Directed Research Project Report: Performance Optimizations of the Tensor Contraction Engine in NWChem

David Ozog Allen Malony
Deptartment of Computer and Information Science
University of Oregon
Eugene, Oregon 97403
Email: {ozog, malony}@uoregon.edu

Jeff R. Hammond Pavan Balaji
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, Illinois 60439
Email: balaji@anl.gov

***Abstract*—Two large-scale performance optimizations of the NWChem computational chemistry framework are described. The first optimization involves hybrid static/dynamic load balancing techniques of distributed tensor contraction operations in the coupled cluster method. Using performance models of the most expensive computational kernels enables the use of static partitioning techniques for dividing work across compute cores. Because fully static partitioning is incapable of dealing with dynamic variation of task costs, we also consider hybrid schemes that utilize dynamic scheduling within subgroups. These schemes are compared to the original centralized dynamic load-balancing algorithm of NWChem as well as an improved centralized scheme. We demonstrate that execution time can be reduced by as much as 50% at scale. The technique is applicable to any scientific application requiring load balance where performance models or estimation of kernel execution times are available.**

The second optimization involves a new execution model that automatically supports the overlap of communication and computation in processing a pool of remote work items (as in NWChem). Typically, NWChem is executed with a one-process-per-core mapping in which each process in the system iterates through a get/compute/put work-processing cycle. We propose the "WorkQ" execution model, in which some number of on-node "producer" processes primarily do communication and the other "consumer" processes do computation, yet processes can switch roles dynamically for the sake of performance. This system is facilitated by a highly tunable node-wise FIFO message queue protocol. Our WorkQ library implementation enables an MPI+X hybrid programming model where the X is comprised of SysV message queues and the user's choice of SysV, POSIX, and MPI shared memory. We develop a simplified software mini-application which mimics the performance behavior of NWChem at arbitrary scale, and show that the WorkQ engine outperforms the original model by over a factor of 2. We also show performance improvement in the coupled cluster module of NWChem across all possible tile sizes.

I. Introduction

In this paper we consider two important parallel optimizations in the context of the tensor contraction engine in NWChem for solving coupled cluster systems of equations. The first optimization involves achieving efficient system load balance, and the second involves overlapping communication with computation. While the two optimizations are considered separately below, we will eventually see that they influence each other.

A. Load Balance

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` [24]), 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) to either single and double (CCSD) or triple (CCSDT) order accuracy. Popular among chemists are perturbative 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

[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.

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.

B. Overlap of Communication and Computation

In order to most effectively utilize hardware resources within large-scale parallel applications, processor clock cycles must be fully dedicated towards performing computation whenever work is readily available. However, we must also be careful to assure that future work will be available when finished processing current work-items. An item of work in this context is comprised of a computational task, such as a matrix multiplication routine, along with some associated data, such as two arrays. In many circumstances of parallel processing, a process which is assigned a work-item must wait for data to be migrated from the memory space of another process before computation can take place. Not only does incoming data often cross the entire memory hierarchy of a compute node, it may also cross a series of network hops from a remote location. The variety and non-uniformity of both compute node architectures and network topologies in modern supercomputers complicates the wait patterns of processing work items in parallel. This in turn complicates the development of distributed computational algorithms which effectively overlap communication and computation while efficiently utilizing system resources.

One long-standing and effective strategy for optimizing parallel applications is to overlap time spent waiting on network communication with time spent doing computation [6], [48]. For example, non-blocking communication routines enable asynchronous progress to occur within a process or thread of execution. Care must be taken to minimize overheads associated with such overlapping, because polling for state and data migration between process spaces can be expensive. These concepts are particularly important in a supercomputing ecosystem in which distributed commodity hardware dominates, and data affinity, global address space programming models, and non-uniform memory access must play a role in designing applications in which a high amount of overlap is achieved. However, overlapping communication and computation has limited utility because at best, it alone can improve the performance of a given application only by a factor of two [46].

The current state of distributed memory parallel computing involves difficulties associated with the diversification of architectures and the pervasiveness of heterogeneous systems. While modern architectures are becoming increasingly heterogeneous, most parallel runtime systems are based on the single-program, multiple data programming schemes which perform best when hardware is homogeneous. There are many obstacles to parallelism that are complicated by heterogeneity: contention on shared resources, load imbalance, data management, and miscellaneous sources of overhead. These diverse complications suggest that autonomous optimization of parallel applications requires an underlying adaptable runtime processing engine that is highly configurable and able to quickly respond to variability in system behavior and changes in architecture. The results of this paper suggest that when a runtime accomplishes effective overlap on top of efficient hardware utilization, speedups can be more than a factor of two.

In addition to inspector-executor load balancing, this paper also considers a new execution model, we call WorkQ, that prioritizes the overlap of communication and computation while simultaneously providing a set of runtime parameters for architecture-specific tuning. Using an implementation of this model, we perform various experiments on a benchmark that mimics the bottleneck computation within an important quantum many-body simulation application (NWChem) and show good performance improvement with our techniques. Section II provides the necessary background regarding the PGAS model and the NWChem application. Section III discusses motivation for the construction of our load balancing and execution models. Section IV outlines the design and implementation of the software, Section V describes a set of experimental evaluations, Section VI discusses related and future work, and Section VII presents our concluding remarks.

II. Background

In this section, we provide the necessary background information for understanding the context of the Inspector-Executor and WorkQ models and their applications throughout the paper. Topics include *A)* the PGAS paradigm, *B)* the Global Arrays programming model, *C)* NWChem and the coupled cluster technique, and *D)* the Tensor Contraction Engine.

A. PGAS

The availability and low cost of commodity hardware components has shaped the evolution of supercomputer design towards distributed memory architectures. While distributed commodity-based systems have been a boon for effectively and inexpensively scaling computational applications, they have also made it more difficult for programmers to write efficient parallel programs. This difficulty takes many forms: diversity of architectures, managing load balance, writing scalable parallel algorithms, exploiting data locality of reference, and utilizing asynchronous control to name a few. A popular parallel programming model which eases the burden on distributed memory programmers is found in partitioned global address space (PGAS) languages and interfaces.

In the PGAS paradigm, programs are written SPMD-style to compute on an abstract global address space. Abstractions are presented such that global data can be manipulated as though it were located in shared memory, when in fact data is logically partitioned across distributed compute nodes with an arbitrary network topology. This arrangement enables productive development of distributed memory programs that are inherently conducive to exploiting data affinity across threads or processes. Furthermore, when presented with an API that exposes scalable methods for working with global address space data, computational scientists are empowered to program vast cluster resources without having to worry about optimization, bookkeeping, and portability of relatively simple distributed operations.

Popular PGAS languages/interfaces include UPC, Titanium, Coarray Fortran, Fortress, X10, Chapel, and Global Arrays. The implementation in this work was built on top of Global Arrays/ARMCI, which is the subject of the next section.

B. Global Arrays

Global Arrays (GA) is a toolkit for doing PGAS computations in HPC codes using C/C++, Fortran, or Python [37]. It is built on top of the aggregate remote memory copy interface (ARMCI), which provides efficient one-sided communication primitives optimized for most remote direct memory access (RDMA) hardware [35]. To understand the utility of GA, consider the transpose operation of a matrix in global memory. Mathematically, this is a very simple operation, but it can involve intensive bookkeeping to program a global transpose in distributed memory. This is a task many computational scientists would rather avoid. GA provides the means for accomplishing transposition of a global matrix with one call that is portable and optimized to efficiently utilize one-sided RDMA operations with ARMCI. Besides the standard put/get/accumulate functionality common in one-sided communication libraries, there are a number of other helpful computational operations provided by the GA API. For example there are functions for matrix addition/multiplication/diagonalization/inversion, ghost cell control, strided gets and puts, solving linear systems of equations, and more.

The canonical dynamic load balancer `NXTVAL` for Global Arrays was inherited from TCGMSG [24], 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 [36]. 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.

