Analysis and Tuning of Libtensor Framework on Multicore Architectures

Khaled Z. Ibrahim, Samuel W. Williams
Computational Research Division,
Lawrence Berkeley National Laboratory,
Berkeley, CA, USA
{kzibrahim, swwilliams}@lbl.gov

Evgeny Epifanovsky, Anna I. Krylov
Department of Chemistry
University of Southern California,
Los Angeles, CA, USA
{epifanov, krylov}@usc.edu

***Abstract*—Libtensor is a framework designed to implement the tensor contractions arising form the coupled cluster and equations of motion computational quantum chemistry equations. It has been optimized for symmetry and sparsity to be memory efficient. This allows it to run efficiently on the ubiquitous and cost-effective SMP architectures. Unfortunately, movement of memory controllers on chip has endowed these SMP systems with strong NUMA properties. Moreover, the manycore trend in processor architecture demands that the implementation be extremely thread-scalable on node. To date, Libtensor has been generally agnostic of these effects. To that end, in this paper, we explore a number of optimization techniques including a thread-friendly and NUMA-aware memory allocator and garbage collector, tuning the tensor tiling factor, and tuning the scheduling quanta. In the end, our optimizations can improve the performance of contractions implemented in Libtensor by up to 2× on representative Ivy Bridge, Nehalem, and Opteron SMPs.**

***Index Terms*—tensor algebra, parallel programming, quantum chemistry software.**

I. Introduction

Quantum chemistry methods are based on the exact quantum-mechanical principles; they enable predictive studies of electronic structure, chemical properties, and spectroscopy [9]. Practical methods are based on approximate solutions of the Schroedinger equations, yet, their cost scales polynomially rather than linearly with the system size. This curbs the size of systems that can be studied by these methods. The time required to perform a certain calculation is also important; for example, certain types of research (e.g., condensed phase) require many thousands of calculations on relatively small systems. Thus, efficient implementations of electronic structure methods have always been crucial for the computational chemistry community. Taking advantage of parallel architectures mitigates the scaling problem and also reduces time to perform a given calculation.

Power-efficiency is motivating computer architects to rapidly increase the number of cores per compute node, instead of increasing the computational capabilities (e.g. frequency or instruction-level parallelism) within a core. Similarly, performance motivations demand a complex memory hierarchy. In addition to multi-level caching, the memory systems could be split across multiple domains, creating a non-uniform memory access (NUMA) address space. Increasing parallelism requires rethinking algorithms to eliminate most serialization events. Complex memory systems make explicit management of locality a necessity to achieve a good performance.

Quantum chemistry methods based on many-particle computations typically involve tensor computations. Tensor computation exists in many other physical and computational sciences. The biggest challenges for such computation are the large dataset and high computational demands.

In this work, we explore performance analysis and tuning for the Libtensor [5] framework. This framework provides an efficient engine for quantum chemistry computation using the shared-memory programming model. In order to solve larger problems in a constrained physical memory, Libtensor takes advantage of tensor sparsity that arises from spin and point symmetry. To load balance the workload, Libtensor uses a task-based computational model. The library outperforms other efficient implementations, such as Molpro [18] for the same class of problems as detailed in [5].

This paper tackles the challenge of improving the scalability of Libtensor framework with respect to thread concurrency. Nominally, this framework typically attains a sublinear speedup and reduced efficiency as one increases the core count. The work presented in this paper analyzes the breakdown of execution time and explains the reasons for impaired scaling. This framework faces challenges in exploiting locality, load-balancing the computation, and avoiding serialization. In this work, we devise solutions to tackle such challenges and achieve better performance and scaling behavior.

This paper makes the following contributions. It provides an in-depth analysis of the computational constraints of the tensor contractions in Libtensor framework. We specifically provide analysis for the locality management challenges and the load-balancing difficulties. We present a new memory management system that allows better control of locality and also reduces the overhead of memory management under high concurrency. Additionally, we demonstrate how to achieve better scaling by tuning several key parameters. The presented strategy achieves a 2× performance gain over the tuned baseline implementation on a 32-core AMD opteron-based systems as well as a 24-core Intel Ivy-bridge.

The rest of this paper is organized as follows. Section introduce a brief review of related work. In Section III, we present the experimental setup. In Section IV, we briefly intro-

TABLE I
TEST CASES FOR CCSD CALCULATIONS.

	P1	P2	P3
Problem	Uracil cc-pVTZ	methylated uracil water dimer (mU—H_2O) 6-311+G(d,p)	methylated uracil water cluster $(mU)_2$—H_2O 6-31+G(d,p)
Basis functions	296	302	489
Symmetry	C_s	C_s	C_1

duce Libtensor framework and its computational constraints. In depth analysis for computation constraints in Libtensor is introduced in Section V. Our performance optimization and tuning efforts are detailed in Section VI. Section VIII concludes our paper.

II. RELATED WORK

Tensor computations, being a natural extension of matrix computations to many dimensions, are used to solve many-body problems in physics, in particular in quantum chemistry and nuclear physics. Most of the tools previously developed are specific to their applications and lack transferability. Among those targeting large-scale distributed systems are TCE [10], the ACES/SIAL framework [15]. CFOUR [17] and MRCC [13] target both distributed and shared-memory architectures, but also remain application-specific. Recently, general-purpose tensor tools started to receive more attention, for example CTF [16] and TiledArray [4].

Generally speaking, there are two primary challenges that all of these tensor tools face when converting a physical problem into an efficient computer program. First, they need to provide an interface that allows one to describe the physical problem as a set of tensor equations, and second, there needs to be a runtime environment that is capable of computing those equations. TCE and SIAL handle the complexity by enabling one to write an electronic structure operator and compute its tensor representation. Programming is done through code generation in Fortran (TCE) or a domain-specific programming language (SIAL). CTF and TiledArray provide a general programming interface making them open to applications outside of electronic structure theory.

