along that dimension in a cyclic way. This way, the load on each individual processor is reduced and communication time is reduced.

In case of multiple replicated dimensions, the same logic applies but the pool of processors from which data can be exchanged from is further increased by including processors along all the replicated dimensions.

Let us consider an example of a case wherein a 2-d tensor A[i;j] is distributed along a 2-d grid with index i distributed along dimension 0 and index j being serialized before redistribution. So, data is replicated along dimension 1.

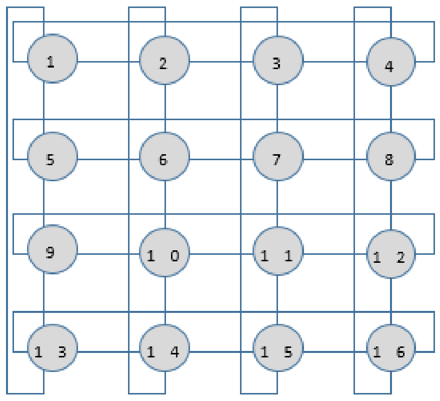

Figure 4: A 2-d <4,4> processor grid with processor ids

The above figure shows the processor grid and since data is distributed along dimension 0, processors with labels 1,2,3,4 all have different data. Now, since data is replicated along dimension 1, processors with label 1,5,9,13 all have the same data and so on. Instead of sending the data multiple times from different processors, data is communicated only once with senders being cycled between all processors that hold same data. Here, whenever a block from processor 1 needs to be sent, the sending processor is cycled between all those four processors so as to balance out the load with a map maintained so as to ensure the correct receives are posted.

3.6.2 Replication after redistribution

In this case, there are dimensions of processors to which no tensor dimension is mapped to after redistribution. This means there is replicated data along those dimensions. Determining these replicated dimensions is a simple function of the new processor grid and the new index dimension mapping. So, now instead of having to receive data onto a single processor, it can be received by all the processors along the replicated dimension. After this receive, a communicator group is formed which includes all processors along the replicated dimension and then an all-to-all broadcast is performed. This way, all the processors along the replicated dimension end up having all the data.

It works the same way in case there are multiple replicated dimensions. All processors along all replicated dimensions receive a part of the data and then a communicator group is formed which includes all those processors and then an all-to-all broadcast is performed as summarized in algorithm in Listing 6.

```
get_replicated_dims(idmap, repl_dims, grid_dims)
        mapped = new int[grid_dims];
        memset(temp, 0, grid_dims*sizeof(int));

        count = 0;

        for i in range(0,grid_dims)
                mapped[idmap[i]] = 1;

        for i in range(0,grid_dims)
                if mapped[i]=0
                        count++;

        repl_dims = new int[count];
        j = 0;
        for i in range(0,grid_dims)
                if mapped[i]=0
                        repl_dims[j++] = i;

        return count;

replicate(int rep_dim)
        // Create a new MPI_Communicator in the replication dimension
        num_procs = new_pgrid[rep_dim];
        dim_group_ranks = new int[num_procs];
        for in in range(0,num_procs)
                new_proc_add[rep_dim] = i;
                dim_group_ranks[i] = get_proc_rank(new_grid,new_proc_add)

        MPI_Comm_group(new_grid->grid_comm, &orig_group);
        MPI_Group_incl(orig_group, num_procs, dim_group_ranks, &new_group);
        MPI_Comm_create(new_grid->grid_comm, new_group, &new_comm);

        count[proc_rank] = get_num_tiles();
        All_All_Bcast(count, new_comm);

        All_All_Bcast(tile_address,new_comm);
        All_All_Bcast(tile_data,new_comm);
```
Listing 6: Algorithm showing how replication after redistribution is handled

Let us consider a 2-d tensor A[i;j] with mapping <0,serial> after redistribution. This means the processors along dimension 1 should have replicated data after redistribution.

Processors with label 1,5,9,13 should now have the same data after redistribution and so on. Now, instead of all the processors receiving the same data, data is communicated only once with the receiver cycled between all processors that should hold the same data. In this case, data that should be received on processor 1 is distributed so that processors 1,5,9,13 receive a part each of the data in a cyclic way. After all the data is received, a communicator is formed containing those processors and an all-to-all broadcast operation is done to ensure that all the processors end up having all of the data.

Chapter 4: Experiments

One of the widely used methods in Quantum Chemistry is the Coupled Cluster family of methods. Used for modeling many-body systems in chemistry, coupled cluster methods are computationally expensive and often require computational power of supercomputers. With the benefit of high performance computing, multi-electron wave functions can be more accurately modeled for molecules. The types of coupled cluster methods are decided by the number of excitations permitted. Coupled Cluster Doubles (CCD) applies for only double excitations, Coupled Cluster Singles and Doubles (CCSD) applies for single and doubles excitations.

In CCSD, using algebraic and diagrammatic techniques, a sequence of equations is derived. These equations involve operations on tensor objects generally addition and contractions. For our experiment, we considered once such sequence of equations. The full list is presented in Appendix A.

The equations were translated to code and an ad-hoc cost function was developed to determine the best grid layout and index dimension mapping for these contractions. The cost function considers the contraction and the number of processors available and first forms a grid of appropriate dimensionality and then determines and valid and optimal index dimension mapping. This is done considering the cost equation of RRR described earlier and assigning an index mapping that minimizes this cost.

The sequence of contractions were run with and without grid changing redistribution and the execution times were noted. The portion of time used for redistribution is also noted. The experiments were run on a cluster with 128 cores of Intel Xeon E5640 CPU running at 2666 MHz. There are 8 cores per node with a total memory of 16GB per node.

