SANDIA REPORT

SAND2010-3388
Unlimited Release
Printed May 2010

Steps toward Fault-Tolerant Quantum Chemistry

Andrew G. Taube

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

Steps toward Fault-Tolerant Quantum Chemistry

Andrew G. Taube
Multiscale Dynamics Materials Modeling Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185
agtaube@sandia.gov

Abstract

Developing quantum chemistry programs on the coming generation of exascale computers will be a difficult task. The programs will need to be fault-tolerant and minimize the use of global operations. This work explores the use a task-based model that uses a data-centric approach to allocate work to different processes as it applies to quantum chemistry. After introducing the key problems that appear when trying to parallelize a complicated quantum chemistry method such as coupled-cluster theory, we discuss the implications of that model as it pertains to the computational kernel of a coupled-cluster program – matrix multiplication. Also, we discuss the extensions that would required to build a full coupled-cluster program using the task-based model.

Contents

1	Introduction	7
2	Quantum Chemistry Models	8
2.1	Coupled-cluster	8
2.2	Convergence Algorithms	9
3	Task-based computation	13
3.1	Matrix multiply	13
3.2	General model for CC iteration	17
4	Conclusion and Further Work	19
	References	20

Appendix

A	Class Definitions	21
---	-------------------------	----

Figures

1	Algorithm for the preconditioned conjugate gradient for LinCC	11
2	Algorithm for a simple matrix multiply	14
3	Algorithm for recursive task-based block subdivision for matrix multiplication ...	15
4	Algorithm for scheduler to run different tasks and access the memory controlled by the storer.	16
5	Workflow for single operation in task-based operation. The order of operations corresponds to the numbers. Single arrows are non-blocking read operations while the lone double arrow [(7)] is a blocking write. Dashed lines indicate that it is one of two possible operations. Bold letters are matrices, a small letter 'r' indicates a reference to actual data, while the letter without the 'r' is the data itself.	17

1 Introduction

Current programming models for high-performance computing are fault-intolerant and use global operations. Those properties are unsustainable as computers scale to millions of CPUs; instead one must recognize that these systems will be hierarchical in structure, prone to constant faults, and global operations will be infeasible.

The FAST-OS HARE project is introducing a scale-free computing model to address these issues. This model is hierarchical and fault-tolerant by design, allows for the clean overlap of computation and communication, reducing the network load, does not require checkpointing, and avoids the complexity of many HPC runtimes. Development of an algorithm within this model requires a change in focus from imperative programming to a data-centric approach.

Quantum chemistry (QC) algorithms, in particular electronic structure methods, are an ideal test bed for this computing model. These methods describe the distribution of electrons in a molecule, which determine the properties of the molecule. The computational cost of these methods is high, scaling quartically or higher in the size of the molecule, which is why QC applications are major users of HPC resources. The complexity of these algorithms means that MPI alone is insufficient to achieve parallel scaling; QC developers have been forced to use alternative approaches to achieve scalability and would be receptive to radical shifts in the programming paradigm.

Initial work in adapting the simplest QC method, Hartree-Fock, to this the new programming model indicates that the approach is beneficial for QC applications. However, the advantages to being able to scale to exascale computers are greatest for the computationally most expensive algorithms; within QC these are the high-accuracy coupled-cluster (CC) methods. Parallel coupled-cluster programs are available, however they are based on the conventional MPI paradigm. Much of the effort is spent handling the complicated data dependencies between the various processors, especially as the size of the problem becomes large. The current paradigm will not survive the move to exascale computers.

Here we discuss the initial steps toward designing and implementing a CC method within this model. First, we introduce the general concepts behind a CC method, focusing on the aspects that make these methods difficult to parallelize with conventional techniques. Then we outline what is the computational core of the CC method – a matrix multiply – within the task-based approach that the FAST-OS project is designed to take advantage of. Finally we outline the general setup to implement the simplest CC method in this model, linearized CC doubles (LinCC).

2 Quantum Chemistry Models

Chemistry is fundamentally about electrons. Bonding, reactions, excitations, and properties such as dipole moments are determined by the distribution of electrons in a molecule and how that distribution changes in time or due to an external perturbation. Knowledge of the distribution of electrons and its response properties allows for the prediction of a wide variety of experimental results. One of the goals of computational chemistry is to calculate these quantities. These results can be used to interpret experimental results, to make predictions about molecules that are yet to be synthesized, or to substitute for experiments that are difficult to perform. For example, the nature of the reactions of energetic materials, such as jet fuels and explosives, makes performing tabletop experiments virtually impossible.

