Special Issue Paper

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2018, Vol. 32(4) 540–551

DOI: 10.1177/1094342016672543
journals.sagepub.com/home/hpc
SAGE

Accelerating NWChem Coupled Cluster through dataflow-based execution

Heike Jagode[1], Anthony Danalis[1] and Jack Dongarra[1,2,3]

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 parallelized 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 with 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 (CC) (Bartlett and Musial, 2007) of the quantum chemistry package NWChem (Valiev et al., 2010), come with a significant computational cost, which stresses the importance of the scalability of NWChem in the context of real science.

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) (Bosilca et al., 2012) framework is a task-based dataflow-driven runtime that enables task

[1]University of Tennessee, Knoxville, USA
[2]Oak Ridge National Laboratory, USA
[3]University of Manchester, UK

Corresponding author:
Heike Jagode, Innovative Computing Laboratory, University of Tennessee, Department of Electrical Engineering and Computer Science, Suite 203 Claxton, 1122 Volunteer Boulevard, Knoxville, TN 37996, USA.
Email: jagode@icl.utk.edu

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) (Cosnard and Loi, 1995; Danalis et al., 2014) 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 with all other task-scheduling systems, which discover the tasks and their dependencies at runtime (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 the CC kernel into a form that can execute in a dataflow-based task-scheduling environment. Second, it confirms 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 implementation by more than a factor of two.

2. Overview of NWChem

Computational modeling has become an integral part of many research efforts in key application areas in chemical, physical, and biological sciences. NWChem is a molecular modeling software developed to take full advantage of the advanced computing systems available. NWChem provides many methods to compute the properties of molecular and periodic systems by using standard quantum-mechanical descriptions of the electronic wave function or density. The CC theory (Bartlett and Musial, 2007) is considered by many to be a gold standard for accurate quantum-mechanical description of ground and excited states of chemical systems. Its accuracy, however, comes at a significant computational cost.

2.1. Tensor Contraction Engine

An important role in designing the optimum memory versus cost strategies in CC implementations is played by the automatic code generator, the Tensor Contraction Engine (TCE) (Hirata, 2003), which abstracts and automates the time-consuming and error-prone processes of deriving the working equations of second-quantized many-electron theories and synthesizing efficient parallel computer programs on the basis of these equations. Current development is mostly focused on CC implementation which can utilize any type of single-determinantal reference function including restricted, restricted open-shell, and unrestricted Hartree–Fock determinants (RHF, ROHF, and UHF respectively) in describing closed- and open-shell molecular systems. All TCE CC implementations take advantage of Global Arrays (GA) (Nieplocha et al., 2006) functionalities, which supports the distributed memory programming model.

2.2. CC Single Double

Especially important in the hierarchy of the CC formalism is the iterative CC model with single and double excitations (CCSD) (Purvis and Bartlett, 1982), which is a starting point for many accurate perturbative CC formalisms including the ubiquitous CCSD(T) approach (Raghavachari et al., 1989). Our starting point for the investigation in this paper is the CCSD version that takes advantage of the alternative task scheduling (ATS). The details of these implementations have been described in previous publications (Kowalski et al., 2011). In summary, the original CCSD TCE implementations aggregated a large number of subroutines, which calculate either recursive intermediates or contributions to a residual vector. The dimensionalities of the tensors involved in a given subroutine greatly impact the memory, computation, and communication characteristics of each subroutine, which can lead to pronounced problems with load balancing. To address this problem and improve the scalability of the CCSD implementations, NWChem exploits the dependencies exposed between the task pools into classes characterized by a collective task pool. This was done in such a

way as to ensure sufficient parallelism in each class while minimizing the total number of such classes.

3. Implementation of CC theory

In the first subsection, we highlight the basics necessary to understand the original parallel implementation of CC through TCE. We then describe the challenges of converting TCE-generated code into a PARSEC-enabled version and its execution model that is based on the PTG abstraction.

3.1. CC theory through TCE

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 GA, through the TCE. In general, NWChem contains about one million lines of human-generated code and over two million lines of TCE-generated code.

