Inspector-Executor Load Balancing Algorithms for Block-Sparse Tensor Contractions

David Ozog,* Jeff R. Hammond,† James Dinan,† Pavan Balaji,† Sameer Shende,* Allen Malony*

*University of Oregon {*ozog, sameer, malony*}*@cs.uoregon.edu*
†Argonne National Laboratory {*jhammond, dinan, balaji*}*@anl.gov*

***Abstract*—Developing effective yet scalable load-balancing methods for irregular computations is critical to the successful application of simulations in a variety of disciplines at petascale and beyond. This paper explores a set of static and dynamic scheduling algorithms for block-sparse tensor contractions within the NWChem computational chemistry code for different degrees of sparsity (and therefore load imbalance). In this particular application, a relatively large amount of task information can be obtained at minimal cost, which enables the use of static partitioning techniques that take the entire task list as input. However, fully static partitioning is incapable of dealing with dynamic variation of task costs, such as from transient network contention or operating system noise, so we also consider hybrid schemes that utilize dynamic scheduling within subgroups. These two schemes, which have not been previously implemented in NWChem or its proxies (i.e. quantum chemistry mini-apps) are compared to the original centralized dynamic load-balancing algorithm as well as improved centralized scheme. In all cases, we separate the scheduling of tasks from the execution of tasks into an inspector phase and an executor phase. The impact of these methods upon the application is substantial on a large InfiniBand cluster: execution time is reduced by as much as 50% at scale. The technique is applicable to any scientific application requiring load balance where performance models or estimations of kernel execution times are available.**

***Keywords*-Dynamic Load Balancing, Static Partitioning, Tensor Contractions, Quantum Chemistry, Global Arrays**

I. Introduction

Load balancing of irregular computations is a serious challenge for petascale and beyond because the growing number of processing elements (PEs) – which now exceeds 1 million on systems such as Blue Gene/Q – makes it increasingly more difficult to find a work distribution that keeps all the PEs busy for the same period of time. Additionally, any form of centralized dynamic load balancing, such as master-worker or a shared counter (e.g., Global Arrays' `NXTVAL` [20]), becomes a bottleneck. The competition between the need to extract million-way parallelism from applications and the need to avoid load-balancing strategies that have components which scale with the number of PEs motivates us to develop new methods for scheduling collections of tasks with widely varying cost; the motivating example in this case is the NWChem computational chemistry package. One of the major uses of NWChem is to perform quantum many-body theory methods such as coupled cluster (CC). Popular among chemists are methods such as CCSD(T) and CCSDT(Q) because of their high accuracy at relatively modest computational cost.[1] In these methods, (T) and (Q) refer to perturbative a posteriori corrections to the energy that are highly scalable (roughly speaking, they resemble MapReduce), while the iterative CCSD and CCSDT steps have much more communication and load imbalance. Thus, this paper focuses on the challenge of load balancing these iterative procedures. However, the algorithms we describe can be applied to noniterative procedures as well.

In this paper, we demonstrate that the inspector-executor model (IE) is effective in reducing load imbalance as well as eliminating the overhead from the `NXTVAL` dynamic load balancer. Additionally, we find that IE algorithms are effective when used in conjunction with static partitioning, which is done both with task performance modeling and empirical measurements. We present three different IE load-balancing techniques which each display unique properties when applied to different chemical problems. By examining symmetric (highly sparse) versus nonsymmetric (less sparse) molecular systems in the context of these three methods, we better understand how to open doors to new families of highly adaptable load-balancing algorithms on modern multicore architectures.

II. Background

In this section, we describe NWChem, coupled-cluster methods, the Global Arrays programming model, and the Tensor Contraction Engine.

A. NWChem

NWChem [9] is the DOE flagship computational chemistry package, which supports most of the widely used methods across a range of accuracy scales (classical molecular dynamics, ab initio molecular dynamics, molecule density-functional theory (DFT), perturbation theory, coupled-cluster theory, etc.) and many of the most popular supercomputing architectures (InfiniBand clusters, Cray XT and XE, and IBM Blue Gene). Among the most popular methods in NWChem are the DFT and CC methods, for which NWChem is one of the few codes (if not the only code) that support these features for massively parallel systems. Given the steep computational cost of CC methods, the scalability of NWChem in this context is extremely important for real science. Many chemical problems

[1] The absolute cost of these methods is substantial when compared with density-functional theory (DFT), for example, but this does not discourage their use when high accuracy is required.

related to combustion, energy conversion and storage, catalysis, and molecular spectroscopy are untenable without CC methods on supercomputers. Even when such applications are feasible, the time to solution is substantial; and even small performance improvements have a significant impact when multiplied across hundreds or thousands of nodes.

B. Coupled-Cluster Theory

Coupled-cluster theory [44] is a quantum many-body method that solves an approximate Schrödinger equation resulting from the CC ansatz,

$$|\Psi_{CC}\rangle = \exp(T)|\Psi_0\rangle,$$

where $|\Psi_0\rangle$ is the reference wavefunction (usually a Hartree-Fock Slater determinant) and $\exp(T)$ is the cluster operator that generates excitations out of the reference. Please see Refs. [11], [6] for more information.

A well-known hierarchy of CC methods exists that provides increasing accuracy at increased computational cost [5]:

$$\cdots < CCSD < CCSD(T) < CCSDT < CCSDT(Q) < CCSDTQ < \cdots .$$

The simplest CC method that is generally useful is CCSD [35], has a computational cost of $O(N^6)$ and storage cost of $O(N^4)$, where N is a measure of the molecular system size. The "gold standard" CCSD(T) method [45], [37], [36], [42] provides much higher accuracy using $O(N^7)$ computation but without requiring (much) additional storage. CCSD(T) is a very good approximation to the full CCSDT [32], [47] method, which requires $O(N^8)$ computation and $O(N^6)$ storage. The addition of quadruples provides chemical accuracy, albeit at great computational cost. CCSDTQ [27], [28], [33] requires $O(N^{10})$ computation and $O(N^8)$ storage, while the perturbative approximation to quadruples, CCSDT(Q) [26], [29], [8], [22], reduces the computation to $O(N^9)$ and the storage to $O(N^6)$. Such methods have recently been called the "platinum standard" because of their unique role as a benchmarking method that is significantly more accurate than CCSD(T) [41].