Electrons are truly quantum mechanical particles: non-quantum mechanical (classical) electrons would not bind molecules. One must deal with quantum mechanics directly when attempting to calculate electronic properties. The equations that determine the properties of a molecule are known; unfortunately, they are not analytically solvable for more than one electron. Instead, one must construct computationally feasible approximations to the exact equations. The search for better approximations and their application to problems of chemical interest defines the field of quantum chemistry. Often, one would like to perform calculations without any input from experiment, using only the fundamental equations of quantum mechanics; this is known as *ab initio* (“from the beginning”) quantum chemistry.

2.1 Coupled-cluster

The most powerful current group of methods for general quantum-chemical simulations are the coupled-cluster (CC) methods [1]. CC methods form a hierarchy of increasing accuracy – at the cost of increasing computer time. Even the simplest useful coupled-cluster method, CC doubles (CCD) or its linearized counterpart LinCCD, scales computationally with the sixth power of the system size. At this time, that means that the largest systems that are accessible to accurate calculation are on the order of tens of atoms, even using massively parallel computers.

Because the coupled-cluster equations are dense and there is substantial data dependency between the equations, efficiently parallelization is quite difficult. Recent work has led to domain-specific languages, such as the “super instruction assembly language” used in the ACES III code [2, 3, 4], that leverage MPI to try to get computational scaling to thousands or tens of thousands of processors. However, these approaches will not be able to scale to the millions of processors that will be necessary to achieve exascale computers. In particular, the tight data dependencies make the CC methods very sensitive to faults that could lead to either the loss of data or to a global stall as the entire code waits for a single piece of data to be provided, killing scalability.

To understand the difficulty involved in parallelizing these methods, we show the equations for the LinCCD method in its most recent and effective form [5]. This method has been implemented within ACES III and is known to scale to the tens of thousands of processors using present-day

technology. Unlike many of the coupled-cluster methods, LinCCD does not require the solution of a nonlinear set of equations, rather it depends on a simpler set of linear equations. LinCCD has less data dependence between the equations than do the nonlinear CC methods. Despite this simplifications, scaling LinCCD to the millions of processors efficiently is an extraordinarily challenging task.

LinCCD involves the iterative solution of a large, dense linear system of equations, whose dimensionality scales quartically in the size of the system. The size of the solution vector makes replicated storage infeasible for all but the smallest systems. Each element of the solution depends on a quadratic (in system size) number of other elements, as well as quadratic number of matrix elements. The linear equations for LinCCD are (within a Hartree-Fock basis),

$$\epsilon_{ij}^{ab} t_{ij}^{ab} = \langle ab || ij \rangle + \frac{1}{2} \sum_{cd} t_{ij}^{cd} \langle ab || cd \rangle + \frac{1}{2} \sum_{kl} t_{kl}^{ab} \langle kl || ij \rangle + P(ij|ab) \sum_{kc} t_{ik}^{ac} \langle kb || cj \rangle. \quad (1)$$

where the unknowns are the four-dimensional “amplitudes,” t_{ij}^{ab} and P is an index permutation operator. The bracketed objects are two-electron integrals in Dirac bra-ket notation. The letters i, j correspond to electron indices and a, b correspond to unoccupied basis functions. In practice the number of unoccupied indices is at least an order of magnitude higher than the number of electron indices. Note that this equations depends on all t amplitudes that share at least two indices with t_{ij}^{ab} , leading to overall dependence on a quadratic number of other equations. For a system of 100 electrons and 1000 unoccupied basis functions (feasible using current massively parallel algorithms), the number of independent variables, once symmetries have been taken into account, is on the order of *2.5 billion*. Given that each of these variables is represented by a double precision number, 20 gigabytes are necessary just to store the amplitudes on each iteration, with no consideration of other quantities in the equations or intermediates. Moving to a system an order of magnitude larger (beyond current abilities, but of great interest scientifically), would require 200 terabytes for storage, implying that replicated storage is infeasible.

2.2 Convergence Algorithms

In practice, one cannot expect to solve the linear equations above directly, or even using a simple fixed-point Jacobi iteration scheme. Instead, one must use a convergence acceleration method to make the iterations smoothly approach the desired solution. There are two main approaches to accelerating the convergence of the coupled-cluster amplitude equations: the direct inversion of the iterative subspace (DIIS) method of Pulay [6], as applied to the t amplitudes, or the reduced linear equation (RLE) method of Purvis and Bartlett [7], which developed from the reduced partitioning procedure [8, 9]. In both cases, one solves a linear equation within the subspace spanned by the prior iterates to determine the next step – they are Krylov subspace methods [10]. The iterates they generate span the space

$$\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) = \{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \mathbf{A}^3\mathbf{r}_0, \dots, \mathbf{A}^n\mathbf{r}_0\}. \quad (2)$$