3.1.1. 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 pseudo 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;
- calls to the subroutines that perform the actual computation on the data `DGEMM()` and `TCE_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. Specifically, tasks with an execution space that is parameterized (by `noab`, `nvab`, etc.), but constant during execution.

```
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
```

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

3.1.2. 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 three different levels and the execution of the 41 `T2` subroutines is divided into four 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 Figure 1 through `SharedCounter()`) that are atomically updated (read–modify–write) using GA operations. This is an excellent case where very good parallelism already exists but where additional parallelism can be obtained by examining the data dependencies in the memory blocks of each matrix. For example, elements of the so-called `T1` amplitude matrices can be used for further computation before all of the elements are computed. However, the current implementation of CC features a significant amount of synchronizations that prevent introducing additional levels of parallelism, which consequently limits the overall scaling on much larger computational resources.

In addition, the use of shared variables, that are atomically updated, which is currently at the heart of the task-stealing and load-balancing solution, 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.

3.2. CC 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.if a code is not affine then a static analysis is not possible at compile time. Affine codes can only contain loop headers (bounds and steps) and array indices with expressions that are limited to addition and subtraction of the induction variables, constants, and numeric literals (as well as multiplication by numeric literals) and branches (if–then–else) that contain only similar arithmetic expressions and comparison operators. These affine codes (e.g. dense linear algebra is largely affine codes) can be fully analyzed using polyhedral analysis and a compact representation of the DAG, the PTG, can be generated. A critical characteristic about the PTG is it can be used by the runtime to evaluate any part of the DAG without having to store the entire DAG in memory. This is one of the important features that differentiate PaRSEC from any other task-scheduling system.

The problem with affine codes, though, is they are a very small subset of the real-world applications. 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 (e.g. loop execution space has holes in it; branches are statically undecidable since their outcome depends on program data, and thus it cannot be resolved at compile time).

While the CC code seems polyhedral, it is not quite so. The code generated by TCE includes branches that perform array lookups into program data. For example, branches such as "`IF(int_mb(k_spin + h7b-1)...)`" (see Figure 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.

However, 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 building meta-data structures in a preliminary step. The details of this "first step" are explained in the next 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[1] 29 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.

Figure 2 illustrates the dataflow of the original 41 T2 subroutines. It shows how the work is divided into four distinct levels. The solid edges of this DAG represent the dataflow where output data (matrix C) is added to output data in another level (C->C); and the dotted edges show where output data (matrix C) is used as input data (matrix B) for different subroutines (C->B).

As mentioned above, the task-stealing model applies only within each level, and there is an explicit synchronization step between the four levels. As for the PaRSEC version of this code, our dataflow implementation of the 29 "PaRSEC:T2" subroutines is displayed in black; and the remaining 12 of the 41 T2 subroutines are displayed in light gray. They are the subroutines that do not perform a GEMM operation; and are, due to insignificance in terms of execution time, not yet converted into a dataflow form but executed as in the original code.

Our chosen subroutines comprise approximately 91% of the execution time of all 41 T2 subroutines when computing the CCSD correlation energy of the *beta-carotene* molecule ($C_{40}H_{56}$). (not including the 19 T1 subroutines and additional execution steps that set up the iterative CCSD computation). More details are discussed in Section 6.1 and illustrated in Figure 7a.

4. Dataflow version of CC

In this section, we describe our design decisions of the CC dataflow version and discuss various levels of parallelism and optimizations that have been studied for the PaRSEC-enabled CC implementation.

4.1. Design decisions

The original code of our chosen subroutines consists of deep loop nests that contain the memory access

