Accelerating NWChem Coupled Cluster Through Dataflow-Based Execution

Heike Jagode[1(✉)], Anthony Danalis[1], George Bosilca[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, USA
jagode@icl.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, USA
[3] University of Manchester, Manchester, UK

Abstract. Numerical techniques used for describing many-body systems, such as the Coupled Cluster methods (CC) of the quantum chemistry package NWChem, are of extreme interest to the computational chemistry community in fields such as catalytic reactions, solar energy, and bio-mass conversion. In spite of their importance, many of these computationally intensive algorithms have traditionally been thought of in a fairly linear fashion, or are parallelised in coarse chunks.

In this paper, we present our effort of converting the NWChem's CC code into a dataflow-based form that is capable of utilizing the task scheduling system PaRSEC (Parallel Runtime Scheduling and Execution Controller) – a software package designed to enable high performance computing at scale. We discuss the modularity of our approach and explain how the PaRSEC-enabled dataflow version of the subroutines seamlessly integrate into the NWChem codebase. Furthermore, we argue how the CC algorithms can be easily decomposed into finer grained tasks (compared to the original version of NWChem); and how data distribution and load balancing are decoupled and can be tuned independently. We demonstrate performance acceleration by more than a factor of two in the execution of the entire CC component of NWChem, concluding that the utilization of dataflow-based execution for CC methods enables more efficient and scalable computation.

Keywords: PaRSEC · Tasks · Dataflow · DAG · PTG · NWChem · CCSD

1 Introduction

Simulating non-trivial physical systems in the field of Computational Chemistry imposes such high demands on the performance of software and hardware, that it comprises one of the driving forces of high performance computing. In particular, many-body methods, such as Coupled Cluster [1] (CC) of the quantum chemistry package NWChem [15], come with a significant computational cost, which stresses the importance of the scalability of nwchem in the context of real science.

R. Wyrzykowski et al. (Eds.): PPAM 2015, Part I, LNCS 9573, pp. 366–376, 2016.
DOI: 10.1007/978-3-319-32149-3_35

On the software side, the complexity of these software packages – with diverse code hierarchies, and millions of lines of code in a variety of programming languages – represents a central obstacle for long-term sustainability in the rapidly changing landscape of high-performance computing. On the hardware side, despite the need for high performance, harnessing large fractions of the processing power of modern large scale computing platforms has become increasingly difficult over the past couple of decades. This is due both to the increasing scale and the increasing complexity and heterogeneity of modern (and projected future) platforms. This paper is centered around code modernization, focusing on adapting the existing NWChem CC methods to a dataflow-based approach by utilizing the task scheduling system PaRSEC. We argue that dataflow-driven task-based programming models, in contrast to the control flow model of coarse grain parallelism, are a more sustainable way to achieve computation at scale.

The Parallel Runtime Scheduling and Execution Control (PaRSEC) [2] framework is a task-based dataflow-driven runtime that enables task execution based on holistic conditions, leading to a better computational resources occupancy. PaRSEC enables task-based applications to execute on distributed memory heterogeneous machines, and provides sophisticated communication and task scheduling engines that hide the hardware complexity from the application developer. The main difference between PaRSEC and other task-based engines lies in the way tasks, and their data dependencies, are represented. PaRSEC employs a unique, symbolic description of algorithms allowing for innovative ways of discovering and processing the graph of tasks. Namely, PaRSEC uses an extension of the symbolic Parameterized Task Graph (PTG) [3,4] to represent the tasks and their data dependencies to other tasks. The PTG is a problem-size-independent representation that allows for immediate inspection of a task's neighborhood, regardless of the location of the task in the Directed Acyclic Graph (DAG). This contrasts all other task scheduling systems, which discover the tasks and their dependencies at run-time (through the execution of skeleton programs) and therefore cannot process a future task that has not yet been discovered, or face large overheads due to storing and traversing the DAG that represents the whole execution of the parallel application.