An essential aspect of an efficient implementation of any variant of CC is the exploiting of symmetries, which has the potential to reduce the computational cost and storage required by orders of magnitude. Two types of symmetry exist in molecular CC: spin symmetry [16] and point-group symmetry [10]. Spin symmetry arises from quantum mechanics. When the spin state of a molecule is a singlet, some of the amplitudes are identical; and thus we need store and compute only the unique set of them. The impact is roughly that N is reduced to $N/2$ in the cost model, which implies a reduction of one to two orders of magnitude in CCSD, CCSDT, and CCSDTQ. Point-group symmetry arise from the spatial orientation of the atoms. For example, a molecule such as benzene has the symmetry of a hexagon, which includes multiple reflection and rotation symmetries. These issues are discussed in detail in Refs. [43], [17]. The implementation of degenerate group symmetry in CC is difficult; and NWChem, like most codes, does not support it. Hence, CC calculations cannot exploit more than the 8-fold symmetry of the D_{2h} group, but this is still a substantial reduction in computational cost.

While the exploitation of symmetries can substantially reduce the computational cost and storage requirements of CC, these methods also introduce complexity in the implementation. Instead of performing dense tensor contractions on rectangular multidimensional arrays, point-group symmetries lead to block diagonal structure, while spin symmetries lead to symmetric blocks where only the upper or lower triangle is unique. This is one reason that one cannot, in general, directly map CC to dense linear algebra libraries. Instead, block-sparse tensor contractions are mapped to BLAS at the PE level, leading to load imbalance and irregular communication between PEs. *Ameliorating the irregularity arising from symmetries in tensor contractions is one of the major goals of this paper.*

C. Global Arrays

Global Arrays (GA) [30], [31] is a PGAS-like global-view programming model that provides the user with a clean abstraction for distributed multidimensional arrays with one-sided access (Put, Get, and Accumulate). GA provides numerous additional functionalities for matrices and vectors, but these are not used for tensor contractions because of the nonrectangular nature of these objects in the context of CC. The centralized dynamic load balancer `NXTVAL` was inherited from TCGMSG, a pre-MPI communication library. Initially, the global shared counter was implemented by a polling process spawned by the last PE, but now it uses ARMCI remote fetch-and-add, which goes through the ARMCI communication helper thread. Together, the communication primitives of GA and `NXTVAL` can be used in a template for-loop code that is general and can handle load imbalance, at least until such operations overwhelm the computation because of work starvation or communication bottlenecks that emerge at scale. A simple variant of the GA "get-compute-update" template is shown in Alg. 1. For computations that are naturally load balanced, one can use the GA primitives and skip the calls to `NXTVAL`, a key feature when locality optimizations are important, since `NXTVAL` has no ability to schedule tasks with affinity to their input or output data. This is one of the major downsides of many types of dynamic load-balancing methods—they lack the ability to exploit locality in the same way that static schemes do.

While there exist several strategies for dynamically assigning collections of tasks to processor cores, most present a trade-off between the quality of the load balance and scaling to a large number of processors. Centralized load balancers can be effective at producing evenly distributed tasks, but they can have substantial overhead. Decentralized alternatives such as work stealing [13], [3] may not achieve the same degree of load balance, but their distributed nature can reduce the overhead substantially.

Algorithm 1 The canonical Global Arrays programming template for dynamic load balancing. `NXTVAL()` assigns each loop iteration number to a process that acquires the underlying lock and atomically increments a global counter. This counter is located in memory on a single node, potentially leading to considerable network communication. One can easily generalize this template to multidimensional arrays, multiple loops, and blocks of data, rather than single elements. As long as the time spent in `FOO` is greater than that spent in `NXTVAL`, `Get`, and `Update`, this is a scalable algorithm.

```
Global Arrays: A, B
Local Buffers: a, b
count = 1
next = NXTVAL()
for i = 1 : N do
  if ( next == count ) then
    Get A(i) into a
    b = FOO(a)
    Update B(i) with b
    next = NXTVAL()
  end if
  count = count + 1
end for
```

Algorithm 2 Pseudocode for the default TCE implementation of Eq. 1. For clarity, aspects of the Alg. 1 DLB template are omitted.

```
Tiled Global Arrays: X, Y, Z
Local Buffers: x, y, z
for all i, j, k ∈ Otiles do
  for all a, b, c ∈ Vtiles do
    if NXTVAL() == count then
      if Symm(i, j, k, a, b, c) == True then
        Allocate z for Z(i, j, k, a, b, c) tile
        for all d, e ∈ Vtiles do
          if Symm(i, j, d, e) == True then
            if Symm(d, e, k, a, b, c) == True then
              Fetch X(i, j, d, e) into x
              Fetch Y(d, e, k, a, b, c) into y
              Contract z(i, j, k, a, b, c)+=
                        x(i, j, d, e)*y(d, e, k, a, b, c)
            end if
          end if
        end for
        Accumulate z into Z(i, j, k, a, b, c)
      end if
    end if
  end for
end for
```

D. Tensor Contraction Engine

The Tensor Contraction Engine (TCE) [21], [4] is a project to automate the derivation and parallelization of quantum many-body methods such as CC. As a result of this project, the first parallel implementations of numerous methods were created and applied to larger scientific problems than previously possible. The original implementation in NWChem by Hirata was general (i.e., lacked optimizations for known special cases) and required significant tuning to scale CC to more than 1,000 processes in NWChem [24]. The tuning applied to the TCE addressed essentially all aspects of the code, including more compact data representations (spin-free integrals are antisymmetrized on the fly in the standard case), reduction in communication by applying additional fusion that the TCE compiler was not capable of applying. In some cases, dynamic load balancing (DLB) was eliminated altogether when the cost of a diagram (a single term in the CC eqns.) was insignificant at scale and DLB was unnecessary overhead.

The TCE code generator uses a stencil that is a straightforward generalization of Alg. 2. For a term such as

$$Z(i,j,k,a,b,c) += \sum_{d,e} X(i,j,d,e) * Y(d,e,k,a,b,c), \quad (1)$$

which is a bottleneck in the solution of the CCSDT equations, the data is tiled over all the dimensions for each array and distributed across the machine in a one-dimensional global array. Multidimensional global arrays are not useful here because they do not support block sparsity or index permutation symmetries. Remote access is implemented by using a lookup table for each tile and a GA `Get` operation. The global data layout is not always appropriate for the local computation, however; therefore, immediately after the `Get` operation completes, the data is rearranged into the appropriate layout for the computation. Alg. 2 gives an overview of a distributed tensor contraction in TCE. For compactness of notation, the `Fetch` operation combines the remote `Get` and local rearrangement. The `Symm` function is a condensation of a number of logical tests in the code that determine whether a particular tile will be nonzero. These tests consider the indices of the tile and not any indices within the tile because each tile is grouped such that the symmetry properties of all its constitutive elements are identical. In Alg. 2, the indices given for the local buffer contraction are the tile indices, but these are merely to provide the ordering explicitly. Each tile index represents a set of contiguous indices so the contraction is between multidimensional arrays, not single elements. However, one can think of the local operation as the dot product of two tiles.