A common misconception is GA's relationship with Message Passing Interface (MPI). Although a large portion of GA's communication is done strictly through ARMCI, GA still requires linking with a message-passing library. This library need not be MPI (another alternative is TCGMSG [23]), but there does need to be a message-passing library underneath the GA stack that provides SPMD capability, process IDs, synchronization, broadcast and reduction operations, etc. As of this writing, MPI is the de facto standard library for satisfying these requirements. In addition, it is now possible to replace the entire ARMCI communication layer with equivalent MPI 3.0 RMA routines for doing one-sided communication [15]. This is typically done on newer systems and interconnects to take advantage of MPI's portability.

C. NWChem and Coupled Cluster

NWChem [10] 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 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.

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
```

The Coupled Cluster (CC) component of NWChem is a important molecular electronic structure module highly utilized by the quantum chemistry and physics communities [7]. CC is a numerical technique for solving the electronic Schrödinger equation using an exponential ansatz operator sum acting upon a one-electron reference wave function [29]:

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

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. [13], [5] for more information.

The sum of operators is truncatable to arbitrary-order accuracy, analogous to a Taylor series expansion. This leads to a hierarchy of CC methods that provides increasing accuracy at increased computational cost [4]:

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

In CC, each operator is evaluated via a series of tensor contractions (as described in the next section). When truncating CC to include only the "doubles-order" term, the method is referred to as CCD. When including both singles and doubles, the method is CCSD. With triples and quadruples, the methods are CCSDT and CCSDTQ, respectively. There also exist important perturbative methods (such as CCSD(T) and CCSD(Q)) that can approximate the addition of a higher order term without requiring the full increase in computational and memory requirements.

The simplest CC method that is generally useful is CCSD [45], has a computational cost of $O(n^6)$ and storage cost of $O(n^4)$, where n is the sum of occupied and virtual electron orbitals. The "gold standard" CCSD(T) method [57], [47], [53] 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 [38], [59] 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 [32], [33], [39] requires $O(n^{10})$ computation and $O(n^8)$ storage, while the perturbative approximation to quadruples, CCSDT(Q) [31], [34], [9], [27], 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) [52].

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 [18] and point-group symmetry [12]. 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. [54], [19]. 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.

Algorithm 2 Pseudocode for the default TCE implementation of a tensor contraction. For clarity, some 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) is a domain-specific language for automatically generating high-performance programs which compute the working equations of second quantized many-electron theories, such as coupled cluster [25]. The primary motivation for supporting such a tool is that symbolic manipulation of these equations is an extremely time consuming and error-prone process when done by hand, but the TCE facilitates the generation of portable and efficient parallel code that is verified for correctness. TCE is a core component of *ab initio* chemistry capabilities in NWChem, as well as in UTChem, developed at the University of Tokyo [60].

The TCE generates GA programs written in Fortran that exploit spin, spatial, and index permutation symmetries among the working set of equations to reduce the computational and memory requirements of these methods. Despite these efforts, the computations of CC methods have polynomial algorithmic complexity in terms of the number of FLOPS and memory usage. As mentioned above, CCSD equations have algorithmic complexity of $O(n^6)$ for operations and $O(n^4)$ for memory. 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 ($Otile$ and $Vtile$ in Algs. 2, 3, and 5).

The overall TCE computation consists of several Jacobi iterations through a directed acyclic graph where each node refers to a calculation of a tensor contraction intermediate (corresponding to the truncatable sum of operators described in the previous section). Before computation begins, the GA data is arranged into tiles that each contain orbitals with the same spin and spatial symmetries. The granularity of these tiles is of crucial significance for performance, as it determines the number of total work-items. It is important that tile-size be small enough for there to be more tasks than the number of processes in the application. At the same time, it is important for tile-sizes to be sufficiently large because an excessive number of work-items leads to unnecessary accumulation of overhead on the dynamic load balancer (see section III).

The TCE reduces the contraction of two high-dimensional tensors into a summation of the product of several two dimensional arrays. Therefore, the performance of the underlying BLAS library is quite influential to the overall performance of TCE. For the purposes of this paper, each tensor contraction routine can be thought of as a global task pool of tile-level 2-dimensional DGEMM (double-precision general matrix-matrix multiplication) operations. As alluded above, this pool of work items is processed according to the following execution model:

1) A unique work-item ID is dynamically assigned via an atomic read-modify-write operation to a dynamic load balancing counter (`NXTVAL`).
2) The global addresses of two tiles (A and B) in the global array space is determined (TCE hash lookup).
3) The corresponding data is copied to the local process space (via one-sided RMA) with `GA_Get()` calls.
4) A contraction is formed between the local copies of tiles A and B and stored into C. When necessary, a permute-DGEMM-permute pattern is performed to arrange the indices of the tensor tiles to align with the format of matrix-matrix multiplication.
5) Steps 2, 3, and 4 repeat over the work-item tile bundle, then C is accumulated (`GA_acc()` call) into a separate global array at the appropriate location.

While this particular algorithm is specific to CC, it is important to note that it falls under a more general `get/compute/put` model which is common to many computational applications.

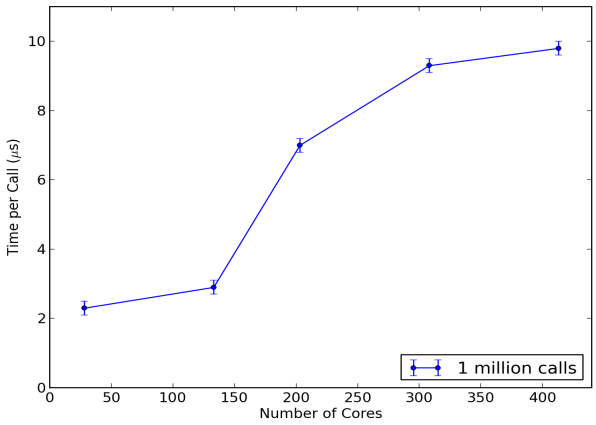

Fig. 1. Flood benchmark showing the execution time per call to `NXTVAL` for 1 million simultaneous calls. The process hosting the counter is being flooded with messages, so when the arrival rate exceeds the processing rate, buffer space runs out and the process hosting the counter must utilize flow control.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

For example, the problem of numerically solving PDEs on domains distributed across memory spaces certainly falls under this category.

The next section discusses motivation for the development of an alternative load balancing technique and a new execution model that are able to perform this same computation in a more efficient manner.

III. Motivation

In this section we present performance measurements that support our motivation for developing inspector-executor (IE) load-balancing algorithms and a new runtime execution model for processing tasks in applications such as the TCE-CC of NWChem. For load balance, we consider the proportion of time spent in `NXTVAL` for large CC simulations, then show that `NXTVAL` scalability suffers in terms of average time per call when strong scaling a flood micro-benchmark. For overlap of communication and computation, we consider measurements from a simple trace of a tensor contraction kernel, then discuss the implications of a new execution model.

A. Load Balance

When conduction 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

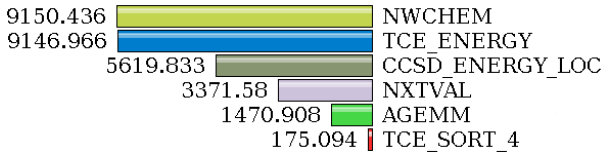

Fig. 2. Average inclusive-time (in seconds) profile of a 14-water monomer CCSD simulation with the aug-cc-PVDZ basis for 861 MPI processes across 123 Fusion nodes connected by InfiniBand. The `NXTVAL` routine consumes 37% of the entire computation. This profile was made using TAU [49]; for clarity, some subroutines were removed.

(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. 1) 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. 2 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.

The average time per call to `NXTVAL` increases with the number of participating PEs, but any simple IE algorithm will improve strong scaling only as much as the proportion of tasks eliminated by spatial and point-group symmetry arguments. Dynamic load balancing 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.

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 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

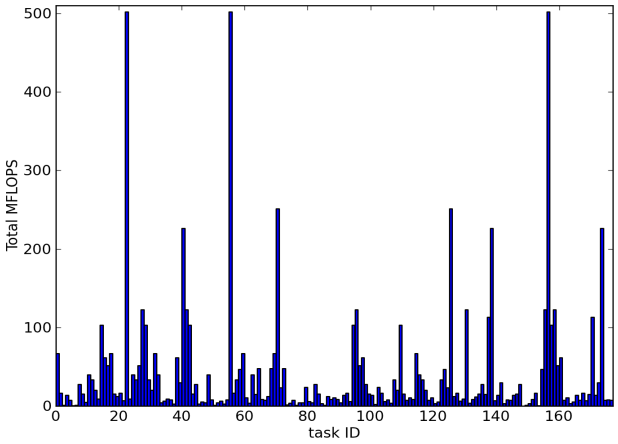

Fig. 3. Total MFLOPS for each task in a single CCSD T_2 tensor contraction for a water monomer simulation. This is a good overall indicator of load imbalance for this particular tensor contraction. Note that each task is independent of the others in this application.

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. For each call to `DGEMM` or `SORT4`, the model estimates the time to execute, and accrues it into $cost_w$. 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 [43]. 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.

B. Communication / Computation Overlap

In order to better understand the performance behavior of the TCE task execution model described at the end of section II-D, we develop a mini-application that executes the same processing engine without the namespace complications introduced by quantum many-body methods. The details of this mini-app will be discussed in sections IV-E and V-B1, but here we present a simple trace of the execution to better motivate and influence the design of our runtime in Section IV.

The top half of Fig. 4 shows an excerpt of a trace collected with the TAU parallel performance system [49] for 12 MPI processes on 1 node within a 16 node application executed on the ACISS cluster (described in Section V). This trace is visualized in Jumpshot with time on the horizontal axis, each row corresponding to an MPI process, and each color to a particular function call in the application. In particular, the purple bars correspond to step 1 in the TCE execution model described in the previous section. The green bars correspond to the one-sided get operation on the two tiles A and B from step 3 (step 2 is implicit in the mini-app and is thus not contained in a function). The yellow bars are non-communication work cycles and the pink bars are DGEMM (these are small due to the relatively small tile-size in this experiment). Both yellow and pink together correspond to step 4. Step 5 is not shown in this timeline, but occurs at a future point at the end of this task bundle.

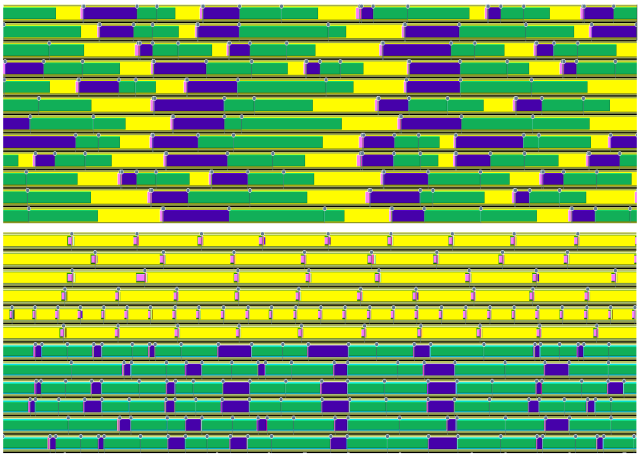

Fig. 4. TAU trace of original code compared to WorkQ. Yellow is computation, purple is `ARMCI_Rmw()` (corresponding to `Nxtval()` call), green is `ARMCI_NbGetS()` (corresponding to `GA_Get()` call), and pink is `DGEMM`.

The bottom half of Fig. 4 shows the equivalent trace with our alternative parallel execution model in which 6 MPI processes dedicate their cycles towards doing more computation, and the other 6 processes dedicate their cycles towards communication. We measure a performance improvement in terms of execution time by a factor of 1.8 in this experiment. The advantage can be inferred from the trace: in the original execution, there are moments when hardware resources are fully saturated with computation (i.e., all rows are yellow at a particular time) yet other moments where starvation is occurring (i.e., rows are green at a particular time). Besides drastically reducing moments of work-starvation, the alternative model enables tunability: for instance, we can empirically determine the optimal number of computation versus communication processes (discussed further in section V-C).

The TCE engine uses blocking `GA_get()` and `GA_put()` calls to gather and place tiles, respectively, into the global space. It is a reasonable proposition to use the corresponding non-blocking API for GA (`GA_nbget` and `GA_nbput`) to process tiles with overlap. In particular, the processing engine can be bootstrapped with two sets of tiles, such that iteration #1 processes the first set while leaving the second set for iteration #2 and so on. In this way tasks can be overlapped inductively by iteration $i+1$ working on data that was initiated with a non-blocking call in iteration i. However, we have verified experimentally that the performance benefit in this system is not as profitable as having a queue of work items. This is due to the variability of the time to execute tasks compared to the time to get/put work-items. Variation in execution time occurs either because of system noise or simply inherent differences in task sizes. For instance, if it takes relatively more time to get the data in iteration i than to compute in iteration $i+1$, then unnecessary time is spent waiting in iteration $i+1$. This can be mitigated by keeping a queue of tasks on the compute node that is populated with several work-items. The design and implementation of this queuing system is the subject of the next section.

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 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
```