Another Krylov subspace method applicable to linear, hermitian problems, such as the LinCC amplitude equations, is conjugate gradient (CG). Unlike the DIIS and RLE methods, CG does not

require the choice of a number of prior iterates to keep, because it satisfies an exact two-term recurrence. This simplicity is helpful for two reasons: first, because it removes one more choice that needs to be made in the course of a calculation and second, because it bounds the memory requirements of the algorithm. For a massively parallel algorithm on a distributed memory computer, memory is often the bottleneck and any approach that reduces the memory requirements is advantageous. Because all three algorithms are Krylov methods, for an n -dimensional problem, they all will converge in at most n iterations. The number of iterations actually required for convergence to some threshold is not immediately obvious and will depend on the problem being considered.

The amplitude equations for linearized coupled-cluster theory can always be written in the form

$$\mathbf{A}\mathbf{t} = -\mathbf{g} \quad (3)$$

As an example, for LinCCD the $\mathbf{A}$ matrix is given by

$$A_{(ijab,klcd)} = \frac{1}{2} \langle ab || cd \rangle \delta_{ik} \delta_{kl} + \frac{1}{2} \langle kl || ij \rangle \delta_{ac} \delta_{bd} + P(ij|ab) \delta_{ik} \delta_{ac} \langle lb || dj \rangle \\ - P(ij) \delta_{ik} \delta_{ac} \delta_{bd} f_{lj} + P(ab) \delta_{ik} \delta_{jl} \delta_{ac} f_{db} \quad (4)$$

The vector $\mathbf{t}$ contains the unknown amplitudes, and the vector $\mathbf{g}$ are the two-electron integrals

$$g_{ijab} = \langle ab || ij \rangle. \quad (5)$$

While for sufficiently large systems this matrix is sparse, for many chemically interesting examples, dense matrix operations are necessary.

Without any form of convergence acceleration, the standard method of solving the LinCC equations is a fixed point Jacobi-like approach

$$\mathbf{A}_{\text{MP}} \mathbf{t}^{[n+1]} = -\mathbf{g} - \mathbf{A}_{\text{r}} \mathbf{t}^{[n]} \quad (6)$$

where $\mathbf{A}_{\text{MP}}$ is the Møller-Plesset part of the matrix $\mathbf{A}$,

$$(A_{\text{MP}})_{(ijab,klcd)} = -P(ij) \delta_{ik} \delta_{ac} \delta_{bd} f_{lj} + P(ab) \delta_{ik} \delta_{jl} \delta_{ac} f_{db} \quad (7)$$

and $\mathbf{A}_{\text{r}}$ is the remainder,

$$\mathbf{A}_{\text{r}} = \mathbf{A} - \mathbf{A}_{\text{MP}}. \quad (8)$$

A true Jacobi iteration scheme would use the full diagonal in place of $\mathbf{A}_{\text{MP}}$; this would correspond to using Epstein-Nesbet (EN) denominators instead of MP denominators [7]. EN denominators are typically smaller than MP denominators, leading to more rapid divergence of the perturbation theory or, in this case, slower convergence of the iterative equations, so the MP choice is preferred. Unfortunately, Jacobi iterations do not converge for many cases of interest, and more powerful convergence approaches must be used.

Conjugate gradient algorithms, in their two-term recurrence form, are simpler than the standard algorithms. One constructs the residual, and then one takes a properly scaled step in the direction of the residual and orthogonal to all of the prior directions. To enhance convergence, one should

input : A diagonal matrix $\mathbf{A}_{\text{MP}}$, an off-diagonal matrix $\mathbf{A}_r$, a vector of integrals $\mathbf{g}$, and a regularization parameter ω^2 .

output: A solution of LinCC equations $\mathbf{t}$ such that the norm of the residual $\|\mathbf{r}\|_2 \leq \epsilon_{\text{tol}}$

begin

Regularized Denominator Inverse: $\Delta^{-1} = \mathbf{A}_{\text{MP}} [\mathbf{A}_{\text{MP}}^T \mathbf{A}_{\text{MP}} + \omega^2 \mathbf{I}]^{-1}$

Initial Guess: $\mathbf{t}_0 = \mathbf{g}$

$\mathbf{z}_0 = [\mathbf{g} + \mathbf{A}\mathbf{t}_0]$

Residual: $\mathbf{r}_0 = -\Delta^{-1}\mathbf{z}_0$

if $\|\mathbf{r}_0\|_2 \leq \epsilon_{\text{tol}}$ **then** return $\mathbf{t}_0$