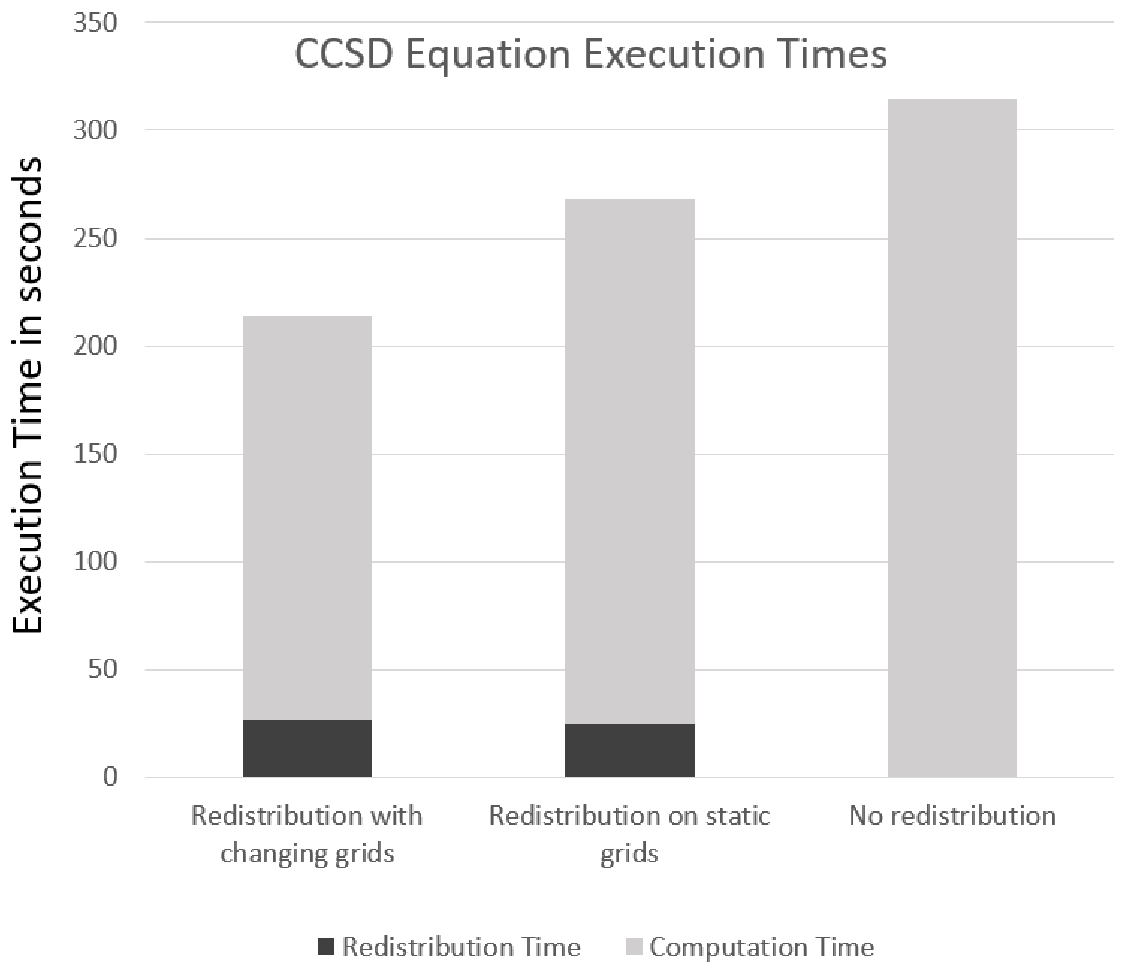

Figure 5: Execution times with/without redistribution

There were a total of 183 contractions that were run in sequence consisting of 2 dimensional and 4-dimensional tensors with tensor sizes ranging from 16 to 128 elements along each dimension.

The graph clearly shows the advantage of redistribution. Even while redistributing on a single static grid, the total execution time was lowered from 314.7 seconds to 267.9 seconds out of which 24.3 seconds were spent on redistribution. For redistribution with changing grids, the redistribution time was slightly more at 26.4 seconds but the total execution time is further reduced to 213.7 seconds.

Chapter 5: NWChem ccsd(t) Optimization

NWChem provides many methods for computing the properties of molecular and periodic systems using standard quantum mechanical descriptions of the electronic wave function or density. Its classical molecular dynamics capabilities provide for the simulation of macromolecules and solutions, including the computation of free energies using a variety of force fields. These approaches may be combined to perform mixed quantum-mechanics and molecular-mechanics simulations.

Coupled Cluster theory has evolved into a widely used and very accurate method for solving the electronic Schrödinger equation. The methods such as the ubiquitous CCSD(T) approach enable precise predictions for the molecular structure, inter-molecular interactions, transition states, and activation barriers. This is an extremely important method and has an extremely high numerical complexity of N^6 where N symbolically represents the system size [2].

The most computationally intense and time consuming part of the ccsd(t) method involve tensor contraction equations of the form shown in figure 5.

```
R[a,b,c,i,j,k] -= T[l,k,c,b]*V[l,a,i,j]
```

$$\textbf{sd2\_1}: t3[h3, h2, h1, p6, p5, p4] -= t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_2}: t3[h2, h1, h3, p6, p5, p4] -= t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_3}: t3[h2, h3, h1, p6, p5, p4] += t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_4}: t3[h3, h2, h1, p6, p4, p5] += t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_5}: t3[h2, h1, h3, p6, p4, p5] += t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_6}: t3[h2, h3, h1, p6, p4, p5] -= t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_7}: t3[h3, h2, h1, p4, p6, p5] -= t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_8}: t3[h2, h1, h3, p4, p6, p5] -= t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$
$$\textbf{sd2\_9}: t3[h2, h3, h1, p4, p6, p5] += t2[p7, p4, h1, h2] \times v2[p7, h3, p6, p5]$$

Figure 6: Tensor contraction equations in ccsd(t)

This part of the method acts as a bottleneck and it can be optimized. The idea was to use multi-threaded dgemm along with openmp to speedup these operations so that there is an improvement in the overall performance.

One such optimization is shown in table 4.

```
#pragma omp parallel for collapse(6)
for (h3=0; h3<= range_h3-1; h3++)
{
   for (h1=0; h1<= range_h1-1; h1++)
         {
                  for (h2=0; h2<= range_h2-1; h2++)
                  {
                           for (p4=0; p4<= range_p4-1; p4++)
                           {
                                    for (p5=0; p5<= range_p5-1; p5++)
                                    {
                                             for (p6=0; p6<= range_p6-1; p6++)
                                             {
                                                               //Contraction code

                                              }
                                    }
                           }
                  }
          }
}
```
Listing 7: One of the loops in ccsd(t) with openmp pragma

Since this is a perfectly nested rectangular loop with no loop carried dependencies, the openmp collapse directive can be used to further improve performance. The collapse clause is used to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread. Since the amount of work to be done by each thread is non-trivial, this improves the parallel scalability of the application.

In case of loops that were not perfectly nested, care was taken to reduce the granularity of the work to be done by each thread as much as possible by placing the openmp pragma as far from the innermost loop as possible.

5.1 Experimental Setup and Benchmark used.

NWChem version used was release version 6.5. The native Fortran code of NWChem was first converted to C before OpenMP pragmas were applied. Intel MKL was used as the BLAS library. The experiments were run on 96 cores of Intel Xeon E5640 CPU running at 2666 MHz. There are 8 cores per node with a total memory of 16GB per node. The Benchmark used was the equations available for Uracil molecule in NWChem.