IV. Design and Implementation

In this section we discuss our design and implementation of the inspector-executor (IE) algorithm and our new task-processing system. We begin by describing a simple version of the IE which eliminates tasks based on spatial/spin symmetry, then augment this 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.

The overlap of communication and computation in a dynamic and responsive manner is accomplished with a library for managing compute node task queuing and processing within SPMD applications. We have implemented this library, and call it WorkQ. This section presents the software architectural design for WorkQ, describes some implementation details and possible extensions, then presents a portion of the API and how it can be deployed for efficient task processing in distributed memory SPMD programs.

A. Eliminating Tasks by Spatial/Spin Symmetry Arguments

As mentioned in section II-C, the TCE is able to reduce computational cost and storage requirements by exploiting spin and point-group symmetries. In essence, when orbitals are known to have no interactions with each other, then corresponding blocks of relevant tensors are zero. Therefore, these block-contractions need not be computed, and the TCE accordingly skips such tasks. However, during our initial work in reducing overhead to the `NXTVAL` counter, it was discovered that tasks eliminated by symmetry arguments where still balanced by the central counter. Using a simple inspection loop, a task list is created that does not include the "null-tasks" and is then fed into the executor. This simple version of the IE is shown in Algs. 3 and 4. It is important to note that the null-tasks can be eliminated by moving the symmetry conditionals