III. Motivation and Design

In this section we discuss our motivation for developing inspector-executor (IE) algorithms, the implementation of the IE in the TCE-CC module of NWChem, and the design of our cost partitioning strategy. We begin by characterizing the function of a simple version of the inspector, then augment the IE model by incorporating performance models of the dominant computational kernels. The performance models provide estimations of task execution time to be fed into the static partitioner, then to the executor. While the IE design and implementation is described within the context of

Algorithm 3 Pseudocode for the inspector used to implement Eq. 1, simple version.

```
for all i, j, k ∈ Otiles do
  for all a, b, c ∈ Vtiles do
    if Symm(i, j, k, a, b, c) == True then
      Add Task (i, j, k, a, b, c) to TaskList
    end if
  end for
end for
```

Algorithm 4 Pseudocode for the inspector used to implement Eq. 1, with cost estimation and static partitioning.

```
for all i, j, k ∈ Otiles do
  for all a, b, c ∈ Vtiles do
    if Symm(i, j, k, a, b, c) == True then
      Add Task (i, j, k, a, b, c, d, e, w) to TaskList
      cost_w = SORT4_performance_model_estm(sizes)
      for all d, e ∈ Vtiles do
        if Symm(i, j, d, e) == True then
          if Symm(d, e, k, a, b, c) == True then
            cost_w = cost_w + ...
            SORT4_performance_model_estm(sizes)
            cost_w = cost_w + ...
            DGEMM_performance_model_estm(m, n, k)
            Compute various SORT4 costs
          end if
        end if
      end for
    end if
  end for
end for
myTaskList = Static_Partition(TaskList)
```

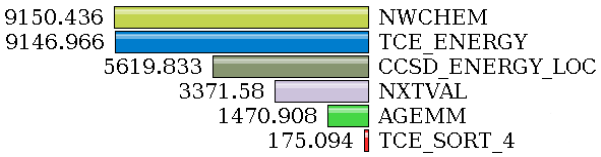

Fig. 1. Average inclusive-time (in seconds) profile of a 14-water monomer CCSD simulation with the aug-cc-PVDZ basis for 861 MPI processes across 123 Fusion nodes connected by InfiniBand. The NXTVAL routine consumes 37% of the entire computation. This profile was made using TAU [38]; for clarity, some subroutines were removed.

NWChem's TCE-CC module, the methods can be applied to any irregular application where reasonably accurate estimation of kernel execution times is possible, either analytically or empirically. In our NWChem implementation, the inspector initially assigns cost estimations by applying performance models of computational kernels for the first iteration, then by doing in situ execution time measurements of tasks for subsequent iterations.

A. Inspector/Executor

When using Alg. 1 in TCE-based CC simulations, the inherently large number of computational tasks (typically hundreds of thousands) each require a call to NXTVAL for dynamic load balancing. Very large systems can potentially require many billions of fine-grained tasks, but the granularity can be controlled by increasing the tile size. Although NXTVAL overhead can be limited by increasing the tile size, it is far more difficult to balance such a coarse-grained task load while preventing starvation, so typically a large number of small-sized tasks is desirable. However, the average time per call to NXTVAL increases with the number of processes (as shown below), so too many tasks is detrimental for strong scaling. This is the primary motivation for implementing the IE.

This increase in time per call to NXTVAL is primarily caused by contention on the memory location of the counter, which performs atomic read-modify-write (RMW) operations (in this case the addition of 1) using a mutex lock. For a given number of total incrementations (i.e., a given number of tasks), when more processes do simultaneous RMWs, on average they must wait longer to access to the mutex. This effect is clearly displayed in a flood-test microbenchmark (Fig. 2) where a collection of processes calls NXTVAL several times (without doing any other computation). In this test, only off-node processes are allowed to increment the counter (via a call to ARMCI_Rmw); otherwise, the on-node processes would be able to exploit the far more efficient shared-memory incrementation, which occurs on the order of several nanoseconds.The average execution time per call to NXTVAL always increases as more processes are added.

The increasing overhead of scaling with NXTVAL is also directly seen in performance profiles of the tensor contraction routines in NWChem. For instance, Fig. 1 shows a profile of the mean inclusive time for the dominant methods in a CC simulation of a water cluster with 10 molecules. The time spent within NXTVAL accounts for about 37% of the entire simulation. We propose an alternate algorithm which is designed to reduce this overhead by first gathering task information, then evenly assigning tasks to PEs, and finally executing the computations.

We collect relevant tensor contraction task information by breaking the problem into two major components: inspection and execution (similar to Refs. [1], [15], [14]). In its simplest form, the inspector agent loops through relevant components of the parallelized section and collates tasks (Alg. 3). This phase is limited to computationally inexpensive arithmetic operations and conditionals that classify and characterize tasks. Specifically, the first conditional of any particular tensor contraction routine in NWChem evaluates spin and point-group symmetries to determine whether a tile of the tensor contraction has a nonvanishing element [21], as introduced in section II-B. Further along, in a nested loop over common indices, another conditional tests for nonzero tiles of a contraction operand by spin and spatial symmetry. While the inspector's primary purpose is to a create an informative list of tasks to help accomplish load balance during the executor phase, it also has this advantage of revealing sparsity information

Algorithm 5 Pseudocode for the executor used to implement Eq. 1.

```
for all Task ∈ Tasklist do
  Extract (i, j, k, a, b, c) from Task
  Allocate local buffer for Z(i, j, k, a, b, c) tile
  if Symm(i, j, d, e) == True then
    if Symm(d, e, k, a, b, c) == True then
      Fetch X(i, j, d, e) tile into local buffer
      Fetch Y(d, e, k, a, b, c) tile into local buffer
      Z(i, j, k, a, b, c) += X(i, j, d, e)*Y(d, e, k, a, b, c)
      Accumulate Z(i, j, k, a, b, c) buffer into global Z
    end if
  end if
end for
```

by applying spin and spatial symmetry arguments before designating a particular task.