From the parallel communication standpoint, the packages mentioned here use either a one-sided model, for instance the use of global arrays in NWChem, or a two-sided model such as MPI in CTF. While providing attractive solutions for large-scale runs, they typically suffer large overheads on small machines. Moreover, they typically rely on the aggregation of the physical memory of many light-weight compute nodes in order to accommodate a large problem. Working in a distributed memory environment influences many of their design choices. For instance, exploitation of some form of symmetry and sparsity can lead to complex communication patterns. In such cases, scaling (parallel) efficient implementations can dictate some performance inefficient consequences. These packages face similar challenges, including how to load-balance the computation, how to manage locality, and how to efficiently execute computation while providing an easy interface.

TABLE II
SYSTEMS USED IN THIS STUDY.

	Edison	Trestles	Carver
Core Arch	Intel Ivy Bridge	AMD Magny-Cours	Intel Nehalem-EX
Clock (GHz)	2.4	2.4	2.0
Cores	24	32	32
DP Gflops	461	307	256
D$/core(KB)	32+256	64+512	32+256
LL$/chip(MB)	30	5	18
NUMA domains	2	8	4
Memory (GB)	64	64	1000
System Software			
Complier	Intel *icc* 14.0.2	Intel *icc* 13.0.1	Intel *icc* 13.0.1
BLAS Routine	Intel *mkl* 13.0.3	Intel *mkl* 13.0.1	Intel *mkl* 13.0.1

Libtensor [5] is a general-purpose tensor algebra library that targets problems that can be solved within a single node. It adopts the shared-memory programming model and exploits multiple forms of symmetry including permutational symmetry, spin symmetry, and point group symmetry to minimize memory usage and floating point operation count.

III. EXPERIMENTAL SETUP

A. Test configurations

Table I summarizes three canonical test cases used in this study. These examples represent typical mid-size systems amenable to high-level electronic structure calculations. All three cases are of a closed-shell type; thus, they exploit both permutational and spin symmetries that reduce the size of unique data. Examples 1 and 2 also feature moderate point-group symmetry.

B. Evaluated Systems

In this study, we used the three systems listed in Table II. These systems have 24-32 cores per compute node. They also exhibit different levels of NUMA locality, core count, caching systems, and physical memory. The strongest NUMA effects are associated with Trestles which is based on the AMD Magny-Cours architecture and presents 8 NUMA domains. Carver nodes, based on Intel Nehalem-EX, have 1TB of physical memory and may thus run the large problems efficiently (without paging). The Edison nodes, based on Intel Ivy Bridge, are the most computationally capable, and also have the largest cache capacity of the three computing systems.

As the computations were dominated by DGEMM calls, on all systems, we used Intel MKL implementations of BLAS routines to maximize performance.

IV. COMPUTATIONAL CHEMISTRY USING LIBTENSOR FRAMEWORK

This section provides a brief overview of Libtensor framework, depicted in Figure 1. The framework is composed of a layered stack of libraries. At the top of the stack, the LibCC library provides an easy interface to express computational chemistry equations including coupled cluster (CC) and equation of motion (EOM). LibCC uses Libtensor to implement the tensor contractions. In turn, Libtensor uses multiple libraries to manage threading and dynamic memory management. For further details refer to [5].

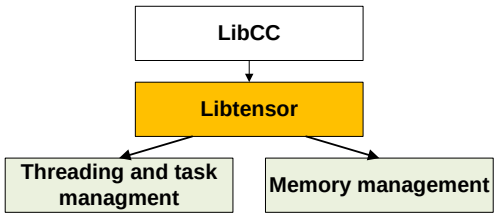


Fig. 1. A simplified depiction of Libtensor framework. The top library (LibCC) facilitates the expression of computational chemistry equations. Tensor computations are done by the Libtensor library using underlying libraries to manage memory and tasking.

A. Computational Chemistry and Libtensor

Computational methods in electronic structure theory range from relatively unsophisticated mean-field methods to state-of-the-art many-body approaches [9]. Coupled cluster (CC) theory offers an hierarchy of practical and systematically improvable many-body computational methods that are most conveniently formulated in terms of tensor operations. Early implementations of CC methods used equations based on explicit matrix multiplication, an approach that requires explicit data manipulation and makes it very hard to develop new methods. This motivated the development of computational tools for tensors that address both the complexity of equations and computational performance. Libtensor provides a natural programming interface that resembles actual mathematical expressions, and a modular internal structure that allows one to adapt the algorithms to new computer architectures without affecting high-level codes. Libtensor powers a family of CC methods in Q-Chem, a popular general-purpose electronic structure package [14]. While offering a convenient API, Libtensor's efficiency is competitive with the best codes based on explicit matrix multiplication and data handling.

B. Block-Tensor

In currently practical CC calculations, tensor sizes are in the gigabyte to terabyte range.To manipulate these arrays of data Libtensor assumes the block-tensor format for the tensor objects. In this representation tensors are broken down into tiles along each dimension such that each resulting small block represents a window in the original tensor and has the same number of dimensions. Tiling size determines the total size of each multi-dimensional block thus affecting the size of the data elements operated on in the divide-and-conquer algorithms for block-tensor algebra.

C. Handling Symmetry and Sparsity

Symmetries in the physical model in many-body theories lead to structure in the tensors used for model description. It is computationally advantageous to account for these symmetries, which at the tensor level manifest as symmetry between tensor entries and sparsity. The block-tensor data structure stores only non-zero symmetry-unique (canonical) blocks and symmetry metadata for obtaining blocks related by symmetry. All block-tensor algorithms must therefore be aware of symmetry and sparsity, and usually a tensor operation is performed in two steps: first the symmetry and sparsity of result is computed, and second the computation of canonical output blocks is performed.

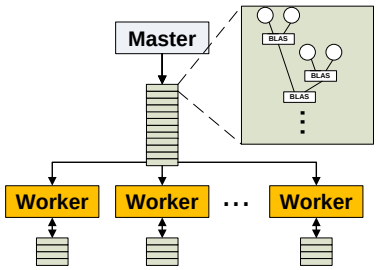


Fig. 2. Baseline Libtensor tasking model. Dependent subtasks are grouped together into a task. Task independence reduces scheduling overhead.

D. Tasking Model

Libtensor uses a simple tasking model, shown in Figure 2 in which the "master" thread is responsible for generating a series of task lists based on the tensor operations. The task lists, generated by the master, are placed into a shared queue for all "worker" threads. The master notifies workers with the availability of tasks and relinquishes its use of the compute resources. Upon notification of task availability, worker threads start moving tasks from the shared queue to local (thread-private) queues. Workers execute their local tasks, and upon completion they check for the availability of more tasks. If more tasks are available, they simply repeat the previous steps. Upon finishing all tasks, the master is awaken to generate the next operation's tasks and the workers sleep.