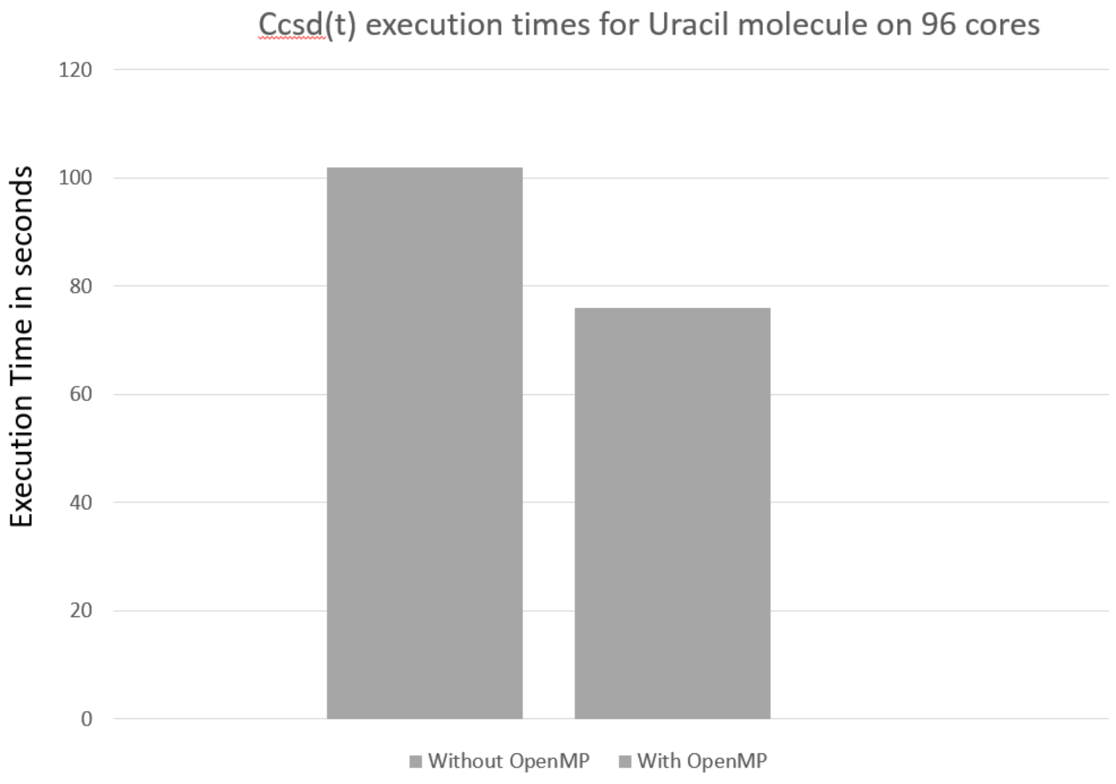

Figure 7: Execution times of ccsd(t) method for Uracil molecule

Speeding up the slowest parts of the code using OpenMP has reduced the execution time by about 25%.

Chapter 6: Conclusion and Future Work

The redistribution algorithm presented can redistribute the tensor data to any index dimension mapping and grid layout we chose. It exploited data replication either before or after redistribution to parallelize communication, thereby reducing redistribution time. It solved the problem of how the RRR framework can perform each contraction in a sequence of contractions in a communication-optimal way by allowing us to redistribute the data distribution between each successive contractions in the sequence in a way that allows the RRR framework to perform the entire sequence of contractions in a communication optimal way.

The algorithm was tested out by having RRR perform a sequence of contractions with and without the redistribution. The results clearly show that the communication time spent on redistribution is worth it as the total execution time is reduced by a significant margin.

Even though the algorithm is quite flexible and can redistribute any tensor on any given changed view of the grid, it can handle only one grid at a time. Having the flexibility of having multiple grids that can be formed dynamically from a pool of processors is extremely beneficial. It allows us to have multiple parallel contractions. It can also be exploited to schedule larger and more expensive contractions on larger grids and smaller, less expensive ones on a smaller one [4]. For this to work, a more generic redistribution algorithm that can redistribute the data onto an entirely different grid needs to be developed. This type of redistribution also needs a more flexible cost model and a scheduler.

References

[1] R. Bartlett and M. Musia, Coupled-cluster theory in quantum chemistry Reviews of Modern Physics, 79(1):291{352, (2007)

[2] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, G. Agrawal, Optimizing tensor contraction expressions for hybrid CPU-GPU execution. Cluster Computing 16(1): 131-155 (2013)

[3] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, and P. Sadayappan, A communication-optimal framework for contracting distributed tensors, SC 2014

[4] P-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, P. Sadayappan, A framework for load balancing of tensor contraction expressions via dynamic task partitioning, SC 2013