In this paper, we describe the transformations of the NWChem CC code to a dataflow version that is executed over PaRSEC. Specifically, we discuss our effort of breaking down the computation of the CC methods into fine-grained tasks with explicitly defined data dependencies, so that the serialization imposed by the traditional linear algorithms can be eliminated, allowing the overall computation to scale to much larger computational resources.

Despite having in-house expertise in PaRSEC, and working closely and deliberately with computational chemists, this code conversion proved to be laborious. Still, the outcome of our effort of exploiting finer granularity and parallelism with runtime/dataflow scheduling is twofold. First, it successfully demonstrates the feasibility of converting TCE generated code into a form that can execute in a dataflow-based task scheduling environment. Second, it demonstrated that utilizing dataflow-based execution for CC methods enables more efficient and scalable computations. We present a thorough performance evaluation and demonstrate that the modified CC component of NWChem outperforms the original by more than a factor of two.

2 Implementation of Coupled Cluster Theory

The Coupled Cluster theory is considered by many to be the gold standard for accurate quantum-mechanical description of ground and excited states of chemical systems. Its accuracy, however, comes at a significant computational cost. An important role in designing the optimum memory vs. cost strategies in Coupled Cluster implementations is played by the automatic code generator, the Tensor Contraction Engine (TCE) [6]. In the first subsection, we highlight the basics necessary to understand the original parallel implementation of CC through TCE. We then describe our design decisions of the dataflow version of the CC code.

2.1 Coupled Cluster Theory Through TCE

Especially important in the hierarchy of the CC formalism is the iterative CC model with Single and Double excitations (CCSD) [13], which is the base for many accurate perturbative CC formalisms. Our starting point for the investigation in this paper is the CCSD version that takes advantage of the alternative task scheduling, and the details of these implementations have been described in [7].

In NWChem, the CCSD code (among other kernels) is generated through the TCE into multiple sub-kernels that are divided into so-called "T1" and "T2" subroutines for equations that determine the `T1` and `T2` amplitude matrices. These amplitude matrices embody the number of excitations in the wave function, where `T1` represents all single excitations and `T2` represents all double excitations. The underlying equations of these theories are all expressed as contractions of many-dimensional arrays or tensors (generalized matrix multiplications). There are typically many thousands of such terms in any one problem, but their regularity makes it relatively straightforward to translate them into FORTRAN code – parallelized with the use of *Global Arrays* (GA) [11] – through the TCE.

Structure of the CCSD Approach. For the iterative CCSD code, there exist 19 `T1` and 41 `T2` subroutines, and all of them highlight very similar code structure and patterns. Figure 1 shows the pseudocode FORTRAN code for one of the generated `T1` and `T2` subroutines, highlighting that most work is in deep loop nests. These loop nests consist of three types of code:

– Local memory management (i.e., `MA_PUSH_GET()`, `MA_POP_STACK()`),
– Calls to functions (i.e., `GET_HASH_BLOCK()`, `ADD_HASH_BLOCK()`) that transfer data over the network via the GA layer,

```
my_next_task = SharedCounter()
DO h7b = 1,noab
  DO p3b = noab+1,noab+nvab
    IF (int_mb(k_spin+h7b).eq.int_mb(...)) THEN
      call MA_PUSH_GET(f(p3b,h7b),..., k_c)

      DO p5b = noab+1,noab+nvab
        DO h6b = 1,noab
          call GET_HASH_BLOCK(dbl_mb(k_b),...,f(p3b,p5b,h7b,h6b))
          call TCE_SORT_4( dbl_mb(k_b),...,f(p3b,p5b,h7b,h6b))
          ...
          call DGEMM( ..., f(p3b,p5b,h7b,h6b))
        END DO
      END DO

      call ADD_HASH_BLOCK(dbl_mb(k_c), ...)
      my_next_task = SharedCounter()
    END IF
  END DO
END DO
```

Fig. 1. Pseudocode of one CCSD subroutine as generated by the TCE.

– Calls to the subroutines that perform the actual computation on the data `GEMM()` and `SORT()` (which performs an $O(n)$ remapping of the data, rather than an $O(n * log(n))$ sorting).