V. Analysis of The Computational Constraints for Libtensor Framework

Tensor frameworks have a complex set of constraints that make performance tuning a challenge. The Libtensor framework tries to solve computational chemistry problems with large datasets using shared memory machines. The challenge is to achieve efficient execution within constrained resources. This section discusses the constraints that Libtensor considers.

A. Computational Composability and Productivity

One of the major computational challenges that most tensor frameworks face is the numerous computational patterns that the framework needs to support. Most frameworks provide an easy interface for computational scientists to express their mathematical formulations in a productive way. The computation is not only affected by the operation type, but also by the data content attributes, for instance symmetry and sparsity.

To ease the problem representation, data manipulation and storage properties are handled in a decoupled fashion. Computation always uses the dense representation of tensors. The storage of a tensor can be optimized based on the symmetry and sparsity properties, which is specified by separate APIs. This makes most of the information about the computation available only at runtime. Obviously, this reduces the chance for static analysis or static tuning. The computation is typically composed of multiple independent steps. Ideally, tuning for a

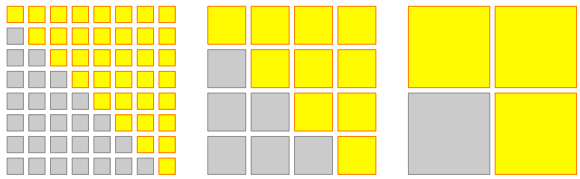


Fig. 3. The impact of tiling size on the exploitation of symmetry. Larger tiles cause explicit storage of non-canonical elements. For simplicity, we show the two-dimensional case.

particular step should not negatively affect other steps. Obviously, composition of locally optimized steps does necessarily guarantee global optimal performance.

B. Memory Requirements

The memory requirements of storing a tensor grow rapidly with the basis set size, associated with molecular orbitals in the studied problems, because of the high dimensionality of tensors (typically in the order of 4 to 6 in the presented cases). Fitting the dataset in the physical memory is critical to performance. The alternative is to swap the dataset in and out of the slow disk system and incur the large I/O latency and limited bandwidth.

The typical technique is to exploit the physical properties of the dataset to save storage. For instance, symmetry can be used to save space by storing only canonical blocks and use metadata to construct the remaining data. Similarly, data sparsity (with zero values) are not stored explicitly. Instead, they are replaced by meta data.

The high-level description of the problem equations typically use the dense format for representing the equations. The symmetry properties of the data are provided by the user using separate APIs. Although this leads to a clean description of the problem, it imposes the following challenges. The processing time for similar computations can differ at runtime depending on the number of encountered zero or symmetric blocks. Zero blocks reduce processing, while non-explicitly stored symmetric blocks require additional processing to allocate a buffer and to apply the symmetry operator.

The symmetry and sparsity properties are specified at the block level as opposed to the natural element-wise level. The objective is to reduce the ratio of meta data to the actual floating-point data. The memory management, involving frequent allocation and deallocation of the memory, is a critical component of the system. This helps in reducing the volume of the active dataset at runtime. This reduction impacts the processing time for identical computations, thus leading to load imbalance, and requiring the adoption of a tasking runtime. In Figure 3, we show the implication of the choice of the tile size on the exploitation of symmetry. The larger the tile-size, the less saving based on symmetry. Larger tiles typically lead to explicitly storing un-needed elements especially across the diagonal blocks. On the other hand, the smaller the tile size, the larger the meta-data needed for non-explicitly stored blocks.

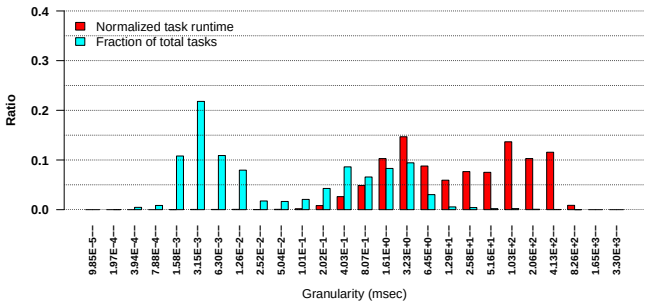


Fig. 4. Contribution of different task granularities (visits) to the execution time (total visits).

C. Load Balancing

The use of master/worker tasking model in Libtensor framework stems from the difficulty to predict the computation cost of a tensor operation, especially if storage optimization are used. Typically a tensor contraction can be split into logically independent tasks. Even for equally sized tasks, the computation time depends on contents of the blocks (sparsity) and whether they are explicitly stored (symmetry). This creates a load-balance problem that a tasking model could handle. Libtensor groups dependent computation into larger logical tasks. Therefore, all tasks in the execution queues are always independent. This alleviates the need for managing dependency by the tasking runtime.

The load imbalance is problematic, whether we consider the task level or subtasks (BLAS routines within a task). In Figure 4, we show a histogram of different task granularities as a ratio of the total execution time (red) as well as the corresponding number of visits (blue) to these task granularities during a typical computation[1]. As shown, the task granularity (run time) varies greatly in size, without a dominant task size. This shows why load balancing is critical to performance. The more alarming fact is that the vast majority of the task visits are for fine-grained tasks. Many tasks need just a few thousand cycles to complete. The implication is that the tasking runtime should not incur a large overhead in making the scheduling decision while dealing with severe load imbalance lest it become the performance bottleneck.

Unfortunately, targeting subtasks (the BLAS routines) does not lead to a less imposing task distribution. As shown in Figure 5, the same trends for task distribution and visits are experienced, except that they are associated with finer-grained tasks. If subtasks are considered as the base scheduling quanta, the scheduler should explicitly manage task dependencies. Currently, Libtensor scheduler can handle only independent tasks. This allows low-overhead scheduling decisions.

D. Data locality

The performance of BLAS routines, typically used in tensor contractions, depends on the locality of the data. In most architectures, cache hierarchies try to capture temporal locality when repeatedly accessing data. Additionally, to scale the core

[1]This experiment is done for problem P1 on Edison node with 24-way parallelism.