While the following section describes a more complicated version of the inspector, the executor is the same in both cases. The pseudocode for the executor is shown in Alg. 5, where tasks gathered in the inspection phase are simply looped over. The executor contains the inner loop of the default TCE implementation with another set of symmetry conditionals, which are found to always return true for the cases considered in this paper (we cannot exclude the possibility that they may be false in some other cases). Because the inner loop is dense in the sense that no additional sparsity is achieved at that point, it is natural to aggregate these computations into a single task in order to reduce the number of calls to Accumulate, which is more expensive than either Put or Get because it requires a remote computation, not just data movement (which might be done in entirely in hardware with RDMA). Combining all the inner loop computations in a single task also has the effect of implying reuse of the output buffer, which is beneficial if it fits in cache or computation is performed on an attached coprocessor (while heterogeneous algorithms are not part of this work, they are a natural extension of it).

B. Task Cost Characterization

The average time per call to NXTVAL increases with the number of participating PEs, but the IE algorithm as presented so far will improve strong scaling only as much as the proportion of tasks eliminated by the spin and spatial symmetry argument. DLB with NXTVAL for large non-symmetric molecular systems (biomolecules usually lack symmetry) will still be plagued by high overhead due to contention on the global counter despite our simple inspection. In this section we further develop the IE model with the intent to eliminate *all* NXTVAL calls from the entire CC module (pseudocode shown in Alg. 4).

By counting the number of FLOPS for a particular tensor contraction (Fig. 3), we see that a great deal of load imbalance is inherent in the overall computation. The centralized dynamic global counter does an acceptable job of handling this imbalance by atomically providing exclusive task IDs to processes

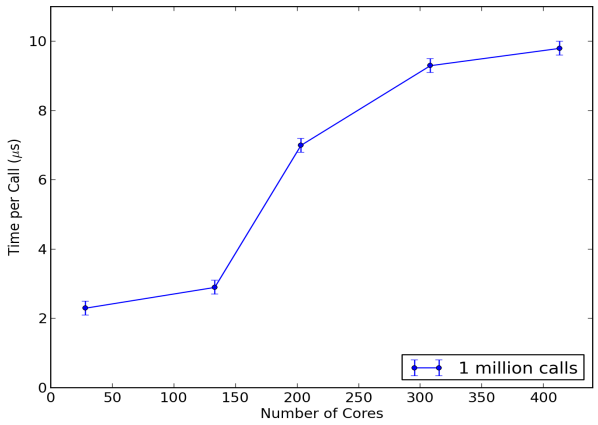

Fig. 2. Flood benchmark showing the execution time per call to NXTVAL for 1 million simultaneous calls. The process hosting the counter is being flooded with messages, so when the arrival rate exceeds the processing rate, buffer space runs out and the process hosting the counter must utilize flow control. The performance gap from 150 to 200 cores is due to this effect occurring at the process hosting the NXTVAL counter.

that request work. To effectively eradicate the centralization, we first need to estimate the cost of all tasks, then schedule the tasks so that each processor is equally loaded.

In the tensor contraction routines, parallel tile-level matrix multiplications and ordering operations execute locally within the memory space of each processor. The kernels that consume the most time doing such computations are the DGEMM and SORT4 subroutines. The key communication routines are the Global Arrays get (ga_get) and accumulate (ga_acc) methods, which consume relatively little time for sizeable and accurate simulations of interest. For this reason we use performance model-based cost estimations for DGEMM and SORT4 to partition the task load of the first iteration of each tensor contraction, as seen in Alg. 4. Details regarding the specific performance models used is beyond the scope of this paper, but others have explored the development of models for such BLAS kernels [34]. During the first iteration of TCE-CC, we measure the time of each task's entire computation (in the executor phase) to capture the costs of communication along with the computation. This new measurement serves as the cost which is fed into the static partitioning phase for subsequent iterations.

C. Static Partitioning

In our IE implementation, the inspector applies the DGEMM and SORT4 performance models to each tile encountered, thereby assigning a cost estimation to each task of the tensor contractions for the first iteration. Costs for subsequent iterations are based on online measurements of the each task's entire execution time, which includes communication. In both cases, the collection of weighted tasks constitutes a static partitioning problem which must be solved. The goal is to collect bundles of tasks (partitions) and assign them to processors in such a way that computational load imbalance is minimized. In general, solving this problem optimally is NP-hard [7], so there is a trade-off between computing an ideal assignment of task partitions and the overhead required

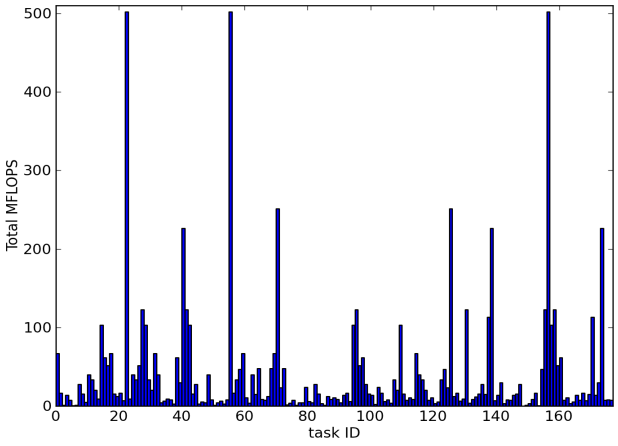

Fig. 3. Total MFLOPS for each task in a single CCSD T_2 tensor contraction for a water monomer simulation. This is a good overall indicator of load imbalance for this particular tensor contraction. Note that each task is independent of the others in this application.

to do so. Therefore, our design defers such decisions to a partitioning library (in our case, Zoltan [12]), which gives us the freedom to experiment with load-balancing parameters (such as the balance tolerance threshold) and their effects on the performance of the CC tensor contraction routines.

Currently we employ static block partitioning, which intelligently assigns "blocks" (or consecutive lists) of tasks to processors based on their associated weights (no geometry or connectivity information is incorporated, as in graph/hypergraph partitioning). However, incorporating task connectivity in terms of data locality has been shown to be a viable means of minimizing data access costs [25]. Our technique focuses on accurately balancing the computational costs of large task groups as opposed to exploiting their connectivity, which also matters a great deal at scale. Fortunately, our approach is easily extendible to include such data-locality optimizations by solving the partition problem in terms of making ideal cuts in a hypergraph representation of the task-data system (see Section V). An application making use of the IE static partitioning technique may partition tasks based on any partitioning algorithm.