Algorithm 5 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)
```

before the call to NXTVAL, but this simple IE is a stepping stone towards a more sophisticated algorithm discusses in the next section which is able to eliminate *all* calls to NXTVAL.

B. Static Partitioning

In our IE implementation, the inspector applies 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 [8], so there is a trade-off between computing an ideal assignment of task partitions and the overhead required to do so. Therefore, our design defers such decisions to a partitioning library (in our case, Zoltan [14]), 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 [30]. 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 VI). An application making use of the IE static partitioning technique may partition tasks based on any partitioning algorithm.

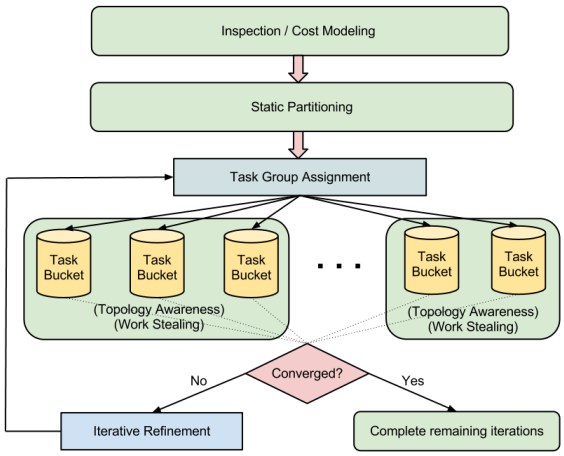

Fig. 5. Inspector-Executor with Dynamic Buckets.

C. 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. 5), 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

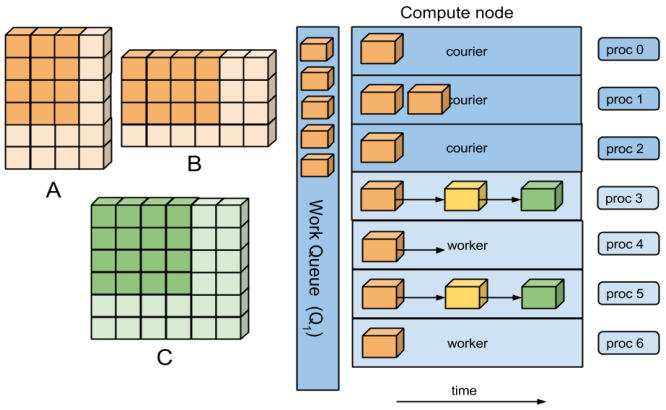

Fig. 6. A simplified view of state for the WorkQ model at a particular moment in time on a single compute node. In this case, there are 3 courier processes, and 4 worker processes. The work queue contains tile metadata from A and B yet to be processed by the workers. Workers place resultant (green) tiles into a separate queue for put/accumulate (not shown).

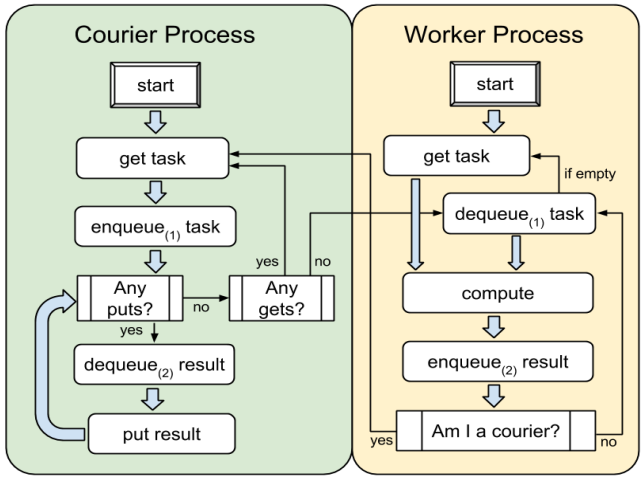

Fig. 7. Flow diagram for the dynamic producer/consumer version of the WorkQ execution model. The left green column represents the activities of courier processes and the right yellow column represents activities of worker processes. In this system, couriers can temporarily become workers and vice versa.

partitioned by applying the Longest Processing Time algorithm [20], 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. 5 also captures elements of topology awareness, iterative refinement, and work stealing. The results in Section V 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 [16], [3] for information on work stealing implementations.

D. WorkQ Library

As described in section III, there is potential for the TCE-CC task processing engine to experience unnecessary wait times and relatively inefficient utilization of local hardware resources. Here we describe an alternative runtime execution model with the goals of: 1) processing tasks with less wait time and core starvation, 2) exposing tunability to better utilize hardware resources, and 3) responding dynamically to real-time processing variation across the cluster.

Here we simplify the operations of the TCE described in section II-D into a pedagogical model that is akin to tiled matrix multiplication of two arrays, A and B . In this model, A and B contain GA data which is distributed across a cluster of compute nodes. The overall goal of the application is to multiply corresponding tiles of A and B, then to accumulate the results into the appropriate location of a separate global array, C. In order to accomplish this within the original execution engine, individual processes each take on the execution loop from section II-D: `get(A); get(B); compute(A,B); put(C)`. The behavior of a single compute node involved in this computation is characterized by the trace in the top half of Fig. 4: at any given moment in time, processes work within a particular stage of the execution loop.

Fig. 6 shows a graphical representation of the WorkQ system on a compute node at a particular instant in time. In the WorkQ runtime, each compute node contains a FIFO message queue, $\mathbf{Q}_1$, in which some number of *courier processes* are responsible for placing A and B tile metadata onto $\mathbf{Q}_1$, then storing the incoming tiles into node-local shared memory. Meanwhile, the remaining *worker processes* dequeue task metadata bundles as they become available, then use this information to follow a pointer to the data in shared memory and perform the necessary processing. Once a worker process is finished computing its task result, it places the resultant data into a separate FIFO message queue, $\mathbf{Q}_2$, which contains data for some courier process to eventually accumulate into C.

We now describe the 4 primary components of the WorkQ implementation: 1) the dynamic producer/consumer system, 2) the on-node message queues 3) the on-node shared memory, and 4) the WorkQ library API.

1) Dynamic Producer/Consumer: This runtime system exhibits a form of the producer/consumer model in which courier processes are the producers and worker processes are the consumers. In the model described so far, couriers mostly perform remote communication, and workers constantly read-/write data from/to local memory and perform a majority of the FLOPS in this phase of the application. However, we found via performance measurements that this system can still struggle with unacceptable wait times and starvation from the point of view of the workers. This occurs, for example, when $\mathbf{Q}_1$ is empty due to either relatively long communication latencies or not enough couriers to keep the workers busy. For this reason, the WorkQ implementation allows for the dynamic switching of roles between courier and worker.

Fig. 7 displays how role switching occurs in the WorkQ

runtime. To bootstrap the system, *both* couriers and workers perform an initial `get()` operation. This "initialization" of $\mathbf{Q}_1$ is done to avoid unnecessary work starvation due to an empty initial queue. We determined this to be very important for performance, especially when the time to get/enqueue a task is greater than or equal to the compute time. If this is the case (and the number of workers approximately equals the number of couriers), the workers may experience several rounds of starvation before the couriers can "catch up".

After the first round (with workers computing on tiles they themselves collected), workers dequeue subsequent tasks to get data placed in $\mathbf{Q}_1$ by the couriers. If $\mathbf{Q}_1$ ever becomes empty when a worker is ready for a task, the worker will get its own data and carry on. On the other hand, if a courier finds that either $\mathbf{Q}_1$ is overloaded (as determined by a tunable runtime threshold parameter described in section V-C) or that $\mathbf{Q}_2$ is empty with no remaining global tasks, then the courier will become a worker, dequeue a task, and compute the result. In either case, the process will return to its original role as a courier until both $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are empty.

2) Message Queues: The node-wise metadata queues are implemented using the System V (SysV) UNIX interface for message queues. This design decision was made because SysV message queues exhibit the best trade-off between latency/bandwidth measurements and portability compared to other Linux variants [55]. Besides providing atomic access to the queue for both readers and writers, SysV queues also provide priority, so that messages can be directed towards specific consumer processes. For example, this functionality is utilized to efficiently end a round of tasks from a pool: when a courier is aware it has enqueued the final task from the pool, it then enqueues a collection of finalization messages with a process-unique `mtype` value corresponding to the other on-node process IDs.