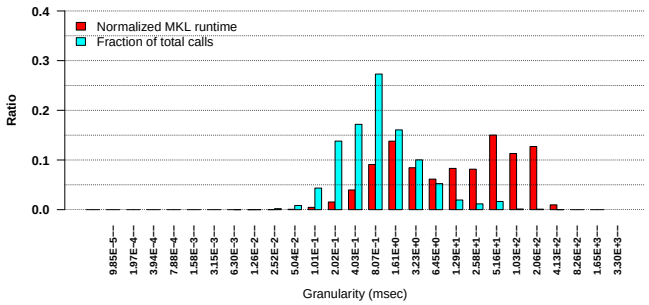


Fig. 5. Contribution of different BLAS task granularities (visits) to the execution time (total visits).

count while allowing a high bandwidth to the memory, architects design compute nodes with multiple memory domains, creating non-uniform memory access (NUMA).

The location of the input data set influences the performance of executing the BLAS routines. Obviously, if the input data already resides in cache, then the time required to perform the computation is greatly reduced. If data does not reside in cache and the computation is memory latency bound, then it is more efficient to fetch data from the same NUMA domain. Conversely, if the computation is memory bandwidth limited, then the distribution of data across multiple NUMA domains can improve the performance by ensuring all memory controllers are utilized. Ideally, we assign executing threads tasks that maximize locality.

Programmers can reason about these constraints if the code is statically scheduled. The challenge for our tensor computations is that tasks are assigned dynamically to improve the load balance. Another challenge is that while a thread may reason about the efficiency of executing a task, it cannot resolve its relative merit compared with other threads. Resolving competing efficiencies will require centralized decision for global optimality, leading to a non-scalable scheme.

The tension between managing locality and load balance is partly solved in this study by improving the memory management system as discussed in Section VI.

E. Parallelism and Task Granularity

The task granularity can be controlled by using the tiling factor of the tensors. For most of the computation we present in this study, the virtual orbits have the largest base values. Thus, their data structures dominate the dataset.

Clearly, larger tiles increase the task granularity, but the number of tasks is reduced. A larger task count allows for increased opportunity for load balancing and for scaling across multiple cores. On the other hand, it increases the sensitivity to scheduling overheads. The tradeoffs are difficult to resolve statically. Whether an application benefits from the increased parallelism or reduced scheduling overheads, is a question that we will try to answer through runtime space exploration with different scheduling parameterizations.

The task creation in Libtensor is inherently a serial operation that is performed solely by the master thread. Additionally, scheduling decisions are also inherently serial, with the caveat that workers may move multiple tasks from the main queue

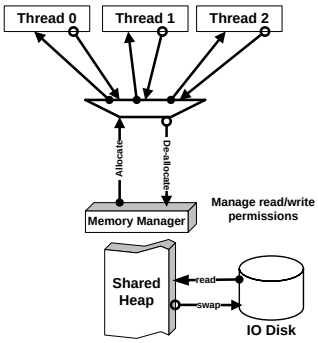


Fig. 6. Base memory allocation systems. Multithreaded access is serialized to provide mutually exclusive access to the shared heap.

to their local queues to make the serialization events brief. Memory management (allocation and deallocation) events need to be serialized if the memory pool is shared across all threads. These serialization instances can limit the maximum attainable speedup based on Amdahl's law [1].

VI. Performance Tuning and Refinement

Performance tuning for tensor contractions is a challenging task because the computational pattern is influenced by many runtime conditions. In this section, we discuss some of the techniques we adopted to improve the performance tensor computations in Libtensor.

A. Explicit Locality Management

The base Libtensor implementation does not handle locality in any fashion because of the difficulty in performing static analysis and the requirement of load balancing. Runtime locality analysis can increase scheduling overheads and load balancing can conflict with locality-based scheduling. Given that the allocation heap is a shared resource, allocation and deallocation must be serialized to maintain a consistent state. This serialization is either explicitly done by the memory management library or by the threading library, for instance, pthreads. The advantage of using a shared heap is the better utilization of memory resources under allocation imbalance.

In this work, we aim to achieve multiple objectives. First, we wish to improve the locality through a better memory allocation strategy. Second, we wish to reduce the serialization associated with the allocation and de-allocation events. To achieve that, we introduce a new memory management system, depicted in Figure 7. The new system creates a separate memory pool for each thread. As allocations are performed using a common object, we resolve the mapping between threads and the allocators using an identifier stored in the thread private data. Thus, concurrent allocations by workers do not involve any serialization. The master thread has also a separate memory resource pool which is typically much larger than those assigned to threads. The system configurations are discovered dynamically at runtime.

The introduced memory management system is built on top of our modified version of mmalloc library [6] in order to provide multiple separate heaps using the functionality provided by libnuma [12] library to manage NUMA-aware

allocations. The memory manager controls the association between threads, the allocator instances, NUMA policy, and garbage collection mechanisms. Some allocators, such as TC-Malloc [8] reduces the overhead of serialization for threaded applications using cached allocations. This allocator targets limited object sizes (typically small) that fits in a small cache, otherwise it uses the central allocator. It does not also provide control over NUMA placement.

In Libtensor, an object allocated by a thread can be used or deallocated by another worker thread or by the master thread. Therefore, while freeing objects, we cannot use the thread *id* to guide the selection of the allocator. Instead we use the affinity of the object to select the allocator. The challenge with this approach is that to have a consistent state we need to serialize the access by using a lock as one thread may be allocating while another is deallocating. To avoid such serialization, we created a simple garbage collection mechanism in which one thread, trying to free an object not allocated by its allocator, places the deallocated object in a recycle buffer for the allocator thread. We allocate recycle buffers such that each pair of threads has a unique one, thus no serialization is involved during deallocation.

The other objective of having these separate allocation pools is to control the NUMA locality. To achieve that, we restrict the migration of worker threads to cores attached to a single NUMA domain. Then, we control the allocation policy to favor the physical memory attached to the thread NUMA domain. Managing NUMA locality while load-balancing workload is typically achieved either through migrating data to workers [3] or assigning workers based on their proximity to data [2]. For Libtensor, a large fraction of the working set is created after task assignment making traditional techniques not directly applicable.