[5] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions, IPDPS – 2013

Appendix A: List of contractions

```
   Contraction* C0 = new Contraction(vaa_vovv/*Input Tensor A*/, ta_vo/*Input
Tensor B*/, _a24849/*Output Tensor C*/);
   C0->contract( "p1a,h2a,p3a,p2a"/*Indices of Tensor A*/, "p3a,h2a"/*Indices of
Tensor B*/, "p1a,p2a"/*Indices of Tensor C*/);

   Contraction* C1 = new Contraction(vab_oovv, tb_vo, _a5246);
   C1->contract( "h2a,h1b,p2a,p1b", "p1b,h2b", "h2a,h1b,p2a,h2b");

   Contraction* C2 = new Contraction(vbb_oovv, tbb_vvoo, _a14811);
   C2->contract( "h1b,h3b,p1b,p3b", "p1b,p3b,h2b,h3b", "h1b,h2b");

   Contraction* C3 = new Contraction(vab_vovo, ta_vo, _a30368);
   C3->contract( "p1a,h1b,p2a,h2b", "p2a,h1a", "p1a,h1b,h2b,h1a");

   Contraction* C4 = new Contraction(vab_ovvv, tab_vvoo, _a34826);
   C4->contract( "h2a,p2b,p2a,p1b", "p2a,p1b,h1a,h2b", "h2a,p2b,h1a,h2b");

   Contraction* C5 = new Contraction(vab_vovv, tb_vo, _a9395);
   C5->contract( "p1a,h1b,p2a,p1b", "p1b,h1b", "p1a,p2a");

   Contraction* C6 = new Contraction(vaa_oovv, tab_vvoo, _a3071);
   C6->contract( "h2a,h3a,p2a,p3a", "p3a,p2b,h3a,h2b", "h2a,p2b,p2a,h2b");

   Contraction* C7 = new Contraction(vab_vovv, tb_vo, _a3715);
   C7->contract( "p1a,h1b,p2a,p1b", "p1b,h2b", "p1a,h1b,p2a,h2b");

   Contraction* C8 = new Contraction(vab_oovv, ta_vo, _a12895);
   C8->contract( "h2a,h1b,p2a,p1b", "p2a,h2a", "h1b,p1b");

   Contraction* C9 = new Contraction(tab_vvoo, _a5246, _a22682);
   C9->contract( "p2a,p2b,h1a,h1b", "h2a,h1b,p2a,h2b", "p2b,h2a,h1a,h2b");

   Contraction* C10 = new Contraction(vab_oovo, tab_vvoo, _a27108);
   C10->contract( "h2a,h1b,p2a,h2b", "p2a,p2b,h1a,h1b", "h2a,p2b,h2b,h1a");
```

```
Contraction* C11 = new Contraction(vbb_oovo, tab_vvoo, _a27232);
C11->contract( "h1b,h3b,p1b,h2b", "p1a,p1b,h1a,h3b", "h1b,p1a,h2b,h1a");

Contraction* C12 = new Contraction(vab_ooov, tab_vvoo, _a27231);
C12->contract( "h2a,h1b,h1a,p1b", "p1a,p1b,h2a,h2b", "h1b,p1a,h1a,h2b");

Contraction* C13 = new Contraction(vbb_oovv, tbb_vvoo, _a3023);
C13->contract( "h1b,h3b,p1b,p3b", "p2b,p3b,h2b,h3b", "h1b,p2b,p1b,h2b");

Contraction* C14 = new Contraction(vab_oovo, taa_vvoo, _a27296);
C14->contract( "h2a,h1b,p2a,h2b", "p1a,p2a,h1a,h2a", "h1b,p1a,h2b,h1a");

Contraction* C15 = new Contraction(vaa_oovv, ta_vo, _a18241);
C15->contract( "h2a,h3a,p2a,p3a", "p3a,h2a", "h3a,p2a");

Contraction* C16 = new Contraction(ta_vo, _a18241, _a18246);
C16->contract( "p2a,h1a", "h3a,p2a", "h3a,h1a");

Contraction* C17 = new Contraction(vbb_oovv, tb_vo, _a12781);
C17->contract( "h1b,h3b,p1b,p3b", "p1b,h3b", "h1b,p3b");

Contraction* C18 = new Contraction(vab_oovv, tab_vvoo, _a3029);
C18->contract( "h2a,h1b,p2a,p1b", "p2a,p2b,h2a,h2b", "h1b,p2b,p1b,h2b");

Contraction* C19 = new Contraction(vab_oovv, taa_vvoo, _a3040);
C19->contract( "h2a,h1b,p2a,p1b", "p1a,p2a,h1a,h2a", "h1b,p1a,p1b,h1a");

Contraction* C20 = new Contraction(vab_ovvv, ta_vo, _a34268);
C20->contract( "h2a,p2b,p2a,p1b", "p2a,h1a", "h2a,p2b,p1b,h1a");

Contraction* C21 = new Contraction(tab_vvoo, _a34269, _a34270);
C21->contract( "p1a,p1b,h2a,h2b", "h2a,p2b,p1b,h1a", "p1a,p2b,h2b,h1a");

Contraction* C22 = new Contraction(tb_vo, _a3040, _a23395);
C22->contract( "p1b,h2b", "h1b,p1a,p1b,h1a", "h1b,p1a,h2b,h1a");

Contraction* C23 = new Contraction(vaa_oovv, ta_vo, _a28768);
C23->contract( "h2a,h3a,p3a,p2a", "p3a,h3a", "h2a,p2a");

Contraction* C24 = new Contraction(vbb_oovv, tb_vo, _a18471);
C24->contract( "h1b,h3b,p1b,p3b", "p3b,h1b", "h3b,p1b");

Contraction* C25 = new Contraction(fa_ov, tab_vvoo, _a29926);
C25->contract( "h2a,p2a", "p2a,p2b,h1a,h2b", "h2a,p2b,h1a,h2b");
```

```
Contraction* C26 = new Contraction(tb_vo, _a18471, _a18476);
C26->contract( "p1b,h2b", "h3b,p1b", "h3b,h2b");

Contraction* C27 = new Contraction(vab_oovv, tab_vvoo, _a4801);
C27->contract( "h2a,h1b,p2a,p1b", "p2a,p2b,h2a,h1b", "p2b,p1b");

Contraction* C28 = new Contraction(vaa_vovv, ta_vo, _a3685);
C28->contract( "p1a,h2a,p2a,p3a", "p2a,h1a", "p1a,h2a,p3a,h1a");

Contraction* C29 = new Contraction(vab_ovvo, ta_vo, _a30499);
C29->contract( "h2a,p2b,p2a,h2b", "p2a,h1a", "h2a,p2b,h2b,h1a");

Contraction* C30 = new Contraction(vab_vovv, tab_vvoo, _a34818);
C30->contract( "p1a,h1b,p2a,p1b", "p2a,p1b,h1a,h2b", "p1a,h1b,h1a,h2b");

Contraction* C31 = new Contraction(vbb_oovv, tbb_vvoo, _a4981);
C31->contract( "h1b,h3b,p1b,p3b", "p2b,p3b,h1b,h3b", "p2b,p1b");

Contraction* C32 = new Contraction(vab_vvov, tb_vo, _a10340);
C32->contract( "p1a,p2b,h1a,p1b", "p1b,h2b", "p1a,p2b,h1a,h2b");

Contraction* C33 = new Contraction(vaa_oovo, ta_vo, _a19150);
C33->contract( "h2a,h3a,p2a,h1a", "p2a,h2a", "h3a,h1a");

Contraction* C34 = new Contraction(vbb_oovv, tab_vvoo, _a3197);
C34->contract( "h1b,h2b,p1b,p2b", "p2a,p2b,h2a,h2b", "h1b,p2a,p1b,h2a");

Contraction* C35 = new Contraction(vab_oovv, tab_vvoo, _a13050);