The control flow of the loops is parameterized, but static. That is, the induction variable of a loop with a header such as "`DO p3b = noab+1,noab+nvab`" (i.e., `p3b`) may take different values between different executions of the code, but during a single execution of CCSD the values of the parameters `noab` and `nvab` will not vary; therefore every time this loop executes it will perform the same number of steps, and the induction variable `p3b` will take the same set of values. This enables us to restructure the body of the inner loop into tasks that can be executed by PaRSEC. That is, tasks with an execution space that is parameterized (by `noab`, `nvab`, etc.), but constant during execution.

Parallelization of CCSD. Parallelism of the TCE generated CC code follows a coarse task-stealing model. The work inside each `T1` and `T2` subroutine is grouped into chains of multiple matrix-multiply kernels (`GEMM`). The `GEMM` operations within each chain are executed serially, but different chains are executed in a parallel fashion. However, the work is divided into levels. More precisely, the 19 `T1` subroutines are divided into 3 different levels and the execution of the 41 `T2` subroutines is divided into 4 different levels. The task-stealing model applies only within each level, and there is an explicit synchronization step between the levels. Therefore the number of chains that are available for parallel execution at any time is a subset of the total number of chains.

Load balancing within each of the seven levels of subroutines is achieved through shared variables (exemplified in Fig. 1 through `SharedCounter()`) that are atomically updated (read-modify-write) using GA operations. The use of shared variables, that are atomically updated is bound to become inefficient at large scale, becoming a bottleneck and causing major overhead.

Also, the notion of *task* in the current CC implementation of NWChem and the notion of *task* in PaRSEC are not identical. As discussed before, in NWChem, a *task* is a whole chain of `GEMMs`, executed serially, one after the other. In our PaRSEC implementation of CC, each individual `GEMM` kernel is a task on its own, and the choice between executing them as a chain, or as a reduction tree, is almost as simple as flipping a switch. *In summary, the most significant impact of porting CC over* PaRSEC *is the ability to eliminate redundant synchronizations between the levels and to break down the algorithms into finer grained tasks with explicitly defined dependencies.*

2.2 Coupled Cluster Theory over PaRSEC

PaRSEC provides a front-end compiler for converting canonical serial codes into the PTG representation. However, due to computability limits, this tool is limited to polyhedral codes, i.e., loops, branches, and array indexes that only depend on affine functions of the loop induction variables, constant variables, and numeric literals. The CC code generated by TCE is neither organized in pure tasks – i.e., functions with no side-effects to any memory other than arguments passed to the function itself – nor is the control flow affine. For example, branches such as "`IF(int_mb(k_spin+h7b-1)...)`" (see Fig. 1) are very common. Such branches make the code not only non-affine, but statically undecidable since their outcome depends on program data, and thus it cannot be resolved at compile time.

While the behavior of the CC code depends on program data, this data is constant during a given execution of the code. Therefore, the code can be expressed as a parameterized DAG, by using lookups into the program data, either directly or indirectly. In our implementation we access the program data indirectly by builting meta-data structures in a preliminary step. The details of this "first step" are described later in this section.

In the work described in this paper, we implemented a dataflow form for all functions of the CCSD computation that are associated with calculating parts of the `T2` amplitudes, particularly the ones that perform a `GEMM` operation (the most time consuming parts). More precisely, we converted a total of 29[1] of the 41 `T2` subroutines – which we refer to under the unified moniker of "`GA:T2`" for the original version, and "`PaRSEC:T2`" for the dataflow version of the subroutines.

Design Decisions. The original code of our chosen subroutines consists of deep loop nests that contain the memory access routines as well as the main computation, namely `SORT` and `GEMM`. In addition to the loops, the code contains several `IF` statements, such as the one mentioned above. When CC executes, the

[1] All subroutines with prefix "icsd_t2_" and suffices: 2_2_2_2(), 2_2_3(), 2_4_2(), 2_5_2(), 2_6(), lt2_3x(), 4_2_2(), 4_3(), 4_4(), 5_2(), 5_3(), 6_2_2(), 6_3(), 7_2(), 7_3(), vt1ic_1_2(), 8(), 2_2_2(), 2_4(), 2_5(), 4_2(), 5(), 6_2(), vt1ic_1, 7(), 2_2(), 4(), 6(), 2().