The master thread in our implementation receives a special treatment. Its migration is not bound to a single NUMA domain. Additionally, its memory pool is spread across multiple NUMA domains in a round-robin fashion. The motivation of this choice is that any thread can access objects allocated by the master, thus we cannot optimally reason about the locality. Instead we can maximize performance by balancing the load on the memory controllers. The data allocated by the master typically persist during the run and are less likely to be deallocated. The placement of the master thread depends on the availability of cores. Although this mechanism does not guarantee optimal locality placement, we will show in the next sections its effectiveness in improving the performance. It also does not involve sophisticated analysis at runtime.

B. Space Exploration for Performance Tuning Parameters

As discussed earlier, it is difficult to statically reason about the performance impact of alternative parameterizations or implementations given the dynamic nature of the workload.

We decided to tackle this challenge using space exploration in which we study the impact of changing multiple tuning parameters on performance. For the purpose of this study, we focus on two parameters. The first is the tiling factor, which is

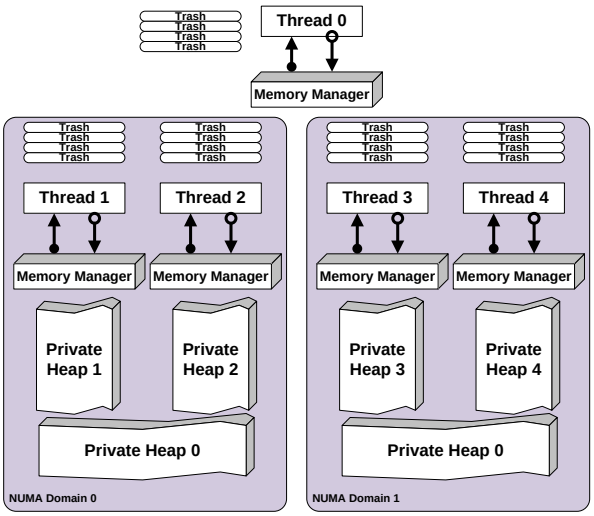


Fig. 7. New memory management system that handles NUMA locality and reduce contention while acquiring or releasing the memory resources.

expressed as a scaling applied to a base block size of four. As the default tile size in Libtensor is 16, the base tile factor is 4. Tiling can be applied to multiple dimensions including virtual orbit, occupied orbits, etc. We have chosen the exploration of different tiling factors for the virtual orbits because they are typically associated with the largest base function, making them the dataset size dominant. Additionally, tensor operations involving virtual orbits dominate the execution time. Nevertheless, the tiling exploration could be performed for other dimensions as well.

We decide also to explore the scheduling quanta, which is used by the worker threads to move tasks en masse from the main queue to their thread local queues. The default quantum in the base implementation is four. This base value is considered to reduce the lock contention during scheduling without severely impacting the load balance.

VII. Performance Results and Analysis

In this section, we show the performance with the base *vs*. the new memory management systems with space exploration of tiling and scheduling parameters.

A. Space Exploration

In Figure 8, we present the performance improvement relative to the tuned baseline implementation for different scheduling quanta and tiling sizes on Trestles system using the original (top) and new (bottom) memory management schemes. The default tasking and tiling factors are both 4.

The first trend we observe is that increasing the tile size generally improves performance when we have a small scheduling quantum (y-axis =1). The reason is that the scheduling overhead is reduced with the fewer scheduling instances. Large tile size also reduces the level of parallelism, which will be quantified later. Combining large scheduling quanta with large tile size leads to a irrecoverable load imbalance thus degrading the performance degrades on the right-bottom corners of the figures. Additionally, larger tiles lead to more computation and

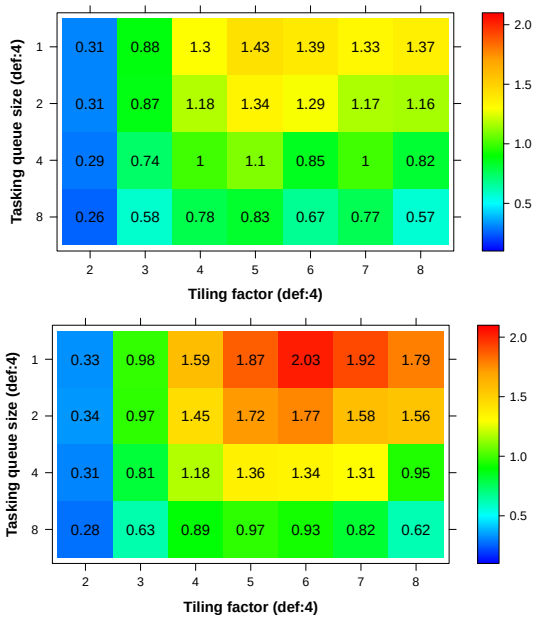


Fig. 8. Performance improvement with space exploration using multiple tiling factors and scheduling quanta (P1 on Trestles using 32 threads) using either the original (top) or new (bottom) memory management schemes

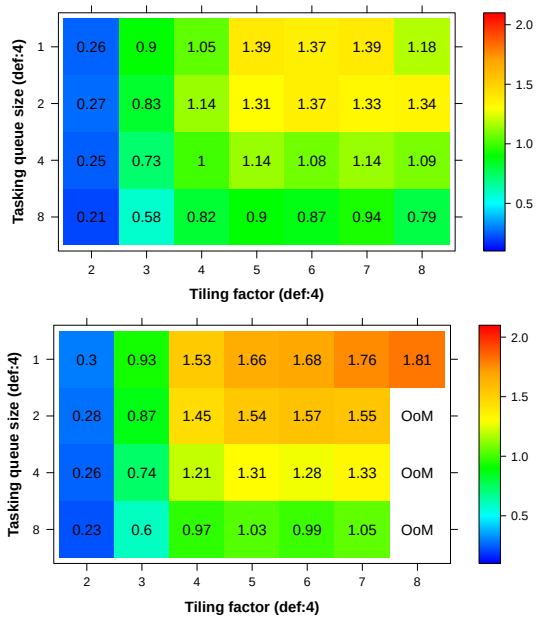
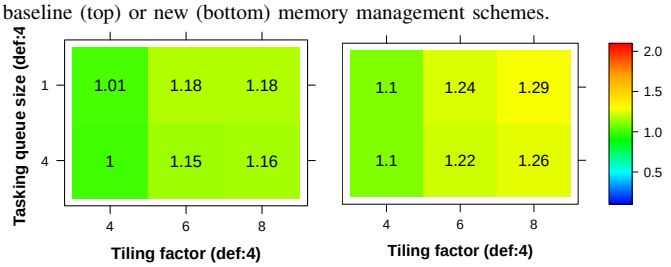


Fig. 9. Performance improvement with space exploration using multiple tiling factors and scheduling quanta (P1 on Edison using 24 threads) for either the baseline (top) or new (bottom) memory management schemes.