C35->contract( "h2a,h1b,p2a,p1b", "p2a,p1b,h1a,h1b", "h2a,h1a");

Contraction* C36 = new Contraction(vab_voov, tb_vo, _a30095);
C36->contract( "p1a,h1b,h1a,p1b", "p1b,h2b", "p1a,h1b,h1a,h2b");

Contraction* C37 = new Contraction(fb_ov, tab_vvoo, _a30094);
C37->contract( "h1b,p1b", "p1a,p1b,h1a,h2b", "h1b,p1a,h1a,h2b");

Contraction* C38 = new Contraction(tb_vo, _a12895, _a18706);
C38->contract( "p1b,h2b", "h1b,p1b", "h1b,h2b");

Contraction* C39 = new Contraction(fa_ov, ta_vo, _a19350);
C39->contract( "h2a,p2a", "p2a,h1a", "h2a,h1a");

Contraction* C40 = new Contraction(vab_vovv, ta_vo, _a3655);
C40->contract( "p1a,h1b,p2a,p1b", "p2a,h1a", "p1a,h1b,p1b,h1a");
```

```
Contraction* C41 = new Contraction(tb_vo, _a24960, _a26520);
C41->contract( "p1b,h2b", "h1b,p1a,p1b,h1a", "h1b,p1a,h2b,h1a");

Contraction* C42 = new Contraction(vab_oovv, ta_vo, _a5130);
C42->contract( "h2a,h1b,p2a,p1b", "p2a,h1a", "h2a,h1b,p1b,h1a");

Contraction* C43 = new Contraction(tbb_vvoo, _a5130, _a22677);
C43->contract( "p2b,p1b,h2b,h1b", "h2a,h1b,p1b,h1a", "p2b,h2a,h2b,h1a");

Contraction* C44 = new Contraction(vab_ovvv, ta_vo, _a9575);
C44->contract( "h2a,p2b,p2a,p1b", "p2a,h2a", "p2b,p1b");

Contraction* C45 = new Contraction(ta_vo, tb_vo, _a34095);
C45->contract( "p2a,h1a", "p1b,h2b", "p2a,p1b,h1a,h2b");

Contraction* C46 = new Contraction(vab_vvvv, _a34096, _a34097);
C46->contract( "p1a,p2b,p2a,p1b", "p2a,p1b,h1a,h2b", "p1a,p2b,h1a,h2b");

Contraction* C47 = new Contraction(vab_oovo, ta_vo, _a19210);
C47->contract( "h2a,h1b,p2a,h2b", "p2a,h2a", "h1b,h2b");

Contraction* C48 = new Contraction(vaa_oovo, tab_vvoo, _a27056);
C48->contract( "h2a,h3a,p2a,h1a", "p2a,p2b,h3a,h2b", "h2a,p2b,h1a,h2b");

Contraction* C49 = new Contraction(vab_ovov, tb_vo, _a29927);
C49->contract( "h2a,p2b,h1a,p1b", "p1b,h2b", "h2a,p2b,h1a,h2b");

Contraction* C50 = new Contraction(vbb_vovv, tb_vo, _a3775);
C50->contract( "p2b,h1b,p1b,p3b", "p1b,h2b", "p2b,h1b,p3b,h2b");

Contraction* C51 = new Contraction(tab_vvoo, _a34382, _a34383);
C51->contract( "p1a,p1b,h1a,h1b", "h1b,p2b,p1b,h2b", "p1a,p2b,h1a,h2b");

Contraction* C52 = new Contraction(vab_oovv, tb_vo, _a12541);
C52->contract( "h2a,h1b,p2a,p1b", "p1b,h1b", "h2a,p2a");

Contraction* C53 = new Contraction(ta_vo, _a12541, _a18015);
C53->contract( "p2a,h1a", "h2a,p2a", "h2a,h1a");

Contraction* C54 = new Contraction(ta_vo, _a28769, _a34717);
C54->contract( "p1a,h2a", "h2a,p2a", "p1a,p2a");

Contraction* C55 = new Contraction(vab_ooov, tbb_vvoo, _a27055);
C55->contract( "h2a,h1b,h1a,p1b", "p2b,p1b,h2b,h1b", "h2a,p2b,h1a,h2b");
```

```
Contraction* C56 = new Contraction(vab_oovv, tab_vvoo, _a13021);
C56->contract( "h2a,h1b,p2a,p1b", "p2a,p1b,h2a,h2b", "h1b,h2b");

Contraction* C57 = new Contraction(vaa_oovv, taa_vvoo, _a5011);
C57->contract( "h2a,h3a,p2a,p3a", "p1a,p3a,h2a,h3a", "p1a,p2a");

Contraction* C58 = new Contraction(vab_oovo, ta_vo, _a10295);
C58->contract( "h2a,h1b,p2a,h2b", "p2a,h1a", "h2a,h1b,h2b,h1a");

Contraction* C59 = new Contraction(fb_ov, tb_vo, _a19380);
C59->contract( "h1b,p1b", "p1b,h2b", "h1b,h2b");

Contraction* C60 = new Contraction(vab_ovvv, tb_vo, _a3805);
C60->contract( "h2a,p2b,p2a,p1b", "p1b,h2b", "h2a,p2b,p2a,h2b");

Contraction* C61 = new Contraction(vab_oooo, tb_vo, _a34780);
C61->contract( "h2a,h1b,h1a,h2b", "p2b,h1b", "h2a,p2b,h1a,h2b");

Contraction* C62 = new Contraction(vab_ooov, tb_vo, _a19120);
C62->contract( "h2a,h1b,h1a,p1b", "p1b,h1b", "h2a,h1a");

Contraction* C63 = new Contraction(tab_vvoo, _a34239, _a34240);
C63->contract( "p3a,p2b,h2a,h2b", "p1a,h2a,p3a,h1a", "p2b,p1a,h2b,h1a");

Contraction* C64 = new Contraction(vab_ooov, tb_vo, _a10265);
C64->contract( "h2a,h1b,h1a,p1b", "p1b,h2b", "h2a,h1b,h1a,h2b");

Contraction* C65 = new Contraction(ta_vo, _a27051, _a30369);
C65->contract( "p1a,h2a", "h2a,h1b,h1a,h2b", "p1a,h1b,h1a,h2b");

Contraction* C66 = new Contraction(vbb_vovv, tb_vo, _a9515);
C66->contract( "p2b,h1b,p1b,p3b", "p1b,h1b", "p2b,p3b");

Contraction* C67 = new Contraction(tab_vvoo, _a34888, _a34889);
C67->contract( "p1a,p1b,h1a,h2b", "p2b,p1b", "p1a,p2b,h1a,h2b");

Contraction* C68 = new Contraction(taa_vvoo, _a34809, _a34810);
C68->contract( "p1a,p2a,h1a,h2a", "h2a,p2b,p2a,h2b", "p1a,p2b,h1a,h2b");

Contraction* C69 = new Contraction(vab_vvvo, ta_vo, _a10345);
C69->contract( "p1a,p2b,p2a,h2b", "p2a,h1a", "p1a,p2b,h2b,h1a");

Contraction* C70 = new Contraction(vab_oovv, tab_vvoo, _a3125);
C70->contract( "h2a,h1b,p2a,p1b", "p1a,p1b,h2a,h2b", "h1b,p1a,p2a,h2b");
```

```
Contraction* C71 = new Contraction(tab_vvoo, _a34520, _a34521);
C71->contract( "p2a,p2b,h1a,h1b", "h1b,p1a,p2a,h2b", "p2b,p1a,h1a,h2b");

Contraction* C72 = new Contraction(ta_vo, _a3125, _a23391);
C72->contract( "p2a,h1a", "h1b,p1a,p2a,h2b", "h1b,p1a,h1a,h2b");

Contraction* C73 = new Contraction(vab_oovv, tab_vvoo, _a4861);