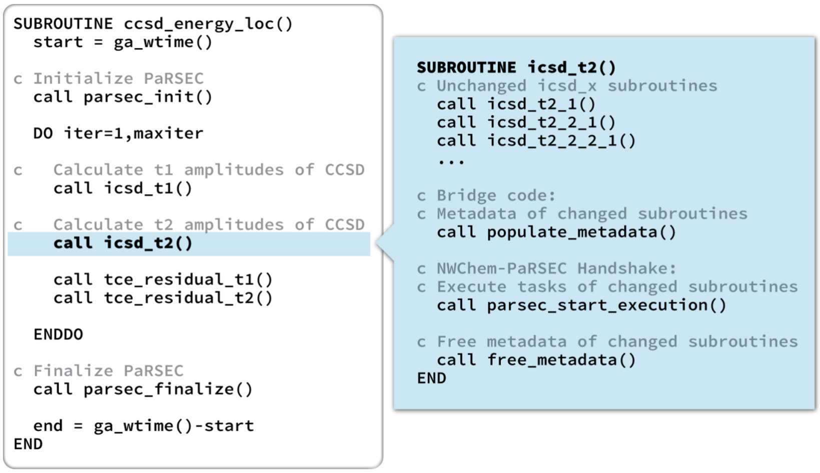
code goes through the entire execution space of the loop nests, and only executes the actual computation kernels (`SORT` and `GEMM`) if the multiple `IF` branches evaluate to `true`. To create the PaRSEC-enabled version of the subroutines (`PaRSEC:T2`), we decomposed the code into two steps:

The first step traverses the execution space and evaluates all `IF` statements, without executing the actual computation kernels (`SORT` and `GEMM`). This step uncovers sparsity information by examining the program data (i.e., `int_mb(k_spin+h7b-1)`) that is involved in the `IF` branches, and stores the results in custom meta-data vectors that we defined.
The custom meta-data vectors merely hold information regarding the actual loop iterations that will execute the computational kernels at run-time, i.e., iterations where all the `IF` statements evaluate to `true`. This step significantly reduces the execution space of the loop nests by eliminating all entries that would not have executed. In addition, this step probes the GA library to discover where the program data resides in memory and stores these addresses into the meta-data structures as well.

The second step is the execution of the PTG representation of the subroutines. Since the control flow depends on the program data, the PTG examines our custom meta-data vectors populated by the first step; this allows the execution space of the modified subroutines over PaRSEC to match the original execution space of `GA:T2`. Also, using the meta-data structures, PaRSEC accesses the program data directly from memory, without using GA.

Parallelization and Optimization. One of the main reasons we are porting CC over PaRSEC is the ability of the latter to express tasks and their dependencies at a finer granularity, as well as the decoupling of work tasks and communication operations that enables us to experiment with more advanced communication patterns than serial chains. Since matrix addition is an associative and commutative operation, the order in which the `GEMMs` are performed does not bear great significance as long as the results are atomically added. This enables us to perform all `GEMM` operations in parallel and sum the results using a binary reduction tree. Clearly, in this implementation there are significantly fewer sequential steps than in the original chain [10]. In addition, the sequential steps are matrix additions, not `GEMM` operations, so they are significantly faster, especially for larger matrices. Reductions only apply to `GEMM` operations that execute on the same node, thus avoiding additional communication.