$\gamma_0 = \mathbf{r}_0^T \mathbf{z}_0$

$\mathbf{d}_0 = \mathbf{r}_0$

$\mathbf{c}_0 = \mathbf{A}\mathbf{d}_0$

$\alpha_0 = \frac{\gamma_0}{\mathbf{r}_0^T \mathbf{c}_0}$

$\mathbf{t}_1 = \mathbf{t}_0 + \alpha_0 \mathbf{d}_0$ and $\mathbf{z}_1 = \mathbf{z}_0 - \alpha_0 \mathbf{c}_0$

$\mathbf{r}_1 = \Delta^{-1} \mathbf{z}_1$

$n = 1$

while $\|\mathbf{r}_n\|_2 > \epsilon_{\text{tol}}$ **do**

$\beta_n = \frac{\mathbf{r}_n^T \mathbf{z}_n}{\gamma_{n-1}}$

$\gamma_n = \gamma_{n-1} \beta_n$

$\mathbf{d}_n = \mathbf{r}_n + \beta_n \mathbf{d}_{n-1}$

$\mathbf{c}_n = \mathbf{A}\mathbf{d}_n$

$\alpha_n = \frac{\gamma_n}{\mathbf{r}_n^T \mathbf{c}_n}$

$\mathbf{t}_{n+1} = \mathbf{t}_n + \alpha_n \mathbf{d}_n$ and $\mathbf{z}_n = \mathbf{z}_n - \alpha_n \mathbf{c}_n$

$\mathbf{r}_{n+1} = \Delta^{-1} \mathbf{z}_n$

$n = n + 1$

end

return $\mathbf{t}_n$

end

Figure 1. Algorithm for the preconditioned conjugate gradient for LinCC

use a preconditioner; it is unsurprising that the best general choice of preconditioner is the matrix $\mathbf{A}_{\text{MP}}$. The pseudocode for the preconditioned conjugate gradient is in Fig. 1.

For preconditioned CG, the memory requirements are the storage of $\mathbf{t}, \mathbf{r}, \mathbf{d}, \mathbf{z}$ [11], or

$$\text{Memory} = 5 \frac{o(o-1)v(v-1)}{4}. \quad (9)$$

Therefore, if we take the example of a 100 electron, 1000 unoccupied functions system, instead of the 20 gigabytes quoted before, one needs at least 100 gigabytes to perform the iterations.

3 Task-based computation

The programming model that this project takes is task-based. Rather than having the program run through a procedural set of commands and place them individually on different processors, the task-based model is focused on *first* breaking the problem into distributable chunks. At the most basic level, each chunk carries information about the location of the data it operates on, and on any additional tasks that must be performed after that chunk. By construction, chunks can operate asynchronously. Because no one chunk represents a substantial portion of the calculation, if any one chunk does not report back in a reasonable amount of time, it can be assumed dead and a new copy of that chunk can be spawned and run. This immediately introduces robustness to faults, including processor faults that may be difficult or impossible to detect. This model is a generalization of the object-oriented programming model because it is designed around not just conceptual dependence (objects) but also explicit data dependencies. It is this latter type of dependency that is critical for parallel scaling.

By examining the structure of the coupled-cluster equations, within the preconditioned conjugate gradient form discussed above, one can see that the main floating point kernel is a “tensor contraction,” a multidimensional generalization of a matrix multiply. Underlying all tensor contraction schemes in practice is index reordering coupled with matrix multiplication. Therefore, it is important to first build a scalable, task-based, matrix multiply. This intermediate step also teaches us about the structure of the problem and introduces most of the difficulties we would expect to face as we scale up to a full coupled-cluster code.

3.1 Matrix multiply

The standard method to implement a matrix multiply is to translate the definition,

$$C_{ij} = \sum_k A_{ik} B_{kj}, \quad (10)$$

directly into code, Fig. 2. Unfortunately, this approach is problematic when one attempts to parallelize it. While there are tuned parallel algorithms, as one gets to larger and larger matrices and has data dependencies between the inputs and output, simple implementations of this algorithm are doomed to fail. Distributing the data to each processor would become the dominant term in the time for the method and that would get worse as the computer got larger, not better.

One way to start to break the problem of dependencies between data is to replicate sub-blocks of the matrices. That is, increase the total amount of storage necessary across the whole machine to reduce the amount of communication. This idea then introduces the idea of overlapping of communication and multiple processes contributing to the calculation of a single term in the output array.