3) Library API: The WorkQ API provides a productive and portable way for an SPMD application to initialize message queues on each compute node in a distributed-memory system, populate them with data, and dequeue work-items. Here we list a typical series of calls to the API (due to space constraints, arguments are not included, but can be found in the source [40]):

- `workq_create_queue()`: a collective operation which includes on-node MPI multicasts of queue info.
- `workq_alloc_task()`: pass task dimensions and initialize pointer to user-defined metadata structure.
- `workq_append_task()`: push a microtask's metadata and real-data onto the two serialized bundles.
- `workq_enqueue()`: place macrotask bundle into the queue then write real-data into shared memory.

Worker side:

- `workq_dequeue()`: remove a macrotask bundle from the queue and read real-data from shared memory.
- `workq_get_next()`: pop a microtask's metadata and real-data in preparation for computation.
- `workq_execute_task()`: (optional) a callback so data can be computed upon with zero copies.

Finalization:

- `workq_free_shm()`: clean up the shared memory.
- `workq_destroy()`: clean up the message queues.

WorkQ also includes a wrapper to SysV semaphores, which is only needed if the explicit synchronization control is needed (i.e., if certain operations should not occur while workers are computing). These functions are `workq_sem_init()`, `sem_post()`, `sem_release()`, `sem_getvalue()`, and `sem_wait()`.

4) Shared Memory: The message queues just described only contain meta-data regarding tasks - the data itself is stored elsewhere in node-local shared memory. This is done for three reasons: 1) to reduce the cost of contention on the queue among other node-resident processes, 2) Linux kernels typically place more rigid system limits on message sizes in queues (as seen with `ipcs -l` on the command line), and 3) the size and dimension of work-items vary drastically. The message queue protocol benefits in terms of simplicity and performance if each queued item has the same size and structure. Within each enqueued meta-data structure, there are elements describing the size and location of the corresponding task data. The WorkQ library allows for either SysV and POSIX shared memory depending on user preference. There is also an option to utilize MPI_3 shared-memory windows (`MPI_Win_allocate_shared`) within a compute node. This provides a proof-of-concept for doing MPI+MPI [26] hybrid programming within the WorkQ model.

E. TCE Mini-App

The performance of the WorkQ runtime system implementation is evaluated in two ways: directly, with the original NWChem application applied to relevant TCE-CC ground-state energy problems, and indirectly, with a simplified mini-app which captures the overall behavior of the TCE performance bottleneck (described in section II-D). The primary advantage of the mini-app is that it removes the need to filter through the plethora of auxiliary TCE functionalities, such as the TCE hash table lookups, or the many other helper functions within the TCE. Although the mini-app will not compute any meaningful computational chemistry results, it captures the performance behavior of the TCE in a way that is more straight-forward to understand and simpler to tune. Furthermore, the tuned runtime configuration within the mini-app environment can be subsequently applied to NWChem on particular system architectures.

The TCE mini-app implements the pedagogical model described in section IV-D: corresponding tiles from two global arrays (A and B) are multiplied via a DGEMM operation and put back into a third global array C. The mini-app is strictly a weak scaling application that allows for a configurable local buffer length allocation on each MPI process. These buffers are filled with arbitrary data in the creation/initialization of A, B, and C. As in the TCE, all global arrays are reduced to their 1-dimensional representations [25]. The heap and stack sizes fed to the global arrays memory allocator [37] are set to as large as possible on a given architecture. Two versions of the code are implemented to calculate the entire pool of DGEMMs: one with the original `get/compute/put` model on every process, and one with the WorkQ model on every compute node. The resulting calculation is verified in terms of the final vector norm calculated on C.

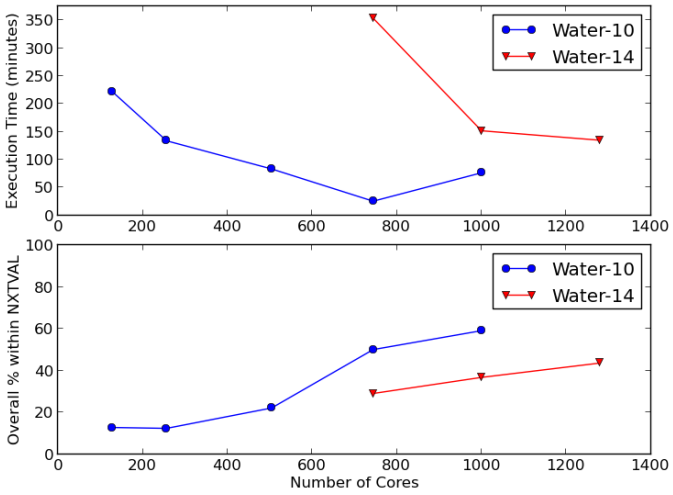

Fig. 8. Total percentage of execution time spent in NXTVAL for a 10-H_2O CCSD simulation (15 iterations) with the aug-cc-pVDZ basis running on the Fusion cluster (without IE). The 14-H_2O test will not fit in global memory on 63 nodes (8 cores per node = 504 cores) or fewer. These data points were extracted from mean inclusive-time profiles as in Fig. 2.

V. Experimental Results

In this section we present performance experiments and measurements to show improvement in execution time for TCE-CC applications in NWChem. Section V-A discusses performance improvements associated with various forms of the IE algorithm, and section V-B discusses experiments in which the WorkQ library is used to overlap communication and computation in the TCE mini-app and in NWChem. Since the WorkQ implementation exposes several runtime parameters, we also briefly discuss machine learning methods of auto-tuning the configurations.

A. Inspector-Executor Experiments

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, 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 [22]. 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. 11, 9, and 10.

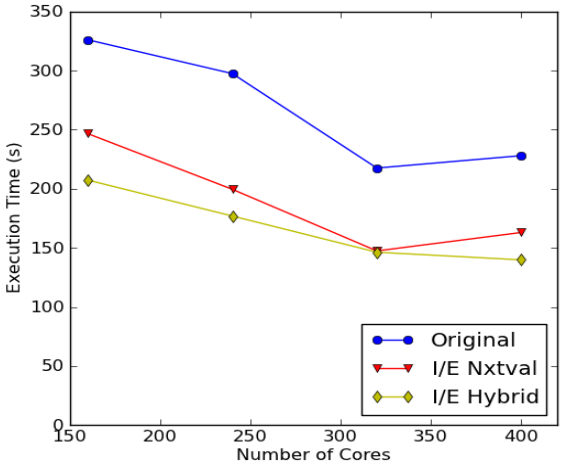

Fig. 9. Benzene aug-cc-pVQZ I/E comparison for a CCSD simulation.

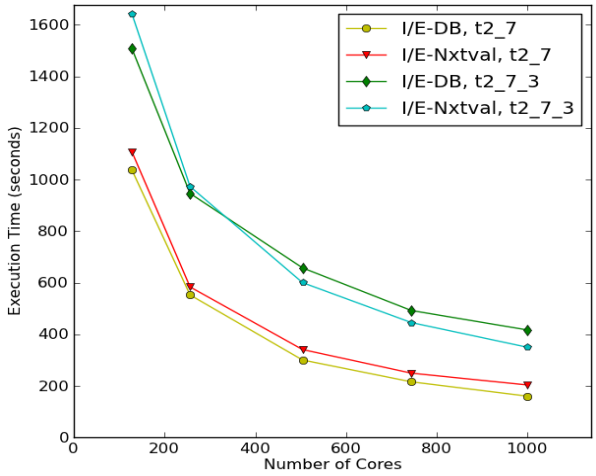

Fig. 10. Comparison of I/E Nxtval with I/E Dynamic Buckets for the two most time consuming tensor contractions during a 10-H_2O simulation, t2_7 and t2_7_3. The execution time of the original code is not shown because it overlaps the performance of I/E Nxtval.

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.

1) 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. 11) 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

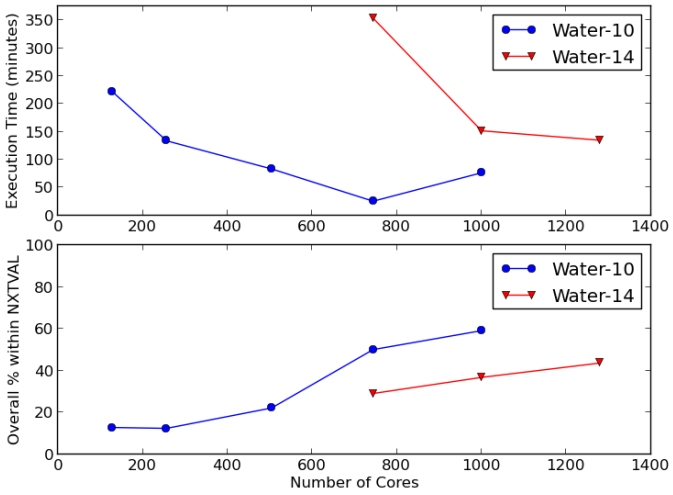

Fig. 11. Total percentage of execution time spent in NXTVAL for a 10-H_2O CCSD simulation (15 iterations) with the aug-cc-pVDZ basis running on the Fusion cluster (without IE). The 14-H_2O test will not fit in global memory on 63 nodes (8 cores per node = 504 cores) or fewer. These data points were extracted from mean inclusive-time profiles as in Fig. 2.

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, $mk + kn$

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.

Fig. 11 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. 1. 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. 11) because of insufficient global memory.

2) 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 categories, symmetrically sparse and dense (i.e., a benzene molecule versus an asymmetric water cluster). We found that problems falling in the sparse

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	-

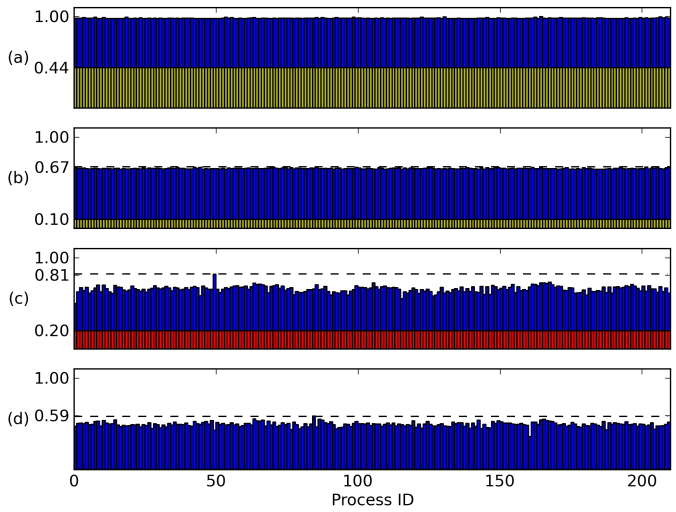

Fig. 12. Comparative load balance of a tensor contraction for benzene CCSD on 210 processes: (a) Original code with total time in NXTVAL overlapped in yellow (all values are normalized to this maximum execution time). (b) I/E with superfluous calls to NXTVAL eliminated. (c) First iteration of I/E with 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 by 8% relative to (b).