The original version of the code performs an atomic accumulate-write operation (via calls to `ADD_HASH_BLOCK()`) at the end of each chain. Since our dataflow version of the code computes the `GEMMs` for each chain in parallel, we eliminate the **global** atomic GA functionality and perform direct memory access instead, using **local** atomic locks within each node to prevent race conditions.



```
SUBROUTINE ccsd_energy_loc()
  start = ga_wtime()

c Initialize PaRSEC
  call parsec_init()

  DO iter=1,maxiter

c   Calculate t1 amplitudes of CCSD
    call icsd_t1()

c   Calculate t2 amplitudes of CCSD
    call icsd_t2()

    call tce_residual_t1()
    call tce_residual_t2()

  ENDDO

c Finalize PaRSEC
  call parsec_finalize()

  end = ga_wtime()-start
END
```

```
SUBROUTINE icsd_t2()
c Unchanged icsd_x subroutines
  call icsd_t2_1()
  call icsd_t2_2_1()
  call icsd_t2_2_2_1()
  ...

c Bridge code:
c Metadata of changed subroutines
  call populate_metadata()

c NWChem-PaRSEC Handshake:
c Execute tasks of changed subroutines
  call parsec_start_execution()

c Free metadata of changed subroutines
  call free_metadata()
END
```

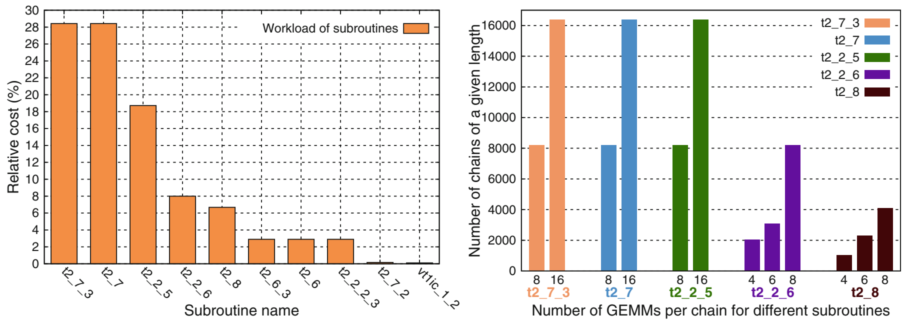
Fig. 2. High level view of PaRSEC code in NWChem.

3 Performance Evaluation

In this section we present the performance of the entire CCSD code using the dataflow version "`PaRSEC:T2`" of the 29 CC subroutines and contrast it with the performance of the original code "`GA:T2`". Figure 2 depicts a high level view of the integration of the PaRSEC-enabled code in NWChem's CCSD component. The code that we timed (see `start` and `end` timers in Fig. 2) includes all 19 `T1` and 41 `T2` subroutines as well as additional execution steps that set up the iterative CCSD computation. The only difference between the original NWChem runs and our modified version is the replacement of the 29 original `T2` subroutines "`GA:T2`" with their dataflow version "`PaRSEC:T2`" and the prerequisites discussed in Sect. 2.2; these prerequisites include: meta-data vector population, initialization, and finalization of PaRSEC. Also, in our experiments we allow for all iterations of the iterative CCSD code to reach completion.

3.1 Methodology

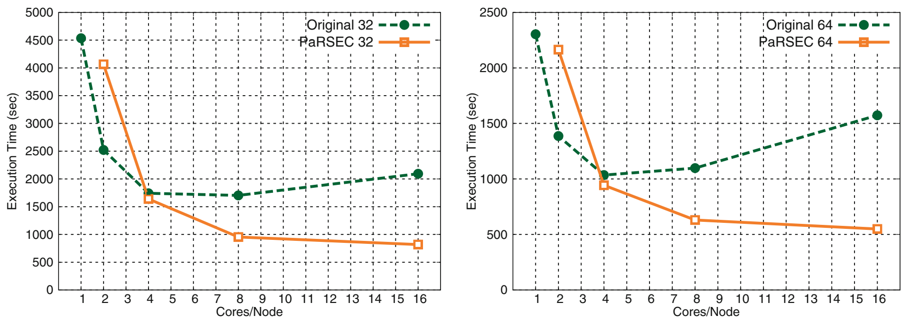
As input, we used the beta-carotene molecule ($C_{40}H_{56}$) in the 6-31G basis set, composed of 472 basis set functions. In our tests, we kept all core electrons frozen, and correlated 296 electrons. Figure 3a shows the relative workload of different subroutines (omitting those that fell under 0.1 %). To calculate this load we sum the number of floating point operations of each `GEMM` that a subroutine performs (given the sizes of the input matrices). Additionally, Fig. 3b shows the distribution of chain lengths for the five subroutines with the highest workload in the case of beta-carotene. The different colors in this figure are for readability only. As can be seen from these statistics, the subroutines that we targeted for our dataflow conversion effort comprise approx. 91 % of the execution time of all 41 `T2` subroutines in the original NWChem TCE CCSD execution.