To be able to achieve this concept, we first design a hierarchical data structure. All of the necessary classes are shown in the appendix. The workhorse of the algorithm is a *SubMatrix*.

```

begin
  for ( $i = 0; i \leq m; i++$ ) do
    for ( $j = 0; j \leq n; j++$ ) do
      for ( $k = 0; k \leq p; k++$ ) do
         $C[i][j] += A[i][k] * B[k][j]$ 
      end
    end
  end
end

```

Figure 2. Algorithm for a simple matrix multiply

A SubMatrix defines some portion of a full array, up to an including the full array. However, it includes no data itself. Instead, a SubMatrix can be recursively subdivided until it is of a size less than or equal to a maximal block size that is externally defined. A SubMatrix of that block size (a minimal SubMatrix) can be converted into a MatrixBlock – actual data that calculations can be performed with.

The next requirement is to define discrete units of work. This is done by using a the fundamental object in this model, a **Task**. A task has some work associated with it and carries pointers to data on which the task will act. To specify where on the machine the task will act, tasks are controlled through a **Scheduler**, which assigns tasks to processors, keeps track of what has been completed and not, and kicks off the calculation. The Scheduler also has access to the **Storer**, which converts references to data (SubMatrices) into actual arrays of data (MatrixBlocks). All incrementing of the output matrix is handled “close to disk” through the Storer as well.

In a multi-threaded environment, one must generalize the concept of a Scheduler. Generally, the idea is to use a work-stealing scheduling system. Each process has an individual scheduler as well as connections with a global scheduler. Furthermore, as we move to a more parallel environment, multiple Storers will be necessary as well. By separating out these three roles: computation (Task), communication (Scheduler), storage (Storer), one can hope to achieve good scaling.

The data model that we are using neatly breaks the tasks into two sets: (1) recursive subdivision of SubMatrices to set up multiplication (task “MxmDecompose”) and (2) actually performing a matrix multiply of MatrixBlocks, using the “Simple matrix multiply” of Fig. 2 (task “Mxm-Chunk”). The algorithm for the recursive subdivision of the matrices is shown in Fig. 3.

The Scheduler distinguishes between two sets of tasks, by dynamically checking the type of the task. This allows a single scheduler structure to hold both recursive (non-data) tasks and non-recursive, computation. The part of the scheduler that actually runs the tasks is shown in Fig. 4.

The program workflow is as follows. First one initializes various matrices with data. Then one starts the scheduler and gives it the full (not subdivided) matrices. The scheduler then schedules a single instance of the MxmDecompose function (Fig. 3). Then the scheduler is told to run its stack,

```

void MxmDecompose::run(const SchedulerPtr &sched)
{
    // When splitting indices, it is up to the
    // data type specialization to split in consistent ways.
    // E.g., A must split rows in the same way that C splits rows.
    vector<vector<SubMatrixPtr> > Asplit, Bsplit, Csplit;
    bool split = false;
    int Arowdim = 1, Acoldim = 1, Browdim = 1, Bcoldim = 1,
        Crowdim = 1, Ccoldim = 1;
    if (A_>can_split_row_index()) {Arowdim = 2; Crowdim = 2; split=true;}
    if (B_>can_split_col_index()) {Bcoldim = 2; Ccoldim = 2; split=true;}
    if (A_>can_split_col_index()) {Acoldim = 2; Browdim = 2; split=true;}
    do_resize(Asplit,Arowdim,Acoldim); do_resize(Bsplit,Browdim,Bcoldim);
    do_resize(Csplit,Crowdim,Ccoldim);

    pair<SubMatrixPtr, SubMatrixPtr> tmp;

    // First split into  $[C_{00} \ C_{10}]^T = [A_{00} \ A_{10}]^T B_{00}$ , if possible
    if (A_>can_split_row_index()) {
        tmp = A_>split_row_index();
        Asplit[0][0] = tmp.first; Asplit[1][0] = tmp.second;
        tmp = C_>split_row_index();
        Csplit[0][0] = tmp.first; Csplit[1][0] = tmp.second;
    }
    else { Asplit[0][0] = A_; Csplit[0][0] = C_; }

    // Repeat to split columns and intermediate indices
    // ... (code skipped) ...

    if (split == true) { // Hand off the work to the scheduler.
        for (int i=0; i<Csplit.size(); i++) {
            for (int j=0; j<Csplit[i].size(); j++) {
                for (int k=0; k<Asplit[i].size(); k++) {
                    sched->schedule(MxmDecompose::construct(
                        Csplit[i][j],Asplit[i][k],Bsplit[k][j]));
                } } } }
    else { //Schedule construction of MxmChunk
        sched->schedule(MxmChunk::construct(C_,A_,B_)); }
}