D. Dynamic Buckets

When conducting static partitioning, variation in task execution times is undesirable because it leads to load imbalance and starvation of PEs. This effect is particularly noticeable when running short tasks where system noise can potentially counteract execution time estimations and lead to a poor static partitioning assignment. Furthermore, even when performance models are acceptably accurate, they generally require determination of architecture-specific parameters found via off-line measurement and analysis.

A reasonable remedy to these problems is a scheme we call dynamic buckets (Fig. 4), where instead of partitioning the task collection across all PEs, we partition across groups of PEs. Each group will contain an instance of a dynamic `NXTVAL` counter. When groups of PEs execute tasks, the imbalance due to dynamic variation is amortized since unbiased variation will lead to significant cancellation. Also, if the groups are chosen such that each counter is resident on a local compute node relative to the PE group, then `NXTVAL` can work within shared memory, for which performance is considerably better. The other motivation for choosing the execution group to be the node is that contention for the NIC and memory bandwidth in multicore systems is very difficulty to model (i.e. predict) in a complicated application like NWChem, hence we hope to observe a reasonable amount of cancellation of this noise if we group the processes that share the same resources. The idea is that the node-level resources are mostly fixed and that noise will average out since the slowdown in one process due to another's utilization of the NIC will cancel more than the noise between processes on different nodes, since there is no correlation between NIC contention in the latter case. Finally, with a more coarse granularity of task groups, it is feasible that load balance would be acceptable even with a round-robin assignment of tasks to groups (i.e. without performance model based task estimation) because of the adaptability inherent to having several dynamic counters.

When using the dynamic buckets approach, tasks are partitioned by applying the Longest Processing Time algorithm [18], [23], which unlike block partitioning, is provably a 4/3 approximation algorithm (meaning it is guaranteed to produce a solution within ratio 4/3 of a true optimum assignment). First, tasks are sorted by execution time estimation in descending order using a parallel quicksort. Then, each task with the longest execution time estimation is assigned to the least loaded PE until all tasks are assigned. To increase the efficiency of the assignment step, the task groups are arranged in a binary minimum heap data structure where nodes correspond to groups. Tasks can be added to this minimum heap in $O(\log n)$ time (on average) where n is the number of task groups.

The dynamic buckets design in Fig. 4 also captures elements of topology awareness, iterative refinement, and work stealing. The results in Section IV are based on an implementation with node-level topology awareness and a single iteration of refinement based on empirically measured execution times. We refer the reader to other works [13], [3] for information on work stealing implementations.

IV. Experimental Results

This section provides experimental performance results of several experiments on Fusion, an InfiniBand cluster at Argonne National Laboratory. Each node has 36 GB of RAM and two quad-core Intel Xeon Nehalem processors running at 2.53 GHz. Both the processor and network architecture are appropriate for this study because NWChem performs very efficiently on multicore x86 processors and InfiniBand networks. The system is running Linux kernel 2.6.18 (x86_64). NWChem was compiled with GCC 4.4.6, which was previously found to be just as fast as Intel 11.1 because of the heavy reliance on BLAS for floating-point-intensive kernels,

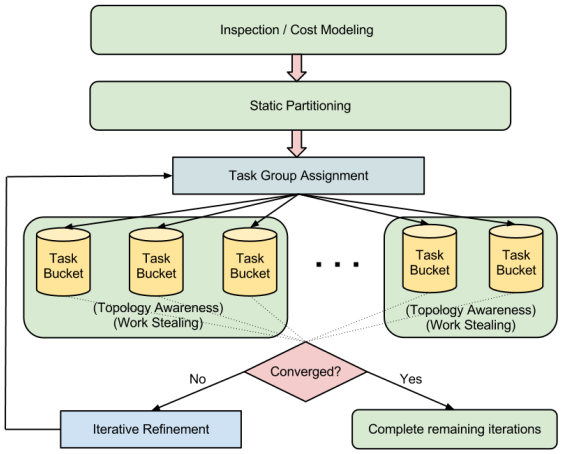

Fig. 4. Inspector/Executor with Dynamic Buckets.

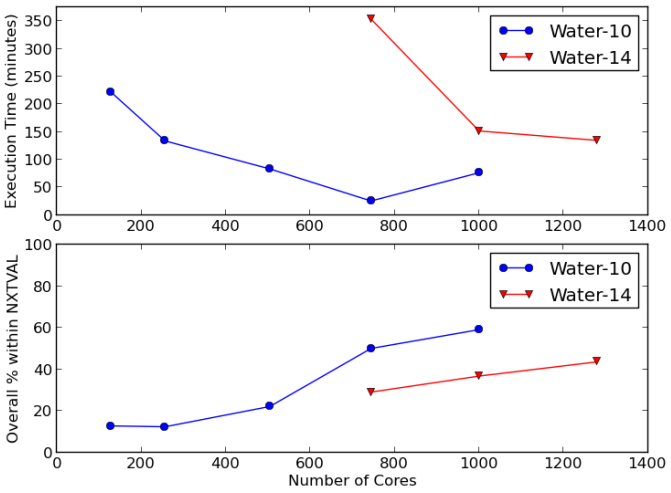

Fig. 5. Total percentage of execution time spent in NXTVAL for a 10-H_2O CCSD simulation (15 iterations) with the aug-cc-pVDZ basis running on the Fusion cluster (without IE). The 14-H_2O test will not fit in global memory on 63 nodes (8 cores per node = 504 cores) or fewer. These data points were extracted from mean inclusive-time profiles as in Fig. 1.

for which we employ GotoBLAS2 1.13. The high-performance interconnect is InfiniBand QDR with a theoretical throughput of 4 GB/s per link and 2 μs latency. The communication libraries used were ARMCI from Global Arrays 5.1, which is heavily optimized for InfiniBand, and MVAPICH2 1.7 (NWChem uses MPI sparingly in the TCE). Fusion is an 8 core-per-node system, but ARMCI requires a dedicated core for optimal performance [19]. We therefore launch all NWChem experiments with 7 MPI processes per node, but reserve all 8 cores using Fusion's job scheduler and resource manager. Because the application is utilizing 8 cores per node, results are reported in multiples of 8 in Figs. 5, 7, and 8.