(a) Relative load of heaviest subroutines (b) Chain Length Distribution

Fig. 3. CCSD statistics for beta-carotene and tilesize = 45.

The scalability tests for the original TCE generated code and the dataflow version of `PaRSEC:T2` were performed on the *Cascade* computer system at EMSL/PNNL. Each node has 128 GB of main memory and is a dual-socket Intel Xeon E5-2670 (Sandy Bridge EP) system with a total of 16 cores running at 2.6 GHz. We performed various performance tests utilizing 1, 2, 4, 8, and 16 cores per node. NWChem v6.5 was compiled with the Intel 14.0.3 compiler, using the optimized BLAS library MKL 11.1, provided on Cascade.



(a) Performance on 32 nodes (b) Performance on 64 nodes

Fig. 4. Execution time comparison using beta-carotene on EMSL/PNNL Cascade (Color figure online)

3.2 Discussion

Figure 4 shows the execution time of the entire CCSD kernel when the implementation found in the original NWChem code is used, and when our PaRSEC based dataflow implementation is used for the (earlier mentioned) 29 `PaRSEC:T2` subroutines. Each of the experiments were run three times; the variance between the runs, however, is so small that it is not visible in the figures. Also, the correctness of the final computed energies have been verified for each run, and differences occur only in the last digit or two (meaning, the energies match for up to the 14th decimal place). In the graph we depict the behavior of the original code using the dark (green) dashed line and the behavior of the PaRSEC implementation using the light (orange) lines. Once again, the execution time of the PaRSEC runs does not exclude any steps performed by the modified code.

On a 32 node partition, the PaRSEC version of the CCSD code performs best for 16 cores/node while the original code performs best for 8 cores/node. Comparing the two, the PaRSEC execution runs more than twice as fast – to be precise, it executes in 48 % of the best time of the original. If we ignore the PaRSEC run on 16 cores/node – in an effort to compare performance when both versions use 8 cores/node and thus have similar power consumption – we find that PaRSEC still runs 44 % faster than the original.

The results are similar on a 64 node partition: the PaRSEC version of CCSD is fastest (for 16 cores/node) with a 43 % runtime improvement compared to the original code (which on 64 nodes performs best for 4 cores/node). It is also interesting to point out that for 64 nodes, while PaRSEC manages to use an increasing number of cores – all the way up to $64 \times 16 = 1024$ cores – to improve performance, the original code exhibits a slowdown beyond 4 cores/node. This behavior is not surprising since (1) the unit of parallelism of the original code (chain of `GEMMs`) is much coarser than that of PaRSEC (single `GEMM`), and (2) the original code uses a global atomic variable for load balancing while PaRSEC distributes the work in a round robin fashion and avoids any kind of global agreement in the critical path.

4 Related Work

An alternate approach for achieving better load balancing in the TCE CC code is the Inspector-Executor methods [12]. This method applies performance model based cost estimation techniques for the computations to assign tasks to processors. This technique focuses on balancing the computational cost without taking into consideration the data locality.

ACES III [9] is another method that has been used effectively to parallelize CC codes. In this work, the CC algorithms are designed in a domain specific language called the Super Instruction Assembly Language (SIAL) [5]. This serves a similar function as the TCE, but with an even higher level of abstraction to the equations. The SIAL program, in turn, is run by a MPMD parallel virtual machine, the Super Instruction Processor (SIP). SIP has components that coordinate the work by tasks, communicate information between tasks for retrieving data, and then for execution.

The Dynamic Load-balanced Tensor Contractions framework [8] has been designed with the goal to provide dynamic task partitioning for tensor contraction expressions. Each contraction is decomposed into fine-grained units of tasks. Units from independent contractions can be executed in parallel. As in TCE, the tensors are distributed among all processes via global address space. However, since GA does not explicitly manage data redistribution, the communication pattern resulting from one-sided accesses is often irregular [14].