```

Figure 3. Algorithm for recursive task-based block subdivision for matrix multiplication

```

void Scheduler::run(const StorerPtr &stor)
{
    while (!scheduled_tasks_.empty()) {
        //while not empty pop
        TaskPtr task = scheduled_tasks_.top();
        scheduled_tasks_.pop();
        SchedulerPtr sched = shared_from_this();

        //try to cast to a different task types
        shared_ptr<TaskOps> taskops = dynamic_pointer_cast<TaskOps>(task);
        shared_ptr<TaskData> taskdata = dynamic_pointer_cast<TaskData>(task);
        if (taskops) {
            // TaskOps cast succeeded, no data access,
            // run without any additional parameters
            taskops->run(sched);
        }
        else if (taskdata) {
            // TaskData succeeded, need Data access
            DataReferencePtr Aref,Bref,Cref;
            DataChunkPtr A,B,C;
            // Extract the relevant DataReferences from TaskData
            Aref = taskdata->A_sub();
            Bref = taskdata->B_sub();
            // Get the data for A and B from storer
            stor->reference_to_chunk(Aref,A);
            stor->reference_to_chunk(Bref,B);
            taskdata->run(sched,C,A,B); // run the matrix multiply
            Cref = taskdata->C_sub();
            // accumulate the resultant C through the storer
            stor->accumulate_chunk(Cref,C);
        }
    }
}

```

Figure 4. Algorithm for scheduler to run different tasks and access the memory controlled by the storer.

so it begins to execute Fig. 4. Running the MxmDecompose, in turn, pushes several new tasks – MxmDecompose for subdivided matrices – onto the Scheduler stack. These are run recursively until a non-subdividable matrix is reached, at which point the Storer is contacted for data. This data is then passed to a simple matrix multiply (MxmChunk), which passes back its result. The resulting matrix is then sent to the storer for storage. These final steps are illustrated in Fig. 5.

Figure 5. Workflow for single operation in task-based operation. The order of operations corresponds to the numbers. Single arrows are non-blocking read operations while the lone double arrow [(7)] is a blocking write. Dashed lines indicate that it is one of two possible operations. Bold letters are matrices, a small letter ‘r’ indicates a reference to actual data, while the letter without the ‘r’ is the data itself.

3.2 General model for CC iteration

When we move from the basic operation of a matrix multiplication to the more complicated tensor contractions in CC much remains the same. Instead of a single form of matrix multiplication, we

now have several differing types of contractions, each of which have slightly different data dependencies and structures. This specialization was studied in an MPI context in [2]. At the core of each of these specializations is a matrix multiplication, where the index subgroups are differing. Therefore, we can easily extend the MxmDecompose and MxmChunk tasks to specialized TensorContractDecompose and TensorContractChunk tasks, which will depend on the exact type of contraction being used.

Using this formulation and fully direct integrals there would only be one level of global synchronization: each iteration of the amplitude equations depends on the prior iteration. One could let subsets of the amplitude equations “get ahead” of the rest of the program, but due to the tight coupling between equations, that would quickly stall. The more important activity is to be able to determine if all the contributions to a t amplitude have been performed. Assuming that we are able to use a balanced partitioning of the t amplitudes and their equations, each term that contributes to a given equation should take roughly the same amount of time. Therefore, if on iteration $n + 1$ a process finds it is missing the a certain t amplitude from iteration n , this serves as a diagnostic of a fault for that term. The scheduler can then be notified to restart the calculation of the missing term, without endlessly waiting for confirmation of a fault.

In LinCCD, each term in the amplitude equation is a simple binary contraction. Therefore, there is no need to define extra intermediates as there is in the nonlinear CC equations. However, in the nonlinear case, it would be necessary to determine a way to handle triple products of the form

$$t_{ij}^{ab(n+1)} + = \sum_{k\ell cd} t_{ik}^{ac(n)} \langle k\ell || cd \rangle t_{\ell j}^{db(n)}. \quad (11)$$

It has been proven [12] that every fully iterative coupled-cluster form can, with the proper choice of intermediates, be written in binary form. There will be a trade-off between forming intermediates and performing the triple summations in a single operation. Forming intermediates will lead to more global synchronization, because they become t amplitude-like quantities that all processes will want access to. However, the storage requirements on an individual processor and the added computation necessary to do the direct triple product may be prohibitive. It seems likely that both options should be built into the code, which might recognize the optimal approach given the specifics of the computer system.