C73->contract( "h2a,h1b,p2a,p1b", "p1a,p1b,h2a,h1b", "p1a,p2a");

Contraction* C74 = new Contraction(tab_vvoo, _a34896, _a34897);
C74->contract( "p2a,p2b,h1a,h2b", "p1a,p2a", "p2b,p1a,h1a,h2b");

Contraction* C75 = new Contraction(vab_oovv, tab_vvoo, _a4920);
C75->contract( "h2a,h1b,p2a,p1b", "p2a,p1b,h1a,h2b", "h2a,h1b,h1a,h2b");

Contraction* C76 = new Contraction(ta_vo, _a4920, _a26518);
C76->contract( "p1a,h2a", "h2a,h1b,h1a,h2b", "p1a,h1b,h1a,h2b");

Contraction* C77 = new Contraction(tab_vvoo, _a28178, _a29111);
C77->contract( "p1a,p3b,h1a,h2b", "h1b,p3b", "p1a,h1b,h1a,h2b");

Contraction* C78 = new Contraction(vaa_oovv, taa_vvoo, _a14870);
C78->contract( "h2a,h3a,p2a,p3a", "p2a,p3a,h1a,h2a", "h3a,h1a");

Contraction* C79 = new Contraction(tab_vvoo, _a34932, _a34933);
C79->contract( "p1a,p2b,h2a,h2b", "h2a,h1a", "p1a,p2b,h2b,h1a");

Contraction* C80 = new Contraction(vbb_oovo, tb_vo, _a19180);
C80->contract( "h1b,h3b,p1b,h2b", "p1b,h1b", "h3b,h2b");

Contraction* C81 = new Contraction(tab_vvoo, _a34924, _a34925);
C81->contract( "p1a,p2b,h1a,h1b", "h1b,h2b", "p1a,p2b,h1a,h2b");

Contraction* C82 = new Contraction(ta_vo, _a25416, _a34676);
C82->contract( "p2a,h1a", "h2a,p2b,p2a,h2b", "h2a,p2b,h1a,h2b");

Contraction* C83 = new Contraction(ta_vo, _a34952, _a34953);
C83->contract( "p1a,h2a", "p2b,h2a,h2b,h1a", "p1a,p2b,h2b,h1a");

Contraction* C84 = new Contraction(tbb_vvoo, _a34463, _a34464);
C84->contract( "p2b,p1b,h2b,h1b", "h1b,p1a,p1b,h1a", "p2b,p1a,h2b,h1a");

Contraction* C85 = new Contraction(ta_vo, _a5246, _a9160);
C85->contract( "p2a,h1a", "h2a,h1b,p2a,h2b", "h2a,h1b,h1a,h2b");
```

```
Contraction* C86 = new Contraction(tab_vvoo, _a34878, _a34879);
C86->contract( "p1a,p2b,h2a,h1b", "h2a,h1b,h1a,h2b", "p1a,p2b,h1a,h2b");

Contraction* C87 = new Contraction(ta_vo, _a9160, _a29106);
C87->contract( "p1a,h2a", "h2a,h1b,h1a,h2b", "p1a,h1b,h1a,h2b");

Contraction* C88 = new Contraction(tb_vo, _a34944, _a34945);
C88->contract( "p2b,h1b", "h1b,p1a,h1a,h2b", "p2b,p1a,h1a,h2b");

Contraction* C89 = new Contraction(vaa_vovv, taa_vvoo, _a4601);
C89->contract( "p1a,h3a,p3a,p4a", "p3a,p4a,h1a,h2a", "p1a,h3a,h1a,h2a");

Contraction* C90 = new Contraction(vaa_oovv, taa_vvoo, _a10790);
C90->contract( "h3a,h4a,p3a,p4a", "p3a,p4a,h1a,h2a", "h3a,h4a,h1a,h2a");

Contraction* C91 = new Contraction(vaa_vovv, ta_vo, _a9455);
C91->contract( "p1a,h2a,p2a,p3a", "p2a,h2a", "p1a,p3a");

Contraction* C92 = new Contraction(vaa_oovo, taa_vvoo, _a9891);
C92->contract( "h3a,h4a,p3a,h1a", "p2a,p3a,h2a,h4a", "h3a,p2a,h1a,h2a");

Contraction* C93 = new Contraction(ta_vo, _a9891, _a9894);
C93->contract( "p1a,h3a", "h3a,p2a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C94 = new Contraction(vaa_oovv, taa_vvoo, _a46510);
C94->contract( "h3a,h4a,p3a,p4a", "p3a,p4a,h2a,h4a", "h3a,h2a");

Contraction* C95 = new Contraction(taa_vvoo, _a9395, _a10354);
C95->contract( "p2a,p3a,h1a,h2a", "p1a,p3a", "p2a,p1a,h1a,h2a");

Contraction* C96 = new Contraction(fa_ov, taa_vvoo, _a18991);
C96->contract( "h3a,p3a", "p2a,p3a,h1a,h2a", "h3a,p2a,h1a,h2a");

Contraction* C97 = new Contraction(vaa_vvvo, ta_vo, _a10540);
C97->contract( "p1a,p2a,p3a,h1a", "p3a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C98 = new Contraction(vab_ooov, tab_vvoo, _a9861);
C98->contract( "h3a,h1b,h1a,p1b", "p2a,p1b,h2a,h1b", "h3a,p2a,h1a,h2a");

Contraction* C99 = new Contraction(ta_vo, _a9861, _a9864);
C99->contract( "p1a,h3a", "h3a,p2a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C100 = new Contraction(vaa_oovo, ta_vo, _a10910);
C100->contract( "h3a,h4a,p3a,h1a", "p3a,h2a", "h3a,h4a,h1a,h2a");
```

```
Contraction* C101 = new Contraction(vaa_oovo, ta_vo, _a42744);
C101->contract( "h4a,h3a,p3a,h2a", "p3a,h4a", "h3a,h2a");

Contraction* C102 = new Contraction(tab_vvoo, _a3655, _a3869);
C102->contract( "p2a,p1b,h2a,h1b", "p1a,h1b,p1b,h1a", "p2a,p1a,h2a,h1a");

Contraction* C103 = new Contraction(vaa_oovv, ta_vo, _a10551);
C103->contract( "h3a,h4a,p3a,p4a", "p4a,h2a", "h3a,h4a,p3a,h2a");

Contraction* C104 = new Contraction(ta_vo, _a10551, _a10556);
C104->contract( "p3a,h1a", "h3a,h4a,p3a,h2a", "h3a,h4a,h1a,h2a");

Contraction* C105 = new Contraction(taa_vvoo, _a5011, _a9818);
C105->contract( "p1a,p3a,h1a,h2a", "p2a,p3a", "p1a,p2a,h1a,h2a");

Contraction* C106 = new Contraction(ta_vo, _a28769, _a52784);
C106->contract( "p3a,h2a", "h3a,p3a", "h3a,h2a");

Contraction* C107 = new Contraction(tab_vvoo, _a3040, _a3283);
C107->contract( "p1a,p1b,h1a,h1b", "h1b,p2a,p1b,h2a", "p1a,p2a,h1a,h2a");

Contraction* C108 = new Contraction(tab_vvoo, _a5130, _a5959);
C108->contract( "p2a,p1b,h2a,h1b", "h3a,h1b,p1b,h1a", "p2a,h3a,h2a,h1a");

Contraction* C109 = new Contraction(ta_vo, _a5959, _a5967);
C109->contract( "p1a,h3a", "p2a,h3a,h2a,h1a", "p1a,p2a,h2a,h1a");

Contraction* C110 = new Contraction(vab_voov, tab_vvoo, _a4490);