First we present an analysis of the strong scaling effects of using NXTVAL. Then we describe experiments comparing the original NWChem code with two versions of inspector/executor: one, called I/E Nxtval, that merely eliminates the extraneous calls to NXTVAL, and one that eliminates all calls to NXTVAL in certain methods by using the performance model to estimate costs and Zoltan to assign tasks statically. Because the second technique incorporates both dynamic load balancing and static partitioning, we call it I/E Hybrid. Finally, we show the improvement of the I/E Dynamic Buckets approach for a simulation where I/E Hybrid cannot overcome the effects from variation in task execution time due to system noise.

A. Scalability of centralized load-balancing

The scalability of centralized DLB with NXTVAL in the context of CC tensor contractions in NWChem was evaluated by measuring the percentage of time spent incrementing the counter (averaged over all processes) in two water cluster simulations. The first simulation (blue curve in Fig. 5) is a simulation of 10-water molecules using the aug-cc-pVDZ basis, and the second simulation (red curve) is the same but with 14-water molecules. The percentages are extracted from TAU profiles of the entire simulation run, with the inclusive time spent in NXTVAL divided by the inclusive time spent in the application.

TABLE I
SUMMARY OF THE PERFORMANCE EXPERIMENTS

	N_2	Benzene	10-H_2O	14-H_2O
Simulation type	CCSDT	CCSD	CCSD	CCSD
# of tasks*	261,120	14,280	2,100	4,060
Ave. data size**	7,418	94,674	2.2 mil.	2.7 mil.
Scale limit (cores)	200	320	750	1,200

*from the largest tensor contraction
** in terms of DGEMM input, $mk + kn$

Fig. 5 shows that the percentage of time spent in NXTVAL always increases as more processors are added to the simulation. This increase is partly because of a decrease in computation per processor, but also because of contention for the shared counter, as displayed in Fig. 2. For 10-water molecules, NXTVAL eventually consumes about 60% of the overall application time as we approach 1,000 processes. In the larger 14-water simulation, NXTVAL consumes only about 30% of the time with 1,000 processes, because of the increase in computation per process relative to the 10-water simulation. The 14-water simulation failed on 504 cores (as seen in Fig. 5) because of insufficient global memory.

B. Inspector/Executor DLB

Table I summarizes the NWChem experiments we performed in terms of their task load in the largest tensor contraction of the simulation. CC simulations fall into two broad

TABLE II
300-NODE PERFORMANCE: ORIGINAL CODE FAILS OVER INFINIBAND DUE TO ARMCI_SEND_DATA_TO_CLIENT() ERROR

Processes	2400
Nodes	300
I/E Nxtval	498.3 s
I/E Hybrid	483.6 s
Original	-

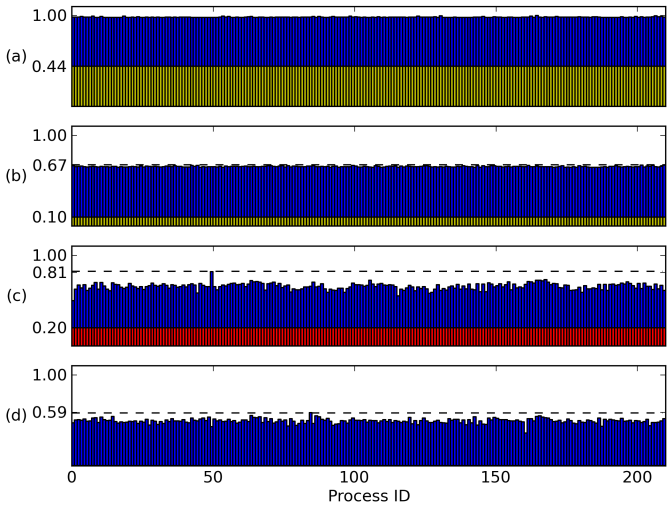

Fig. 6. Comparative load balance of a tensor contraction for benzene CCSD on 210 processes: (a) Original code with total time in NXTVAL overlapped in yellow (all values are normalized to this maximum execution time). (b) I/E with superfluous calls to NXTVAL eliminated. (c) First iteration of I/E with performance modeling and static partitioning (overhead time shown in red). (d) Subsequent iterations of I/E static (with zero overhead and iterative refinement). Despite the increase in load variation in (d), the overall time is reduced noticeably relative to (b).

categories, symmetrically sparse and dense (i.e., a benzene molecule versus an asymmetric water cluster). We found that problems falling in the sparse category are suitable for the I/E Nxtval method because they have a large number of extraneous tasks to be eliminated. While the water cluster systems can potentially eliminate a similar percentage of tasks, their relatively larger average task size results in DGEMM dominating the computation. The differences in task loads between these problems necessitate different I/E methods for optimal performance, as shown below in Figs. 7 and 8.

Applying the I/E Nxtval model to a benzene monomer with the aug-cc-pVTZ basis in the CCSD module results in as much as 33% faster execution of code compared with the original (Fig. 7). The I/E Nxtval version consistently performs about 25-30% faster for benzene CCSD. At high numbers of processes, the original code occasionally fails on the Fusion InfiniBand cluster with an error in `armci_send_data_to_client()`, whereas the I/E Nxtval version continues to scale to beyond 400 processes. This suggests that the error is triggered by an extremely busy NXTVAL server.

C. Static Partition

The I/E Hybrid version applies complete static partitioning using the performance model cost estimation technique to long-running tensor contractions which are experimentally observed to outperform the I/E Nxtval version. Fig. 7 shows that this method always executes in less time than both the original code and the simpler I/E Nxtval version. Though it is not explicitly proven by any of the figures, this version of the code also appears to be capable of executing at any number of processes on the Fusion cluster, whereas the I/E Nxtval and original code eventually trigger the ARMCI error mentioned in the previous section.

Unfortunately, it is a difficult feat to transform the machine-generated tensor contraction methods from within the TCE generator, so we have taken a top-down approach where the generated source is changed manually. Because there are over 70 individual tensor contraction routines in the CCSDT module and only 30 in the CCSD module, we currently have I/E Hybrid code implemented only for CCSD.

D. Dynamic Buckets

I/E Dynamic Buckets (I/E-DB) is usually the method with the best performance, as seen in Fig. 8. This plot shows the two most time consuming tensor contractions in a 10-H_2O system. In this problem, I/E Nxtval performs no better than the original code because of relatively less sparsity and larger task sizes in the overall computation. I/E Hybrid (not shown) performs slightly worse than the original code. As explained in section III-D, this is due to error in the task execution time estimations. The I/E-DB technique shows up to 16% improvement due to better load balance when dynamic counters manage groups of tasks.