4 Conclusion and Further Work

The task-based model is an ideal one for scaling tightly-coupled, large, dense data problems like coupled-cluster theory. Currently, we have not demonstrably shown that one can use this model to fully implement a fault-tolerant and optimally scaling code. The most important kernel of this process – the matrix multiply – is well-understood. We are currently working on understanding the proper decompositions for the task-based CC. The primary requirement for scaling is the ability to separate out the individual contributions such that the tasks can run asynchronously as much as possible. Unlike the individual matrix multiply, by the iterative nature of the CC equations, global synchronization will be necessary between iterative cycles. However, that can serve as a fault diagnostic, so it does not represent a substantial penalty.

The most important work that needs to be done is to begin to apply these algorithms and codes to problems that would challenge the current MPI implementations. That task requires using a runtime that will work well with the task-based code. At that point, this work will build more directly on the work that the FAST-OS HARE project has completed and will allow for definitive proof-of-concept for the task-based programming model as it applies to quantum chemistry.

References

- [1] R. J. Bartlett and M. Musiał, *Rev. Mod. Phys.* **79**, 291 (2007).
- [2] V. F. Lotrich, N. Flocke, M. Ponton, A. D. Yau, S. A. Perera, E. Deumens, and R. J. Bartlett, *J. Chem. Phys.* **128**, 194104 (2008).
- [3] N. Flocke and V. F. Lotrich, *J. Comput. Chem.* **29**, 2722 (2008).
- [4] T. Kuś, V. F. Lotrich, and R. J. Bartlett, *J. Chem. Phys.* **130**, 124122 (2009).
- [5] A. G. Taube and R. J. Bartlett, *J. Chem. Phys.* **130**, 144112 (2009).
- [6] P. Pulay, *Chem. Phys. Lett.* **73**, 393 (1980).
- [7] G. D. Purvis III and R. J. Bartlett, *J. Chem. Phys.* **75**, 1284 (1981).
- [8] R. J. Bartlett and E. J. Brändas, *J. Chem. Phys.* **56**, 5467 (1972).
- [9] R. J. Bartlett and E. J. Brändas, *J. Chem. Phys.* **59**, 2032 (1973).
- [10] P. E. S. Wormer, F. Visser, and J. Paldus, *J. Comput. Phys.* **48**, 23 (1982).
- [11] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edition, 2000.
- [12] S. A. Kucharski and R. J. Bartlett, *Theor. Chim. Acta* **80**, 387 (1991).

A Class Definitions

```
typedef shared_ptr<DataChunk> DataChunkPtr;  
typedef shared_ptr<DataReference> DataReferencePtr;  
typedef shared_ptr<Matrix> MatrixPtr;  
typedef shared_ptr<MatrixBlock> MatrixBlockPtr;  
typedef shared_ptr<MxmDecompose> MxmDecomposePtr;  
typedef shared_ptr<Scheduler> SchedulerPtr;  
typedef shared_ptr<Storer> StorerPtr;  
typedef shared_ptr<SubMatrix> SubMatrixPtr;  
typedef shared_ptr<Task> TaskPtr;
```

Figure A.1. Typedefs

```
class Scheduler: public enable_shared_from_this<Scheduler>{  
    stack<TaskPtr> scheduled_tasks_; // single-threaded  
public:  
    ~Scheduler();  
    static SchedulerPtr construct();  
    void schedule(const TaskPtr &task);  
    void run(const StorerPtr &stor);  
};
```

Figure A.2. Scheduler class

```

class Storer {
    globalsize_t unique_id_; // unique among all Storer objects
    globalsize_t next_data_; // used to get unique id among all storers
    int nstorer_;
    typedef map<DataReferencePtr, DataChunkPtr,
        DataReferencePtrCompare > local_data_t;
    local_data_t local_data_;
    Storer();
public:
    static StorerPtr construct();
    ~Storer();
    void reference_to_chunk(const DataReferencePtr &ref, DataChunkPtr &data);
    void accumulate_chunk(const DataReferencePtr &ref,
        const DataChunkPtr &data);
    void init(const DataReferencePtr &ref, const DataChunkPtr &data);
    data_id_t unique_id();
};

```

Figure A.3. Storer class

```

class Task {
public:
    virtual ~Task();
    virtual void run(const SchedulerPtr &sched);
};

class TaskOps : public Task {
public:
    virtual ~TaskOps();
};

class TaskData : public Task {
public:
    virtual ~TaskData();
    virtual void run(const SchedulerPtr &sched, DataChunkPtr &C,
        const DataChunkPtr &A, const DataChunkPtr &B) = 0;
    virtual DataReferencePtr A_sub() = 0;
    virtual DataReferencePtr B_sub() = 0;
    virtual DataReferencePtr C_sub() = 0;
};