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. 9 and 10.

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. 9). 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.

3) 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. 9 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.

4) Dynamic Buckets: I/E Dynamic Buckets (I/E-DB) is usually the method with the best performance, as seen in Fig. 10. 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 IV-C, this is due to error in the task execution time estimations. The I/E-DB technique shows up to 16% improvement over IE-Nxtval due to better load balance when dynamic counters manage groups of tasks.

B. WorkQ Experiments

The performance of the WorkQ execution runtime compared to the standard `get/compute/put` model is evaluated on two different platforms. The first is the ACISS cluster located at the University of Oregon. Experiments are run on the 128 generic compute nodes, each an HP ProLiant SL390 G7 with 12 processor cores per node (2x Intel X5650 2.67 GHz 6-core CPUs) and 72 GB of memory per node. This is a NUMA architecture with one memory controller per processor. ACISS employs a 10 gigabit Ethernet interconnect based on a 1-1 non-blocking Voltaire 8500 10 GigE switch that connects all compute nodes and storage fabric. The operating system is RedHat Enterprise Linux 6.2 and MPICH 3.1 is used with the `-O3` optimization flag.

The second platform is the Carver IBM iDataPlex system provided by NERSC. The compute nodes each have 8 cores (2x Intel Xeon X5550 2.67 GHz quad-core CPUs) and 24GB of memory per node. All nodes are interconnected by 4X QDR InfiniBand technology, providing 32 Gb/s of point-to-point bandwidth. The operating system is Linux running kernel version 2.6.18 and OpenMPI 1.4.5 is used (as of this writing Carver's modules do not include a functioning MVAPICH installation) with `-O3` optimizations.

Unless otherwise specified, performance experiments are executed with 1 MPI process per core, leaving 1 core open on each compute node for the ARMCI helper thread (for example, 11 processes per node on ACISS and 7 processes per node on Carver). Previous work has shown this mapping to be optimal for reducing execution time as suggested by detailed TAU measurements in NWChem TCE-CC [22].

The systems above provide a juxtaposition of the performance benefits gained with the WorkQ runtime between two very different network interconnects: Ethernet and InfiniBand (IB). The GA/ARMCI and MPI layers utilize socket-based connections on ACISS, meaning that the servicing of message requests involves an active role of each compute node's operating system. Carver on the other hand has full RDMA support, so data can be transferred between nodes without involvement of the sender and receiver CPUs. The following sections compare performance of the TCE mini-app and NWChem TCE-CCSD over Ethernet and IB.

1) TCE Mini-App: The first experiment considers the weak scaling performance of the TCE mini-app on ACISS and Carver for two different tile-sizes. The tile-size in the mini-app corresponds to the common dimension of the blocks of data collected from the GAs described in section IV-E. In this experiment, all DGEMM operations are performed on matrices with square dimensions, $N \times N$, where N is the so-called tile-size. Fig. 13 considers tile-sizes 50 (2,500 total double precision floating point elements) and 500 (250,000 elements). The mini-app is a weak-scaling application in which a constant amount of memory is allocated to each process/core at any given scale. That is, if the scale is doubled, then the size of the overall computation is doubled. GA's internal memory allocator is initialized so that the total heap and stack space per node is about 20 GB.

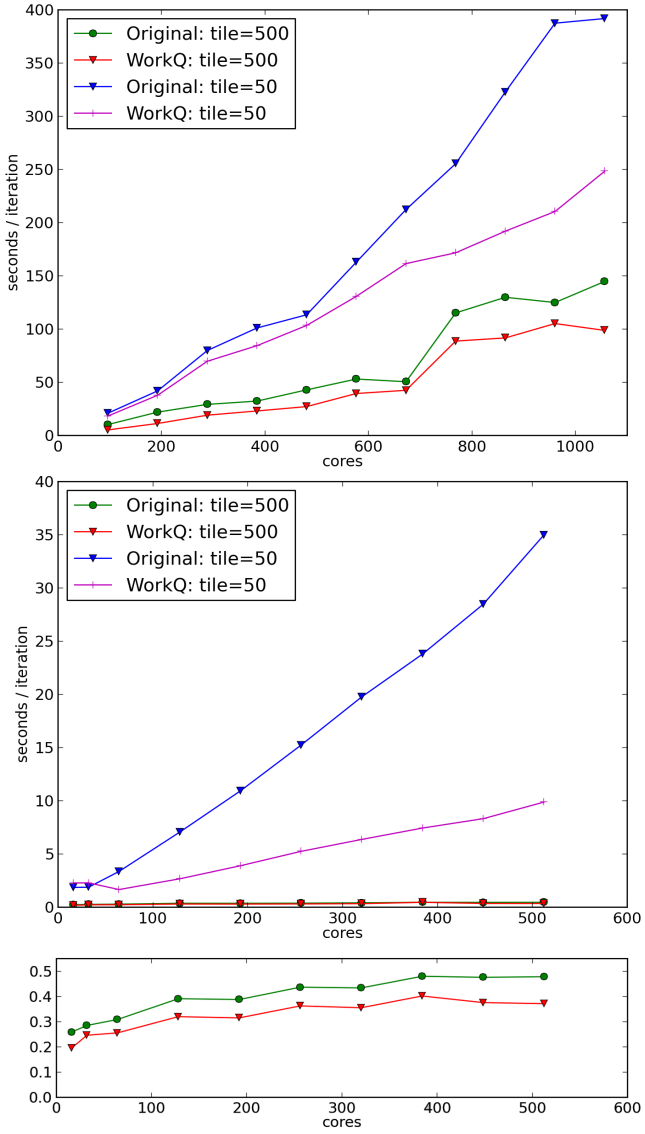

Fig. 13. Weak scaling performance of the TCE mini-app with ARMCI over sockets on ACISS (top) and ARMCI over InfiniBand on Carver (middle) for different tile-sizes. On ACISS, the WorkQ implementation was run with 6 courier processes and 5 worker processes, and on Carver, with 3 couriers and 4 workers. The bottom plot is a zoomed view of the 500 tile-size execution on Carver. On both architectures, the WorkQ execution shows better relative speedup with small tile-sizes, but better absolute performance for relatively larger tile-sizes.

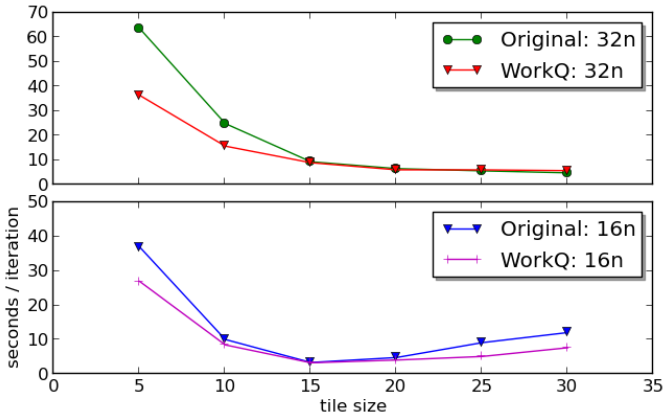

Fig. 14. Time per CCSD iteration for w3 aug-cc-pVDZ on ACISS versus tile-size. The top row contains execution measurements on 32 nodes (384 MPI processes) and the bottom row contains measurements on 16 nodes (192 MPI processes).

Fig. 13 clearly shows that using the relatively large tile-size of 500 results in better overall absolute performance for both the WorkQ execution model and the original execution model. This phenomenon is well-understood [41] and mainly due to the overhead associated with data management and dynamic load balancing when tile-size is relatively small. In general, larger tile-sizes are desirable so as to minimize this overhead, but at a certain point large tiles are detrimental to performance because it leads to work-starvation. For instance, if there are more processes/cores available to the application than there are number of tiles, then work-starvation will surely occur.

On the other hand, the best speedups achieved with the WorkQ model on both systems are seen using the smaller tile-size of 50, particularly at relatively large scales. Our TAU profiles show that at a tile-size of 50, the total time spent in communication calls (`ARMCI_NbGetS` and `ARMCI_NbPutS`) is considerably larger than with a tile-size of 500. This suggests that at smaller tile-sizes, there is more cumulative overhead from performing one-sided operations, and therefore more likelihood that processes will spend time waiting on communication. This scenario results in more opportunity for overlap, but worse absolute performance due to the incurred overhead of dealing with more tasks than necessary.

When tuned appropriately, the WorkQ execution model can attain a speedup of about 2 from the complete overlap of communication with computation. However, at certain configurations the speedup is greater than 2 (the largest is about 3.5 with 512 MPI processes on Carver). The origin of this performance benefit can be inferred from traces such as Fig. 4 (more traces are not included due to space constraints). Occasionally the average time spent in corresponding communication calls is longer in the original model than in the WorkQ model. The original model can fall into lock-step pattern in which all processes are communicating and computing at the same time, respectively. This causes system network contention and local memory contention that is not seen in the WorkQ traces.

2) NWChem: We now analyze the performance of the WorkQ model applied to the TCE in NWChem by measuring the time of execution to calculate the total energy of water molecule clusters. These problems are important due to their prevalence in diverse chemical and biological environments [2]. We examine the performance of the tensor contraction which consistently consumes the most execution time in the TCE CCSD calculation, corresponding to the term:

$$r_{h_1 h_2}^{p_3 p_4} \mathrel{+}= \frac{1}{2} t_{h_1 h_2}^{p_5 p_6} v_{p_5 p_6}^{p_3 p_4}$$