V. Related Work

Alexeev and coworkers have applied novel static load balancing techniques to the fragment molecular orbital (FMO) method [2]. FMO differs in computational structure from iterative CC, but the challenge of load balancing is similar, and their techniques parallel the IE cost estimation model. The FMO system is first split into fragments that are assigned to groups of CPU cores. The size of those groups is chosen based on the solution of an optimization problem, with three major terms representing time that is linearly scalable, nonlinearly scalable, and nonparallel.

Hypergraph partitioning was used by Krishnamoorthy and coworkers to schedule tasks originating from tensor contractions [25]. Their techniques optimize static partitioning based on common data elements between tasks. Such relationships are represented as a hypergraph, where nodes correspond to tasks, and hyperedges (or sets of nodes) correspond to common data blocks the tasks share. The goal is to optimize a partitioning of the graph based on node and edge weights. Their hypergraph cut optimizes load balance based on data element size and total number of operations, but such research lacks a thorough model for representing task weights, which the IE cost estimation model accomplishes.

The Cyclops Tensor Framework [39], [40] implements CC using arbitrary-order tensor contractions which are implemented by using a different approach from NWChem. Tensor contractions are split into redistribution and contraction phases, where the former permutes the dimensions such that the latter can be done by using a matrix-matrix multiplication algorithm such as SUMMA [46]. Because CTF uses a cyclic data decomposition, load imbalance is eliminated, at least for dense contractions. Point-group symmetry is not yet implemented in CTF and would create some of the same type

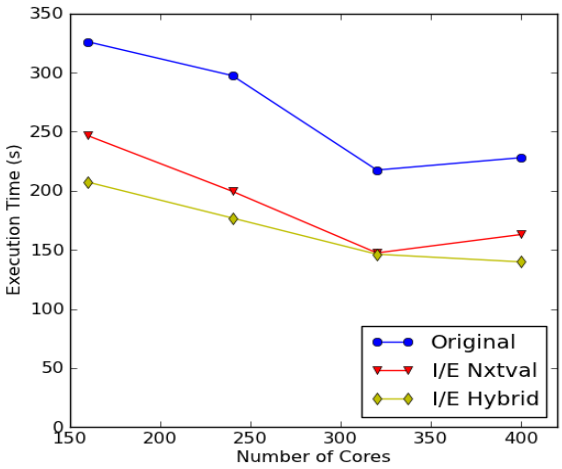

Fig. 7. Benzene aug-cc-pVQZ I/E comparison for a CCSD simulation.

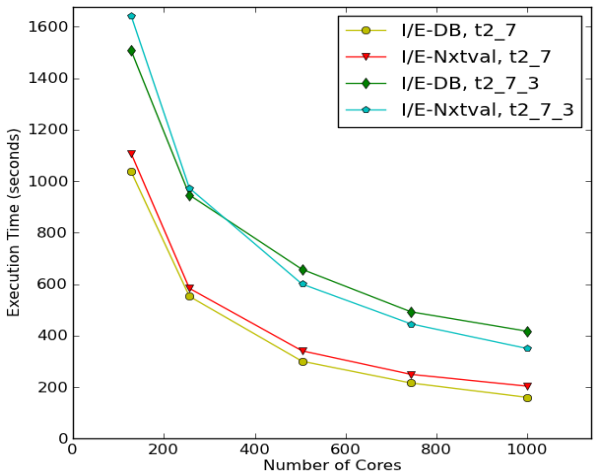

Fig. 8. Comparison of I/E Nxtval with I/E Dynamic Buckets for the two most time consuming tensor contractions during a 10-H_2O simulation, `t2_7` and `t2_7_3`.

of load imbalance as seen in this paper, albeit at the level of large distributed contractions rather than tiles. We hypothesize that static partitioning would be effective at mitigating load-imbalance in CTF resulting from point-group symmetry.

VI. Conclusions and Future Work

We have presented an alternate approach for conducting load balancing in the NWChem CC code generated by the TCE. In this application, good load balance was initially achieved by using a global counter to assign tasks dynamically, but application profiling reveals that this method has high overhead which increases as we scale to larger numbers of processes. Splitting each tensor contraction routine into an inspector and an executor component allows us to evaluate the system's sparsity and gather relevant cost information regarding tasks, which can then be used for static partitioning. We have shown that the inspector-executor algorithm obviates the need for a dynamic global counter when applying performance model prediction, and can improve the performance of the entire NWChem coupled cluster application. In some cases the overhead from a global counter is so high that the inspector-executor algorithm enables the application to scale to a number of processes that previously was impossible because of the instability of the `NXTVAL` server when bombarded with tasks.

The technique of generating performance models for `DGEMM` and `SORT4` to estimate costs associated with load balancing is general to all compute-kernels and can be applied to applications that require large-scale parallel task assignment. While other noncentralized DLB methods (such as work stealing and resource sharing) could potentially outperform such static partitioning, such methods tend to be difficult to implement and may have centralized components. The approach of using a performance model and a partitioning library together to achieve load balance is easily parallelizable (though in NWChem tensor contractions, we have found a sequential version to be faster because of the inexpensive computations in the inspector) and easy to implement and requires few changes to the original application code.

Because the technique is readily extendible, we plan to improve our optimizations by adding functionality to the inspector. For example, we can exploit proven data locality techniques by representing the relationship of tasks and data elements with a hypergraph and decomposing the graph into optimal cuts [25].

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility and Laboratory Computing Resource Center at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

The research at the University of Oregon was supported by grants from the U.S. Department of Energy, Office of Science, under contracts DE-FG02-07ER25826, DE-SC0001777, and DE-FG02-09ER25873.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6:747–754, 1995.

[2] Yuri Alexeev, Ashutosh Mahajan, Sven Leyffer, Graham Fletcher, and Dmitri Fedorov. Heuristic static load-balancing algorithm applied to the fragment molecular orbital method. *Supercomputing*, 2012.

[3] Humayun Arafat, P. Sadayappan, James Dinan, Sriram Krishnamoorthy, and Theresa L. Windus. Load balancing of dynamical nucleation theory Monte Carlo simulations through resource sharing barriers. In *IPDPS*, pages 285–295, 2012.