```

Figure A.4. Task classes

```

class MxmDecompose: public TaskOps {
    SubMatrixPtr A_, B_, C_;
public:
    static MxmDecomposePtr construct(const SubMatrixPtr &C,
        const SubMatrixPtr &A, const SubMatrixPtr &B);
    void run(const SchedulerPtr &sched);
};

// This performs the floating point computation.
class MxmChunk: public TaskData {
    SubMatrixPtr A_, B_, C_;
public:
    static MxmChunkPtr construct(const SubMatrixPtr &C,
        const SubMatrixPtr &A, const SubMatrixPtr &B);
    // You need to be able to get at the submatrices A_,B_ and C_
    // so that the storer can handle them.
    DataReferencePtr A_sub();
    DataReferencePtr B_sub();
    DataReferencePtr C_sub();
    void run(const SchedulerPtr &sched, DataChunkPtr &C_chunk,
        const DataChunkPtr &A_chunk, const DataChunkPtr &B_chunk);
};

```

Figure A.5. Matrix Multiply Task classes


```

// Base class for global data objects.
// This describes data, it does not store it.
class Data {
    data_id_t data_id_;
protected:
    Data(const StorerPtr &stor);
public:
    virtual ~Data();
    // Returns an estimate of the global size of this object in bytes.
    virtual globalsize_t globalsize_hint() const = 0;
    // Returns a hint of the number of blocks of data.
    virtual globalsize_t nblock_hint() const = 0;
    // Return an identifier which is unique for this data object.
    const data_id_t &data_id() const;
};

// A reference to a portion of the data.
class DataReference {
    data_id_t data_id_;
protected:
    DataReference(const data_id_t & data_id_in);
public:
    // The data_id() is used to uniquely determine the data reference.
    const data_id_t &data_id() const;
    virtual ~DataReference();
};

// A reference to data. This stores the data (possibly a copy).
class DataChunk {
    double *data_;
    globalsize_t data_size_;
protected:
    // only allow a derived class to create a DataChunk
    DataChunk(globalsize_t data_size);
public:
    virtual ~DataChunk();
    double& operator[] (unsigned index);
    const double& operator[] const (unsigned index);
    globalsize_t& data_size ();
};

```

Figure A.6. Data Classes

```

class Matrix: public Data {
    globalsize_t nrow_, ncol_;
    localsize_t blocksize_;
    Matrix(const StorerPtr &stor, globalsize_t nrow, globalsize_t ncol,
        localsize_t blocksize);
public:
    static MatrixPtr construct(const StorerPtr &stor, globalsize_t nrow,
        globalsize_t ncol, localsize_t blocksize);
    globalsize_t nrow() const;
    globalsize_t ncol() const;
    globalsize_t globalsize_hint() const;
    globalsize_t nblock_hint() const;
    localsize_t blocksize() const;
};

// A SubMatrix is a reference to a portion of a matrix.
// It is subdividable or can be used to obtain a MatrixBlock
class SubMatrix: public DataReference,
    public enable_shared_from_this<SubMatrix>
{
    MatrixPtr matrix_;
    globalsize_t row_block_, col_block_;
    globalsize_t row_blocksize_, col_blocksize_;
    SubMatrix(const data_id_t &data_id_in);
public:
    static SubMatrixPtr construct(const MatrixPtr &matrix);
    static SubMatrixPtr construct(const MatrixPtr &matrix,
        globalsize_t row_blocksize, globalsize_t row_block,
        globalsize_t col_blocksize, globalsize_t col_block);
    bool can_split_row_index() const;
    bool can_split_col_index() const;
    globalsize_t row_blocksize() const;
    globalsize_t col_blocksize() const;
    MatrixPtr matrix() const;
    pair<SubMatrixPtr, SubMatrixPtr > split_row_index() const;
    pair<SubMatrixPtr, SubMatrixPtr > split_col_index() const;
    void initialize_storage(const StorerPtr &stor, const double &val);
    void print_data(const StorerPtr &stor);
};

```

Figure A.7. Matrix Reference classes

```

// A matrix block contains the storage for a block of the matrix.
class MatrixBlock: public DataChunk {
    MatrixPtr matrix_;
    localsize_t row_blocksize_, col_blocksize_;
    MatrixBlock(globalsize_t data_size);
public:
    static MatrixBlockPtr construct(const SubMatrixPtr &submatrix);
    globalsize_t row_blocksize() const;
    globalsize_t col_blocksize() const;
    // Return an identifier which is unique for this data object.
    const data_id_t &data_id() const;
    void initialize_storage(const double &val);
};

```

Figure A.8. Matrix classes

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic)
1	MS 0123	D. Chavez, LDRD Office, 1011