(see [25] for details regarding the above notation). In the TCE, this calculation is encapsulated within routine `ccsd_t2_8()` and occurs once per iteration of the Jacobi method.

Fig. 14 shows the minimum measured time spent in an iteration of `ccsd_t2_8()` on a 3-water molecule cluster using the aug-cc-pVDZ basis set across a range of tile-sizes. These measurements are on the ACISS cluster at two different scales: 32 compute nodes in the top plot and 16 compute nodes in the bottom plot with 12 cores per node in each case. Here we use the minimum measured execution time for a series of runs because it is more reproducible than the average time [21]. On 16 nodes, we see overall performance improvement with WorkQ across all measured tile-sizes. As in the TCE mini-app (Fig. 13), WorkQ shows better performance improvement at small tile-sizes but best absolute performance with a medium sized tile. This relatively small input problem does not strong-scale well to 32 nodes, which is evident by the execution times and the fact that performance stays constant after a certain tile-size. The WorkQ system lessens this problem at small tile-sizes.

The performance of NWChem over IB is quite different than over Ethernet. The WorkQ system shows reasonable speedup in the mini-app with intelligent choices in runtime parameters (discussed in the next section), but relatively poor performance with other parameter choices. Fig. 15 compares the communication and computation time distributions of the mini-app and TCE/CCSD on a 5-water cluster with the aug-cc-pVDZ basis set. Although the mini-app can be tuned to match the distribution of task times for the NWChem application (top row of Fig. 15), this parameter set shows poor speedup with WorkQ ($<$2%). The mini-app can achieve over 3x speedup with larger tiles (bottom row of Fig. 15), but TCE's node-local memory requirements scale as (tile-size)^4 in CCSD. This leads to the application running out of memory at these corresponding larger tile-sizes at a 32-node scale. The application will run successfully at larger scales, but converges towards the performance of the work-starved scenario of Fig. 14.

C. Mini-App Auto-tuning

In this section our efforts in auto-tuning the WorkQ runtime using machine learning (ML) algorithms are briefly described. It is well known that choosing performance-optimal compile-time and runtime parameters in parallel systems is an inherently difficult problem [11]. This is due to the unknown and potentially complicated relationships between parameters and the sheer size of the configuration space. For example, Fig. 16 shows the variation in speedup of the WorkQ TCE mini-app with respect to a baseline configuration for a small subset of the total configuration space. The performance of each set of parameters depends on the memory occupancy, and the best configuration at high occupancy may be a poor configuration at low occupancy. ML techniques have been used to tackle this problem, for instance by optimizing the MPI

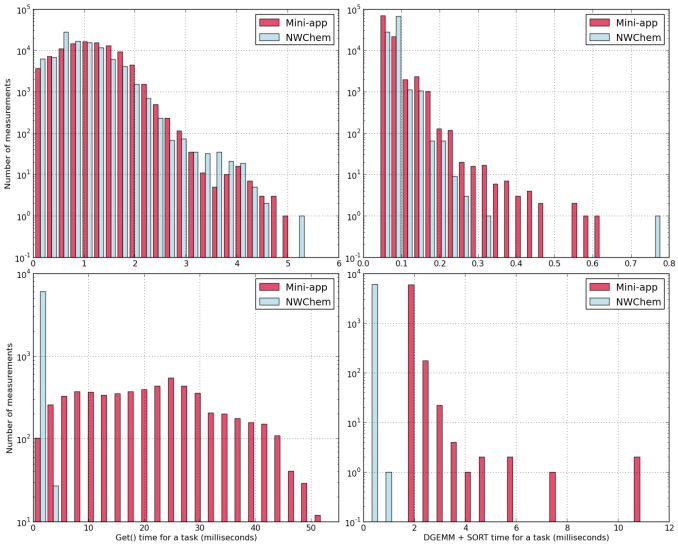

Fig. 15. Log-scale histograms of communication time measurements (left column) and computation time (right column). The top row corresponds to a parameter set chosen to mimic the task pool in a 5-water cluster in TCE/CCSD NWChem on a 32-node IB interconnect, but this set shows little performance improvement in the mini-app. The bottom row shows a different parameter set which achieves optimal speedup in the mini-app by using the WorkQ model, but does not match the distribution pattern of the 5-water task pool.

runtime parameters [44]. In these techniques, a set of training data is collected and measured, and an ML model is created to predict the performance of unseen instances. The WorkQ runtime exposes the following parameters that serve as features for an ML model:

- Number of processes per node
- Number of courier processes per node
- Tile-size
- Min. length of the work queue before a role-switch
- Max. length of the work queue before a role-switch

We use the Scikit-Learn tool [42] to construct a simple regression-based decision tree model from a set of training data. In this tree, each node corresponds to a range of one of the WorkQ parameters above, and each leaf node is the predicted speedup for a given path. For unknown instances, we can choose the leaf with the highest speedup and work back up the tree to select the ideal set of runtime parameters. Using this approach, we find that a tree with a depth of only 4 results in the lowest root mean squared error for performance prediction.

VI. Related Work

A. Inspector-Executor Load Balancing

Alexeev and coworkers have applied novel static load balancing techniques to the fragment molecular orbital (FMO) method [1]. 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.

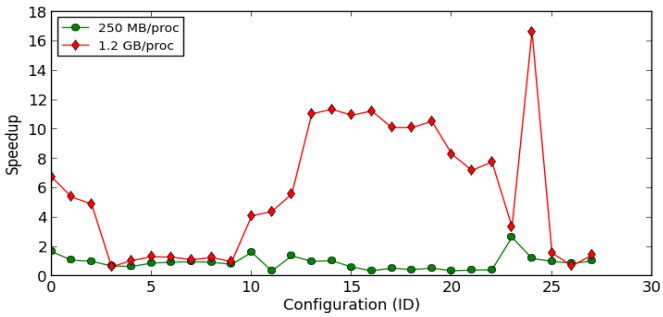

Fig. 16. Variation in speedup across several sets of runtime configuration parameters. The two lines compare mini-app executions with different amounts of Global Array memory per process. Some parameters combinations perform well in one case, but not the other. In particular, configuration #23 is a good choice with 250 MB/proc but a relatively poor choice with 1.2 GB/proc.

Hypergraph partitioning was used by Krishnamoorthy and coworkers to schedule tasks originating from tensor contractions [30]. 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 [50], [51] 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 [58]. 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 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.

B. WorkQ

We are unaware of any dynamic producer/consumer (or publish/subscribe) model in which role switching occurs for the sake of performance in distributed memory systems. However, the publish/subscribe model is quite common, and takes many other dynamic forms [17], [56].

Interesting developments in wait-free and lock-free queuing algorithms with multiple enqueuers and dequeuers could potentially improve performance of this execution system [28]. SysV and POSIX queues provide atomicity and synchronization in a portable manner, but neither are wait-free or lock-free.

The sophistication of the behavior of the WorkQ system makes it fairly difficult to understand the exact characteristics of the performance improvement, and why certain configurations work better than others. We have begun work in

constructing a discrete event simulation of the system in order to develop a model to potentially clarify the dynamics, and to allow for independent tuning of the runtime parameters in conjunction with performance analysis.

VII. Conclusions and Future Work

In this section we present our conclusions regarding inspector-executor load balancing (section VII-A) and the new WorkQ execution model (section VII-B). Finally, section VII-C describes our future goals to integrate the inspector-executor paradigm with the WorkQ implementation.

A. Inspector-Executor

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 [30].

B. WorkQ