[4] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[5] Rodney J. Bartlett. Coupled-cluster approach to molecular structure and spectra: a step toward predictive quantum chemistry. *Journal of Physical Chemistry*, 93(5):1697–1708, 1989.

[6] Rodney J. Bartlett and Monika Musiał. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291–352, 2007.

[7] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, 30(3):207–214, March 1981.

[8] Yannick J. Bomble, John F. Stanton, Mihály Kállay, and Jürgen Gauss. Coupled-cluster methods including noniterative corrections for quadruple excitations. *Journal of Chemical Physics*, 123(5):054101, 2005.

[9] E. J. Bylaska et. al. NWChem, a computational chemistry package for parallel computers, version 6.1.1, 2012.

[10] F.A. Cotton. *Chemical Applications of Group Theory*. John Wiley & Sons, 2008.

[11] T. Daniel Crawford and Henry F. Schaefer III. An introduction to coupled cluster theory for computational chemists. In K. B. Lipkowitz and D. B. Boyd, editors, *Reviews in Computational Chemistry*, volume 14, chapter 2, pages 33–136. VCH Publishers, New York, 2000.

[12] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[13] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, 2009. ACM.

[14] Stephen Fink and Scott Baden. Runtime support for multi-tier programming of block-structured applications on SMP clusters. In Yutaka Ishikawa, Rodney Oldehoeft, John Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin / Heidelberg, 1997.

[15] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1–2):61–82, 1998.

[16] J. Fuchs and C. Schweigert. *Symmetries, Lie Algebras and Representations: A Graduate Course for Physicists*. Cambridge University Press, 2003.

[17] Jürgen Gauss, John F. Stanton, and Rodney J. Bartlett. Coupled-cluster open-shell analytic gradients: Implementation of the direct product decomposition approach in energy gradient calculations. *Journal of Chemical Physics*, 95(4):2623–2638, 1991.

[18] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[19] Jeff R. Hammond, Sriram Krishnamoorthy, Sameer Shende, Nichols A. Romero, and Allen D. Malony. Performance characterization of global address space applications: a case study with NWChem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012.

[20] Robert J. Harrison. Portable tools and applications for parallel computers. *International Journal of Quantum Chemistry*, 40(6):847–863, 1991.

[21] So Hirata. Tensor Contraction Engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A*, 107:9887–9897, 2003.

[22] Mihály Kállay and Jürgen Gauss. Approximate treatment of higher excitations in coupled-cluster theory. *Journal of Chemical Physics*, 123(21):214105, 2005.

[23] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, Massachusetts, USA, 2005.

[24] Karol Kowalski, Jeff R. Hammond, Wibe A. de Jong, Peng-Dong Fan, Marat Valiev, Dunyou Wang, and Niranjan Govind. Coupled cluster calculations for large molecular and extended systems. In Jeffrey R. Reimers, editor, *Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology*. Wiley, 2011.

[25] Sriram Krishnamoorthy, Ümit V. Çatalyürek Umit Catalyurek, Jarek Nieplocha, and Atanas Rountev. Hypergraph partitioning for automatic memory hierarchy management. In *Supercomputing (SC06)*, 2006.

[26] Stanislaw A. Kucharski and Rodney J. Bartlett. Coupled-cluster methods that include connected quadruple excitations, T_4: CCSDTQ-1 and Q(CCSDT). *Chemical Physics Letters*, 158(6):550–555, 1989.

[27] Stanislaw A. Kucharski and Rodney J. Bartlett. Recursive intermediate factorization and complete computational linearization of the coupled-cluster single, double, triple, and quadruple excitation equations. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 80:387–405, 1991.

[28] Stanislaw A. Kucharski and Rodney J. Bartlett. The coupled-cluster single, double, triple, and quadruple excitation method. *Journal of Chemical Physics*, 97(6):4282–4288, 1992.

[29] Stanislaw A. Kucharski and Rodney J. Bartlett. An efficient way to include connected quadruple contributions into the coupled cluster method. *Journal of Chemical Physics*, 108(22):9221–9226, 1998.

[30] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349, New York, 1994. ACM.

[31] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:10–197, 1996.

[32] Jozef Noga and Rodney J. Bartlett. The full CCSDT model for molecular electronic structure. *Journal of Chemical Physics*, 86(12):7041–7050, 1987.

[33] Nevin Oliphant and Ludwik Adamowicz. Coupled-cluster method truncated at quadruples. *The Journal of Chemical Physics*, 95(9):6645–6651, 1991.

[34] Elmar Peise and Paolo Bientinesi. Performance modeling for dense linear algebra. In *Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS12)*, November 2012.

[35] George D. Purvis III and Rodney J. Bartlett. A full coupled-cluster singles and doubles model: the inclusion of disconnected triples. *Journal of Chemical Physics*, 76(4):1910–1918, 1982.

[36] Krishnan Raghavachari, John A. Pople, Eric S. Replogle, and Martin Head-Gordon. Fifth-order Møller-Plesset perturbation theory: comparison of existing correlation methods and implementation of new methods correct to fifth order. *Journal of Physical Chemistry*, 94:5579–5586, 1990.

[37] Krishnan Raghavachari, Gary W. Trucks, John A. Pople, and Martin Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157:479–483, May 1989.

[38] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[39] Edgar Solomonik. Cyclops Tensor Framework. http://www.eecs.berkeley.edu/ solomon/cyclopstf/index.html.

[40] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, may 2013.

[41] John Stanton. This remark is attributed to Devin Matthews.

[42] John F. Stanton. Why CCSD(T) works: a different perspective. *Chemical Physics Letters*, 281:130–134, 1997.

[43] John F. Stanton, Jürgen Gauss, John D. Watts, and Rodney J. Bartlett. A direct product decomposition approach for symmetry exploitation in many-body methods, I: Energy calculations. *Journal of Chemical Physics*, 94(6):4334–4345, 1991.

[44] Jiří Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in Ursell-Type expansion using quantum-field theoretical methods. *Journal of Chemical Physics*, 45(11):4256–4266, December 1966.

[45] Miroslav Urban, Jozef Noga, Samuel J. Cole, and Rodney J. Bartlett. Towards a full CCSDT model for electron correlation. *Journal of Chemical Physics*, 83(8):4041–4046, 1985.

[46] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[47] John D. Watts and Rodney J. Bartlett. The coupled-cluster single, double, and triple excitation model for open-shell single reference functions. *Journal of Chemical Physics*, 93(8):6104–6105, 1990.