C110->contract( "p1a,h1b,h1a,p1b", "p2a,p1b,h2a,h1b", "p1a,p2a,h1a,h2a");

Contraction* C111 = new Contraction(vaa_vovo, ta_vo, _a13260);
C111->contract( "p1a,h3a,p3a,h1a", "p3a,h2a", "p1a,h3a,h1a,h2a");

Contraction* C112 = new Contraction(vaa_vovo, taa_vvoo, _a4495);
C112->contract( "p1a,h3a,p3a,h1a", "p2a,p3a,h2a,h3a", "p1a,p2a,h1a,h2a");

Contraction* C113 = new Contraction(ta_vo, _a13260, _a13265);
C113->contract( "p2a,h3a", "p1a,h3a,h1a,h2a", "p2a,p1a,h1a,h2a");

Contraction* C114 = new Contraction(ta_vo, _a3685, _a6775);
C114->contract( "p4a,h2a", "p1a,h3a,p4a,h1a", "p1a,h3a,h2a,h1a");

Contraction* C115 = new Contraction(taa_vvoo, _a3685, _a3919);
C115->contract( "p2a,p4a,h2a,h3a", "p1a,h3a,p4a,h1a", "p2a,p1a,h2a,h1a");
```

```
Contraction* C116 = new Contraction(taa_vvoo, _a9455, _a10379);
C116->contract( "p2a,p4a,h1a,h2a", "p1a,p4a", "p2a,p1a,h1a,h2a");

Contraction* C117 = new Contraction(taa_vvoo, _a4861, _a5043);
C117->contract( "p1a,p3a,h1a,h2a", "p2a,p3a", "p1a,p2a,h1a,h2a");

Contraction* C118 = new Contraction(vaa_oovv, ta_vo, _a12666);
C118->contract( "h2a,h3a,p2a,p3a", "p2a,h3a", "h2a,p3a");

Contraction* C119 = new Contraction(taa_vvoo, _a12666, _a15737);
C119->contract( "p2a,p4a,h1a,h2a", "h3a,p4a", "p2a,h3a,h1a,h2a");

Contraction* C120 = new Contraction(tab_vvoo, _a3197, _a3200);
C120->contract( "p1a,p1b,h1a,h1b", "h1b,p2a,p1b,h2a", "p1a,p2a,h1a,h2a");

Contraction* C121 = new Contraction(ta_vo, _a6775, _a6788);
C121->contract( "p2a,h3a", "p1a,h3a,h2a,h1a", "p2a,p1a,h2a,h1a");

Contraction* C122 = new Contraction(vaa_oooo, ta_vo, _a42703);
C122->contract( "h3a,h4a,h1a,h2a", "p1a,h3a", "h4a,p1a,h1a,h2a");

Contraction* C123 = new Contraction(taa_vvoo, _a12541, _a15498);
C123->contract( "p2a,p3a,h1a,h2a", "h3a,p3a", "p2a,h3a,h1a,h2a");

Contraction* C124 = new Contraction(taa_vvoo, _a121238, _a121239);
C124->contract( "p1a,p2a,h1a,h3a", "h3a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C125 = new Contraction(vaa_vvvv, taa_vvoo, _a3645);
C125->contract( "p1a,p2a,p3a,p4a", "p3a,p4a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C126 = new Contraction(taa_vvoo, _a107369, _a107370);
C126->contract( "p1a,p2a,h2a,h3a", "h3a,h1a", "p1a,p2a,h2a,h1a");

Contraction* C127 = new Contraction(ta_vo, _a45798, _a48054);
C127->contract( "p2a,h4a", "h3a,h4a,h1a,h2a", "p2a,h3a,h1a,h2a");

Contraction* C128 = new Contraction(ta_vo, _a86275, _a86276);
C128->contract( "p1a,h3a", "p2a,h3a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C129 = new Contraction(taa_vvoo, _a57517, _a57518);
C129->contract( "p1a,p2a,h3a,h4a", "h3a,h4a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C130 = new Contraction(vaa_oovv, taa_vvoo, _a3331);
C130->contract( "h3a,h4a,p3a,p4a", "p2a,p4a,h2a,h4a", "h3a,p2a,p3a,h2a");
```

```
Contraction* C131 = new Contraction(ta_vo, _a3331, _a8218);
C131->contract( "p3a,h1a", "h3a,p2a,p3a,h2a", "h3a,p2a,h1a,h2a");

Contraction* C132 = new Contraction(ta_vo, _a8218, _a8227);
C132->contract( "p1a,h3a", "h3a,p2a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C133 = new Contraction(taa_vvoo, _a3331, _a3334);
C133->contract( "p1a,p3a,h1a,h3a", "h3a,p2a,p3a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C134 = new Contraction(fa_vv, taa_vvoo, _a10530);
C134->contract( "p1a,p3a", "p2a,p3a,h1a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C135 = new Contraction(ta_vo, _a93335, _a93336);
C135->contract( "p2a,h3a", "h3a,p1a,h1a,h2a", "p2a,p1a,h1a,h2a");

Contraction* C136 = new Contraction(vaa_vvvv, ta_vo, _a4511);
C136->contract( "p1a,p2a,p3a,p4a", "p4a,h2a", "p1a,p2a,p3a,h2a");

Contraction* C137 = new Contraction(ta_vo, _a4511, _a4514);
C137->contract( "p3a,h1a", "p1a,p2a,p3a,h2a", "p1a,p2a,h1a,h2a");

Contraction* C138 = new Contraction(tb_vo, _a3029, _a7197);
C138->contract( "p3b,h1b", "h3b,p2b,p3b,h2b", "h3b,p2b,h1b,h2b");

Contraction* C139 = new Contraction(tab_vvoo, _a3071, _a3543);
C139->contract( "p1a,p1b,h1a,h1b", "h1a,p2b,p1a,h2b", "p1b,p2b,h1b,h2b");

Contraction* C140 = new Contraction(vbb_vovo, tb_vo, _a13350);
C140->contract( "p1b,h3b,p3b,h1b", "p3b,h2b", "p1b,h3b,h1b,h2b");

Contraction* C141 = new Contraction(tbb_vvoo, _a4981, _a10043);
C141->contract( "p1b,p3b,h1b,h2b", "p2b,p3b", "p1b,p2b,h1b,h2b");

Contraction* C142 = new Contraction(tbb_vvoo, _a4801, _a5089);
C142->contract( "p1b,p3b,h1b,h2b", "p2b,p3b", "p1b,p2b,h1b,h2b");

Contraction* C143 = new Contraction(tb_vo, _a3023, _a8628);
C143->contract( "p3b,h1b", "h3b,p2b,p3b,h2b", "h3b,p2b,h1b,h2b");

Contraction* C144 = new Contraction(fb_ov, tbb_vvoo, _a19071);
C144->contract( "h3b,p3b", "p2b,p3b,h1b,h2b", "h3b,p2b,h1b,h2b");

Contraction* C145 = new Contraction(vbb_vvvo, tb_vo, _a10545);
C145->contract( "p1b,p2b,p3b,h1b", "p3b,h2b", "p1b,p2b,h1b,h2b");
```

```
Contraction* C146 = new Contraction(vbb_vovo, tbb_vvoo, _a4500);