(a) Base memory management (b) New memory management

Fig. 10. Performance improvement with space exploration using multiple tiling factors and scheduling quanta (P3 on Carver using 32 threads).

larger dataset. Therefore, these configurations further stress the memory and floating-point units. The second observation is that the performance improvement for the right side of the figure (subfigure b—the new memory system), outperforms the base system. The performance difference ranges between 20% to 60%. Overall, we achieve by combining space exploration with the new memory management system more than 2× improvement.

In Figure 9, we show the same problem running on an Edison node. While, we generally see similar trends to those on Trestles, we see a couple of distinct behaviors. First, the improvement is more monotonic with the new memory system. As such, the top-right corner is delivering the best performance. Second, combining large tile-size with large scheduling quantum leads to an imbalance that leads to exhausting the local allocator in which one runs out of memory (OoM). Please note that threads are allocated equal amount of memory and we do not implement any heap stealing or dynamic resizing at runtime. The monotonic improvement on Edison attests for its powerful compute node, its ability to handle higher flop rates, and its ability to sustain more memory traffic. Please note the absolute performance for Edison is about 1.8× better than Trestles.

The optimal configuration for performance differs from one machine to the other. On Trestles a tiling factor of 6 and smallest scheduling quanta delivers the best performance, while on Edison the tile factor has to be 8. This suggests a need for auto-tuning if we wish to relieve the user from the burden of manually searching for the best configuration.

Edison and Trestles stress two different aspects of the new memory systems. Trestles has a strong NUMA locality (8 NUMA domains and a more than 2× difference in access latency between the domains). Edison has only 2 NUMA domains, with much smaller sensitivity to NUMA locality. Edison benefits more from reducing the serialization in the allocation and deallocation processes, while Trestles benefits more from the better NUMA management.

Figure 10 shows the performance across the tuning space for a relatively large problem with a dataset of several hundred GBs. We ran this problem on a special compute node with 1TB physical memory (Carver). The first observation is that, while we observe similar performance improvement trends, but the improvement is smaller, at most 29%. We attribute the difference between the improvement on Edison and Carver to the computational power of the cores. The higher the performance, the more likely to be impacted by the serialization events. Carver uses an older generation of SSE-based Intel Nehalem-EX processors, while Edison uses modern AVX-based Intel Ivy-bridge processors. The large memory node also has a much slower memory system. The Ivy bridge has a larger L3 cache (30MB per chip compared with 18MB for the Nehalem-EX). The larger dataset is not a big factor in the performance difference because the task size is more influenced by the tile size than the whole dataset. We conclude that the importance of having a distributed allocator increases with the improvement in processor performance.

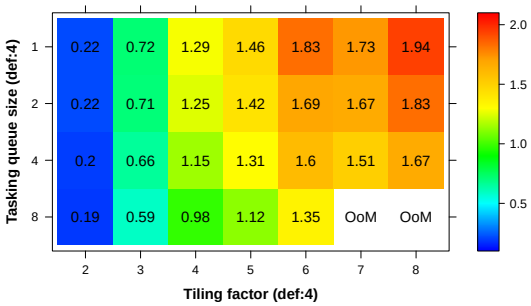


Fig. 11. Performance improvement with space exploration using multiple tiling factors and scheduling quanta for the new memory management system (P2 on Edison using 24 threads).

In Figure 11, we present the space exploration for P2 on Edison. This problem also achieves significant performance improvements, up to 1.94×. These results suggest that the gains arising from our optimizations and tuning are independent of the problem. The improvement trends are consistent across multiple problems.

B. Execution Time Decomposition

In Figure 12, we show the execution time decomposition for the master and the worker threads for different run configuration. The first set of configurations correspond a vertical cut at Figure 9 bottom at a tile size of six. The second set of configurations corresponds to a horizontal cut at a scheduling quantum of one. For the master we split the execution time between task creation and wait for the worker. For the worker the execution time is split between BLAS computation (using Intel MKL implementation [11]), queue management, and load imbalance.

The fixed-tile configuration (top) shows that increasing the scheduling quantum increases the load imbalance. The imbalance almost doubles the worker execution time. The fixed-schedule configuration (bottom) shows that the scheduling overhead (worker other) increases as we decrease the tile size. The load imbalance increases by having a small tile size. As the overheads decrease, we start exposing some of the algorithmic limitations of the Libtensor framework. For instance, the master overhead for creating tasks starts becoming a scaling bottleneck. In the current implementation, the master does not overlap the task creation with workers executing them. Additionally, the tasking library does not handle dependencies, thus one cannot enqueue tasks from multiple dependent computational steps and have the tasks execute in a correct order.

C. Parallelism and Memory Requirements

Figure 13 shows the frequency of allocation and deallocation during the execution of the P1 problem by the master and the worker threads. Despite the fact that this problem executes in relatively short period of time (12-30 seconds), we observe that the allocation events have high frequency. For instance for a tiling factor of 4, the application generates 468K instances of allocations and deallocation events with more than

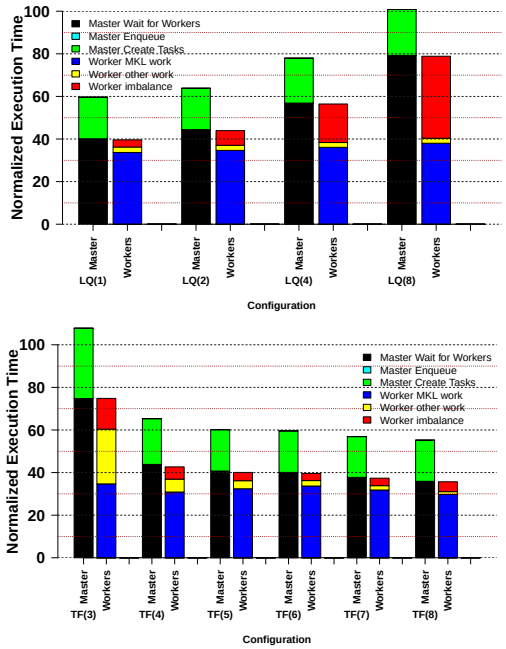


Fig. 12. Decomposition of execution time for the master and the worker threads. top: tile size is fixed at 6, but the scheduling quanta is varied. Bottom: scheduling quanta is fixed at 1, but the tiling is varied.