5 Conclusion and Future Work

We have successfully demonstrated the feasibility of converting TCE generated code into a form that can execute in a dataflow-based task scheduling environment, such as PaRSEC. Our effort substantiates that utilizing dataflow-based execution for Coupled Cluster methods enables more efficient and scalable computation – as our performance evaluation reveals a performance boost of 2x for the entire CCSD kernel.

As a next step, we will automate the conversion of the entire NWChem TCE CC implementation into a dataflow form so that it can be integrated to more software levels of NWChem with minimal human involvement. Ultimately, the generation of a dataflow version will be adopted by the TCE engine.

Acknowledgment. This material is based upon work supported in part by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476, and the DOE Office of Science, Advanced Scientific Computing Research, under award No. DE-SC0006733 "SUPER - Institute for Sustained Performance, Energy and Resilience". A portion of this research was performed using EMSL, a DOE Office of Science User Facility sponsored by the Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory.

References

1. Bartlett, R.J., Musial, M.: Coupled-cluster theory in quantum chemistry. Rev. Mod. Phys. **79**(1), 291–352 (2007)
2. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: a generic distributed DAG engine for high performance computing. Parallel Comput. **38**(12), 37–51 (2012)
3. Cosnard, M., Loi, M.: Automatic task graph generation techniques. In: Proceedings of the 28th Hawaii International Conference on System Sciences (1995)
4. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: PTG: an abstraction for unhindered parallelism. In: Proceedings of International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC) (2014)
5. Deumens, E., Lotrich, V.F., Perera, A., Ponton, M.J., Sanders, B.A., Bartlett, R.J.: Software design of ACES III with the super instruction architecture. Wiley Interdisc. Rev. Comput. Mol. Sci. **1**(6), 895–901 (2011)
6. Hirata, S.: 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)
7. Kowalski, K., Krishnamoorthy, S., Olson, R., Tipparaju, V., Aprà, E.: Scalable implementations of accurate excited-state coupled cluster theories: application of high-level methods to porphyrin-based systems. In: High Performance Computing, Networking, Storage and Analysis (SC), 2011, pp. 1–10 (2011)

8. Lai, P.W., Stock, K., Rajbhandari, S., Krishnamoorthy, S., Sadayappan, P.: A framework for load balancing of tensor contraction expressions via dynamic task partitioning. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–10. ACM (2013)
9. Lotrich, V., Flocke, N., Ponton, M., Yau, A., Perera, A., Deumens, E., Bartlett, R.: Parallel implementation of electronic structure energy, gradient and hessian calculations. J. Chem. Phys. **128**, 194104-1–194104-15 (2008)
10. McCraw, H., Danalis, A., Herault, T., Bosilca, G., Dongarra, J., Kowalski, K., Windus, T.: Utilizing dataflow-based execution for coupled cluster methods. In: Proceedings of IEEE Cluster 2014, pp. 296–297 (2014)
11. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., Apra, E.: Advances, applications and performance of the global arrays shared memory programming toolkit. Int. J. High Perform. Comput. Appl. **20**(2), 203–231 (2006)
12. Ozog, D., Shende, S., Malony, A., Hammond, J., Dinan, J., Balaji, P.: Inspector/executor load balancing algorithms for block-sparse tensor contractions. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 483–484. ACM (2013)
13. Purvis, G., Bartlett, R.: A full coupled-cluster singles and doubles model - the inclusion of disconnected triples. J. Chem. Phys. **76**(4), 1910–1918 (1982)
14. Solomonik, E., Matthews, D., Hammond, J., Demmel, J.: Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. In: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), pp. 813–824 (2013)
15. Valiev, M., Bylaska, E.J., Govind, N., Kowalski, K., Straatsma, T.P., Van Dam, H.J.J., Wang, D., Nieplocha, J., Aprà, E., Windus, T.L., de Jong, W.: NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. Comput. Phys. Commun. **181**(9), 1477–1489 (2010)