C146->contract( "p1b,h3b,p3b,h1b", "p2b,p3b,h2b,h3b", "p1b,p2b,h1b,h2b");

Contraction* C147 = new Contraction(vab_ovvo, tab_vvoo, _a4505);
C147->contract( "h1a,p1b,p1a,h1b", "p1a,p2b,h1a,h2b", "p1b,p2b,h1b,h2b");

Contraction* C148 = new Contraction(vbb_vvvv, tb_vo, _a4541);
C148->contract( "p1b,p2b,p3b,p4b", "p4b,h2b", "p1b,p2b,p3b,h2b");

Contraction* C149 = new Contraction(tb_vo, _a4541, _a4544);
C149->contract( "p3b,h1b", "p1b,p2b,p3b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C150 = new Contraction(tbb_vvoo, _a3775, _a4209);
C150->contract( "p2b,p4b,h2b,h3b", "p1b,h3b,p4b,h1b", "p2b,p1b,h2b,h1b");

Contraction* C151 = new Contraction(tbb_vvoo, _a12781, _a16377);
C151->contract( "p2b,p4b,h1b,h2b", "h3b,p4b", "p2b,h3b,h1b,h2b");

Contraction* C152 = new Contraction(tb_vo, _a13350, _a13355);
C152->contract( "p2b,h3b", "p1b,h3b,h1b,h2b", "p2b,p1b,h1b,h2b");

Contraction* C153 = new Contraction(tbb_vvoo, _a9575, _a10469);
C153->contract( "p2b,p3b,h1b,h2b", "p1b,p3b", "p2b,p1b,h1b,h2b");

Contraction* C154 = new Contraction(vbb_oovv, tbb_vvoo, _a10850);
C154->contract( "h3b,h4b,p3b,p4b", "p3b,p4b,h1b,h2b", "h3b,h4b,h1b,h2b");

Contraction* C155 = new Contraction(tbb_vvoo, _a12895, _a16595);
C155->contract( "p2b,p3b,h1b,h2b", "h3b,p3b", "p2b,h3b,h1b,h2b");

Contraction* C156 = new Contraction(vbb_oovo, tb_vo, _a11010);
C156->contract( "h3b,h4b,p3b,h1b", "p3b,h2b", "h3b,h4b,h1b,h2b");

Contraction* C157 = new Contraction(vbb_oovo, tbb_vvoo, _a10086);
C157->contract( "h3b,h4b,p3b,h1b", "p2b,p3b,h2b,h4b", "h3b,p2b,h1b,h2b");

Contraction* C158 = new Contraction(tb_vo, _a10086, _a10089);
C158->contract( "p1b,h3b", "h3b,p2b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C159 = new Contraction(tb_vo, _a7197, _a7206);
C159->contract( "p1b,h3b", "h3b,p2b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C160 = new Contraction(tb_vo, _a8628, _a8637);
C160->contract( "p1b,h3b", "h3b,p2b,h1b,h2b", "p1b,p2b,h1b,h2b");
```

```
Contraction* C161 = new Contraction(vbb_vovv, tbb_vvoo, _a4701);
C161->contract( "p1b,h3b,p3b,p4b", "p3b,p4b,h1b,h2b", "p1b,h3b,h1b,h2b");

Contraction* C162 = new Contraction(tab_vvoo, _a3805, _a4259);
C162->contract( "p1a,p2b,h1a,h2b", "h1a,p1b,p1a,h1b", "p2b,p1b,h2b,h1b");

Contraction* C163 = new Contraction(vbb_vvvv, tbb_vvoo, _a3650);
C163->contract( "p1b,p2b,p3b,p4b", "p3b,p4b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C164 = new Contraction(vbb_oovv, tb_vo, _a10671);
C164->contract( "h3b,h4b,p3b,p4b", "p4b,h2b", "h3b,h4b,p3b,h2b");

Contraction* C165 = new Contraction(tbb_vvoo, _a3029, _a3499);
C165->contract( "p1b,p3b,h1b,h3b", "h3b,p2b,p3b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C166 = new Contraction(tb_vo, _a10671, _a10676);
C166->contract( "p3b,h1b", "h3b,h4b,p3b,h2b", "h3b,h4b,h1b,h2b");

Contraction* C167 = new Contraction(tbb_vvoo, _a9515, _a10444);
C167->contract( "p2b,p4b,h1b,h2b", "p1b,p4b", "p2b,p1b,h1b,h2b");

Contraction* C168 = new Contraction(tbb_vvoo, _a3023, _a3473);
C168->contract( "p1b,p3b,h1b,h3b", "h3b,p2b,p3b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C169 = new Contraction(vab_oovo, tab_vvoo, _a10116);
C169->contract( "h1a,h3b,p1a,h1b", "p1a,p2b,h1a,h2b", "h3b,p2b,h1b,h2b");

Contraction* C170 = new Contraction(tb_vo, _a10116, _a10119);
C170->contract( "p1b,h3b", "h3b,p2b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C171 = new Contraction(vbb_oovv, tbb_vvoo, _a174692);
C171->contract( "h3b,h4b,p3b,p4b", "p3b,p4b,h1b,h3b", "h4b,h1b");

Contraction* C172 = new Contraction(tbb_vvoo, _a241201, _a241202);
C172->contract( "p1b,p2b,h2b,h3b", "h3b,h1b", "p1b,p2b,h2b,h1b");

Contraction* C173 = new Contraction(tbb_vvoo, _a254814, _a254815);
C173->contract( "p1b,p2b,h1b,h3b", "h3b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C174 = new Contraction(vbb_oooo, tb_vo, _a177605);
C174->contract( "h3b,h4b,h1b,h2b", "p1b,h3b", "h4b,p1b,h1b,h2b");

Contraction* C175 = new Contraction(tb_vo, _a227425, _a227426);
C176->contract( "p2b,h3b", "h3b,p1b,h1b,h2b", "p2b,p1b,h1b,h2b");
```

```
Contraction* C176 = new Contraction(tb_vo, _a180621, _a182867);
C177->contract( "p2b,h4b", "h3b,h4b,h1b,h2b", "p2b,h3b,h1b,h2b");

Contraction* C177 = new Contraction(tbb_vvoo, _a192257, _a192258);
C178->contract( "p1b,p2b,h3b,h4b", "h3b,h4b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C178 = new Contraction(tb_vo, _a220495, _a220496);
C179->contract( "p1b,h3b", "p2b,h3b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C179 = new Contraction(fb_vv, tbb_vvoo, _a10535);
C180->contract( "p1b,p3b", "p2b,p3b,h1b,h2b", "p1b,p2b,h1b,h2b");

Contraction* C180 = new Contraction(tb_vo, _a3775, _a7785);
C181->contract( "p4b,h2b", "p1b,h3b,p4b,h1b", "p1b,h3b,h2b,h1b");

Contraction* C181 = new Contraction(tb_vo, _a7785, _a7798);
C182->contract( "p2b,h3b", "p1b,h3b,h2b,h1b", "p2b,p1b,h2b,h1b");
```