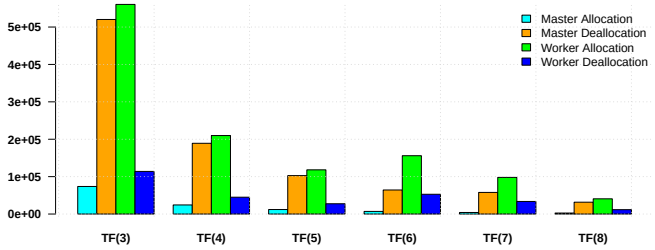


Fig. 13. The impact of tiling on allocation and deallocation by the master and the worker threads.

90% occurring dynamically during the computation. This data shows that any serialization, for instance due to the use of a shared heap, can significantly impact the performance.

The increase in the tiling size reduces the frequency of allocations. For instance, increasing the tiling factor from 4 to 8 reduces the frequency of allocations by 5.6×. This suggests that increasing the tiling factor is beneficial for reducing the serialization, with the use of a centralized allocator.

The other observation is that the worker threads execute most of the allocations, and the master executes most of the deallocations. Typically, threads deallocate less than one third of their allocations. Therefore, a garbage collection mechanism is needed to break any dependency.

While these results suggest that increasing tiling factor benefits performance, there are performance limiting factors associated with increasing the tile size. In Figure 14, we show that impact of increasing the tiling size on the volume of allocation by noting the high watermark (memory allocated) on Edison. For the worker threads, we aggregate values across all workers. The increase in tiling factor that typically improves the performance leads to an increase in

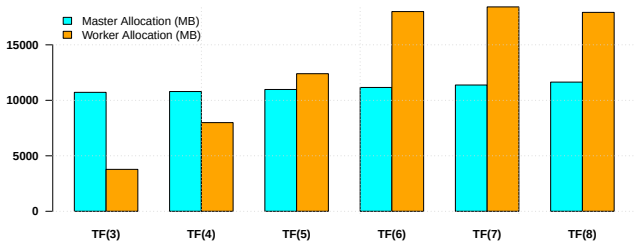


Fig. 14. The impact of tiling on the highest watermark for allocation by the master and workers for P1 (measured on Edison).

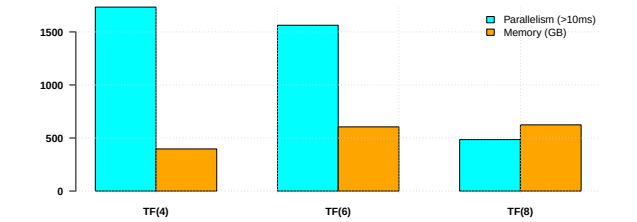


Fig. 15. The effect of tiling size on level of parallelism (for tasks needing more than 10ms execution time) and the memory requirements (in GB) for P3 (measured on Carver).

memory usage. Doubling the tile size from 4 to 8 increases the memory requirement of the workers by 2.2×. We note that the master allocation requirement is fixed for all tiling factors. These results show that the performance improvement from tuning the tiling size is conditional upon fitting the dataset within the physical memory. If we cause the memory requirements to exceed the physical memory, an out-of-core memory management will severely impact the performance. Consequently, improving the performance for different tiling sizes is of critical importance because physical memory could dictate the operating point.

Figure 15 highlights that impact of increasing the tiling size on the level of parallelism in addition to the corresponding memory requirements. We consider a problem with large dataset, P3. We focused only on coarse-grained tasks (needing at least 10ms for execution). Fine-grained tasks do not typically improve the scaling across many cores because of the overhead of scheduling. As we increase the tiling factor from 4 to 8, the memory requirements increase by 1.6× . The alarming trend is the decrease in the degree of parallelism by 3.6×. Obviously, this is due to combining smaller tasks into larger ones. This decrease in parallelism can pose a challenge if the number of cores increases. It also makes load-balancing more challenging. The increase in core count may force the library to deal with small tile-size. In such cases, reducing the scheduling overhead and removing any serialization become of crucial importance.

D. Strong Scaling

Figure 16 shows the scaling behavior for two problems P1 & P2. In these figures we show the performance of the base implementation, the performance with task stealing (tuned), the performance with our new memory management system (with the best of space exploration). We additionally show a theoretical limit based on linear scaling of the base implementation, and the the theoretical limit based on the master serial time (Amdahl's Limit). The performance improves for all concurrency levels compared with the tuned version by up to 2×. For P1, we see that we achieve the best possible performance allowed by Amdahl's law. To improve the performance further, we need to change the tasking library such that tasks from dependent computations are put in the master queue concurrently. This put an additional requirement of handling dependency between tasks at runtime. The reason this has not been explored so far is the large count of visits to the small tasks, shown in Figure 4. Managing dependency at runtime is likely to increase the scheduling overhead across all task sizes.

The data for P2 shows what appears to be an anomaly with respect to scaling. The problem exhibits a superlinear speedup. The reason for such behavior is that the problem exhibits a speedup over the base scheme even for the serial execution (by about 20%). This speedup is due to the distribution of the dataset across multiple NUMA domains, which improves the bandwidth to the memory by using multiple memory controllers. The problem also benefits from the increased cache size from the second chip as worker threads get distributed across both sockets. Consequently, the performance exceeds the linear scaling of the base. We noticed the same behavior on Trestles systems for this problem. The performance of this problem is sublinear with respect to the serial run with our optimizations and space search. We explored the distribution of the dataset for multiple NUMA domains for the P1 problem with single thread, but the performance degrades. Having the need to have a common way to compare the scaling, we have chosen the base implementation as a reference.