The `get/compute/put` model is a common approach for processing a global pool of tasks, particularly in PGAS applications. This model suffers from unnecessary wait times on communication and data migration that could potentially be overlapped with computation and node-level activities. The WorkQ model introduces an SPMD-style programming technique in which node-wise message queues are initialized on each compute node. A configurable number of courier processes dedicate their efforts towards communication and populating the queue with data. The remaining worker processes dequeue and compute tasks. We show that a mini-application which emulates the performance bottleneck of the TCE achieves performance speedups up to 3.5x with the WorkQ model. We also show that WorkQ can improve the performance of NWChem TCE-CCSD across many tile-sizes on the ACISS cluster. The TCE mini-app is configured to mimic the tasks of NWChem over InfiniBand and the distributions of communication and computation times suggest why performance improvement there is more difficult.

C. Future Work: Inspector-Executor and WorkQ Integration

While this paper has presented the inspector-executor and WorkQ models somewhat tangentially, they are in-fact quite compatible with one another. Clearly, the inspector entity introduces a new layer of functionality and complexity to the runtime of the TCE in NWChem. However, the executor (as seen in Alg. 4) is actually quite similar to the original TCE generated code (Alg. 2) - it only lacks the nested loops over tiles and the implicit global dynamic load balancing. It is clear that the WorkQ implementation itself is independent of the inspector entity. On the other hand, the WorkQ execution comprises the *entire* executor entity. In a sense, the WorkQ model is an improvement on the executor, so it is completely compatible with the IE model. It should be a fairly straightforward matter to combine the two models into a single, coherent execution engine that simultaneously reduces overhead from centralized dynamic load balancing and effectively overlaps communication with computation.

References

[1] 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.

[2] Edoardo Aprà, Alistair P. Rendell, Robert J. Harrison, Vinod Tipparaju, Wibe A. deJong, and Sotiris S. Xantheas. Liquid Water: Obtaining the Right Answer for the Right Reasons. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 66:1–66:7, New York, NY, USA, 2009. ACM.

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

[5] Rodney J. Bartlett and Monika Musiał. Coupled-cluster theory in quantum chemistry. 79(1):291–352, 2007.

[6] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10, April 2006.

[7] R.F. Bishop. An Overview of Coupled Cluster Theory and Its Applications in Physics. *Theoretica chimica acta*, 80(2-3):95–148, 1991.

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

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

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

[11] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M.F.P. O'Boyle, and O. Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, pages 185–197, March 2007.

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

[13] T. Daniel Crawford and Henry F. Schaefer III. An introduction to coupled cluster theory for computational chemists. volume 14, chapter 2, pages 33–136. VCH Publishers, New York, 2000.

[14] 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.

[15] J. Dinan, P. Balaji, J.R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 739–750, May 2012.

[16] 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.

[17] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

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

[19] 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. 95(4):2623–2638, 1991.

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

[21] William Gropp and Ewing Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer Berlin Heidelberg, 1999.

[22] 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.

[23] R. J. Harrison. Portable Tools and Applications for Parallel Computers. *International Journal of Quantum Chemistry*, 40(6):847–863, 1991.

[24] Robert J. Harrison. Portable tools and applications for parallel computers. 40(6):847–863, 1991.

[25] So Hirata. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[26] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121–1136, 2013.

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

[28] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[29] Karol Kowalski, Jeff R Hammond, Wibe A de Jong, Peng-Dong Fan, Marat Valiev, Dunyou Wang, Niranjan Govind, and JR Reimers. *Coupled cluster calculations for large molecular and extended systems*. Wiley Hoboken, NJ, 2011.

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

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

[32] 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. 80:387–405, 1991.

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

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

[35] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer Berlin Heidelberg, 1999.

[36] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, pages 533–546, London, UK, 1999. Springer-Verlag.

[37] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[38] Jozef Noga and Rodney J. Bartlett. The full CCSDT model for molecular electronic structure. 86(12):7041–7050, 1987.

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

[40] D. Ozog. TCE mini-app source code repository. https://github.com/davidozog/NWChem-mini-app.

[41] D. Ozog, J.R. Hammond, J. Dinan, P. Balaji, S. Shende, and A. Malony. Inspector-Executor Load Balancing Algorithms for Block-Sparse Tensor Contractions. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 30–39, Oct 2013.

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[43] 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.

[44] S. Pellegrini, R. Prodan, and T. Fahringer. Tuning MPI Runtime Parameter Setting for High Performance Computing. In *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pages 213–221, Sept 2012.

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

[46] Michael J. Quinn and Philip J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(2):197 – 204, 1996.

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

[48] J.C. Sancho, K.J. Barker, D.K. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 17–17, Nov 2006.

[49] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

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

[51] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions. may 2013.

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

[53] John F. Stanton. Why CCSD(T) works: a different perspective. 281:130–134, 1997.

[54] 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. 94(6):4334–4345, 1991.

[55] W. Richard Stevens. *UNIX Network Programming, Volume 2 (2Nd Ed.): Interprocess Communications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[56] Muhammad Adnan Tariq, Gerald G. Koch, Boris Koldehofe, Imran Khan, and Kurt Rothermel. Dynamic Publish/Subscribe to Meet Subscriber-Defined Delay and Bandwidth Constraints. In *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 458–470. Springer Berlin Heidelberg, 2010.

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

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

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

[60] Takeshi Yanai, Haruyuki Nakano, Takahito Nakajima, Takao Tsuneda, So Hirata, Yukio Kawashima, Yoshihide Nakao, Muneaki Kamiya, Hideo Sekino, and Kimihiko Hirao. UTChem A Program for ab initio Quantum Chemistry. In Peter M.A. Sloot, David Abramson, Alexander V. Bogdanov, Yuriy E. Gorbachev, Jack J. Dongarra, and Albert Y. Zomaya, editors, *Computational Science ICCS 2003*, volume 2660 of *Lecture Notes in Computer Science*, pages 84–95. Springer Berlin Heidelberg, 2003.