Ultimately, to attain linear speedups, a major redesign is necessary. First, we need to incorporate task dependencies into the task scheduling. We need to execute bookkeeping operations done by the master threads more efficiently. The viability of such a design shift is still under-investigation mainly because of the large percentage of small tasks involved in the computations.

E. Complexity Hiding and Tuning Space

So far, we showed that performance improvement can be obtained using a better memory management and space exploration. While the new memory management proves to provide advantage to all configurations, the space exploration can be tricky due to the possibility to exhaust the physical memory. To hide this complexity from the user, the Libtensor library needs to embody this information into the library. This information can be stored as tuning experiences, analogous to FFTW wisdoms [7]. This feature can be considered in future releases of Libtensor.

VIII. Conclusions

In this paper, we present our efforts to analyze and tune the Libtensor framework for quantum chemistry computations. Our analysis shows that load-balancing the computation is challenged by the large variability in task granularity and dominance of small tasks. We show that memory management is

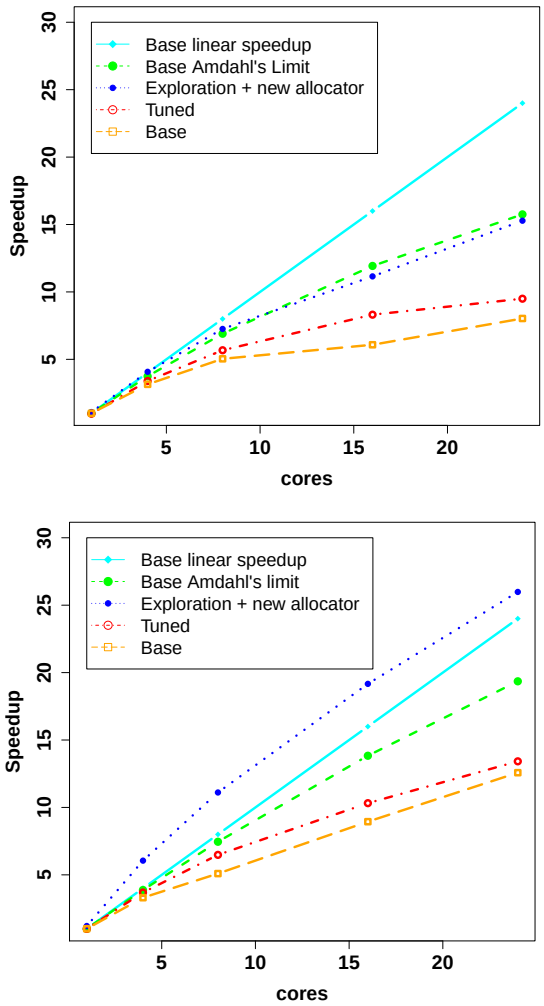


Fig. 16. Strong scaling with the number of cores for the P1 (top) & P2 (bottom) problems on Edison with linear extrapolations and Amdahl's limits.

also a major bottleneck for scaling because of the serialization associated with the frequent dynamic allocations. We devise a new memory management system that provides NUMA-aware allocations. This system breaks the serialization between threads during allocation through thread-specific heaps and during deallocation using a garbage collector. We show also that space exploration is needed to find optimal tiling and scheduling decisions. Future work will target automating this process. Overall, the studied benchmarks observed more than $2\times$ improvement over the base implementation.

Acknowledgments

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485, 1967.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, Feb. 2011.

[3] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. Forestgomp: An efficient OpenMP environment for numa architectures. *International Journal of Parallel Programming*, 38(5-6):418–439, 2010.

[4] J. Calvin. TiledArray, a massively-parallel, block-sparse tensor library written in C++. https://github.com/ValeevGroup/tiledarray. (Accessed on May 30, 2014).

[5] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. Krylov. New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations. *J. Comput. Chem.*, 34:2293–2309, 2013.

[6] F. Fish, C. Support, and M. Haertel. The gnu memory-mapped malloc package. http://www.math.utah.edu/docs/info/mmalloc_toc.html.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):216–231, 2005.

[8] S. Ghemawat and P. Menage. Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[9] T. Helgaker, P. Jørgensen, and J. Olsen. *Molecular electronic structure theory*. Wiley & Sons, 2000.

[10] S. Hirata. Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J. Phys. Chem. A*, 107(46):9887–9897, 2003.

[11] Intel. Math kernel library. http://developer.intel.com/software/products/mkl/.

[12] A. Kleen. A NUMA API for Linux. http://halobates.de/numaapi3.pdf.

[13] M. Kállay and P. Surján. Higher excitations in coupled-cluster theory. *J. Chem. Phys.*, 115:2945, 2001.

[14] A. Krylov and P. Gill. Q-Chem: An engine for innovation. *WIREs Comput. Mol. Sci.*, 3:317–326, 2013.

[15] V. Lotrich, N. Flocke, M. Ponton, B. A. Sanders, E. Deumens, R. J. Bartlett, and A. Perera. An infrastructure for scalable and portable parallel programs for computational chemistry. *International Conference of Supercomputing (ICS)*, pages 523–524, 2009.

[16] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops Tensor Framework: Reducing communication and eliminating load imbalance in massively parallel contractions. *The IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 813–824, 2013.

[17] J. Stanton, J. Gauss, M. Harding, P. Szalay, A. Auer, R. Bartlett, U. Benedikt, C. Berger, D. Bernholdt, Y. Bomble, O. Christiansen, M. Heckert, O. Heun, C. Huber, T.-C. Jagau, D. Jonsson, J. Juslius, K. Klein, W. Lauderdale, D. Matthews, T. Metzroth, D. O'Neill, D. Price, E. Prochnow, K. Ruud, F. Schiffmann, S. Stopkowicz, M. Varner, J. Vzquez, F. Wang, and J. Watts. in CFOUR, Coupled Cluster techniques for Computational Chemistry, a quantum-chemical program package (www.cfour.de).

[18] H.-J. Werner and P. J. Knowles. Molpro quantum chemistry package. http://http://www.molpro.net.