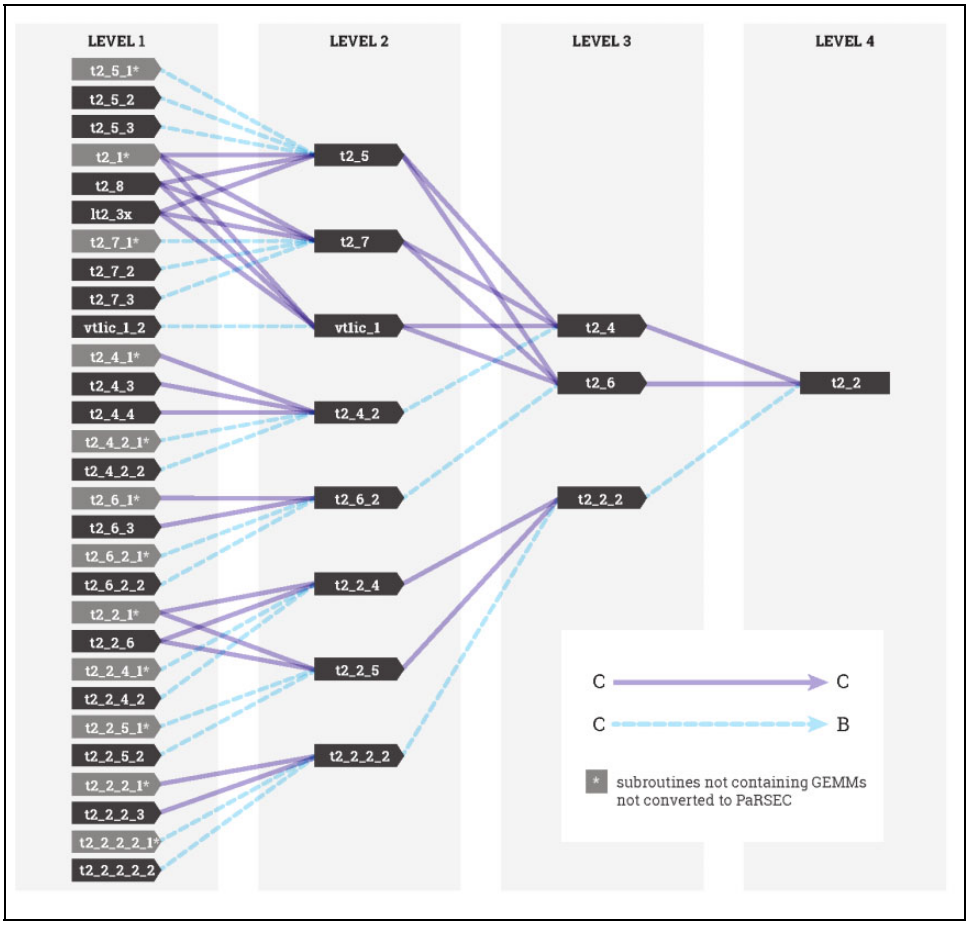

Figure 2. Directed acyclic graph (DAG) of the 41 T2 subroutines and its data dependencies.

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 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 runtime, i.e. iterations where all of 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.

4.2. 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

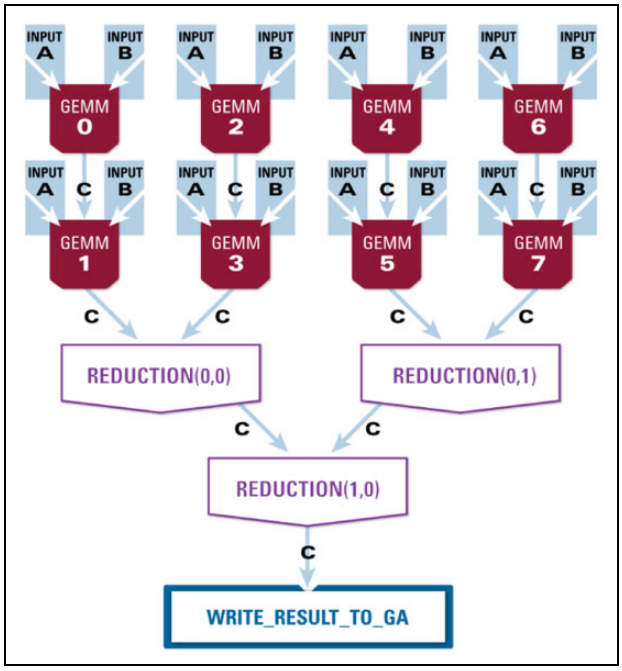

Figure 3. Parallel GEMM operations followed by reduction.

are atomically added. This enables us to perform all GEMM operations in parallel and sum the results using a binary reduction tree. Figure 3 shows the DAG of eight GEMM operations utilizing a binary tree reduction (as supposed to a serial "chain" of GEMMs). Clearly, in this implementation there are significantly fewer sequential steps than in the original chain (McCraw et al., 2014). For the sake of completeness, Figure 4 depicts such a chain where eight GEMM operations are computed sequentially.

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. The choice of our implementation, discussed in this paper, is based on earlier investigations presented in Danalis et al. (2015); Jagode et al. (2016), where we compared the performance of different variants of the modified code and explain the different behaviors that lead to the differences in performance.

4.3. Additional levels of parallelism

It is important to note that our work of converting the entire NWChem CC "FORTRAN plus Global Arrays"

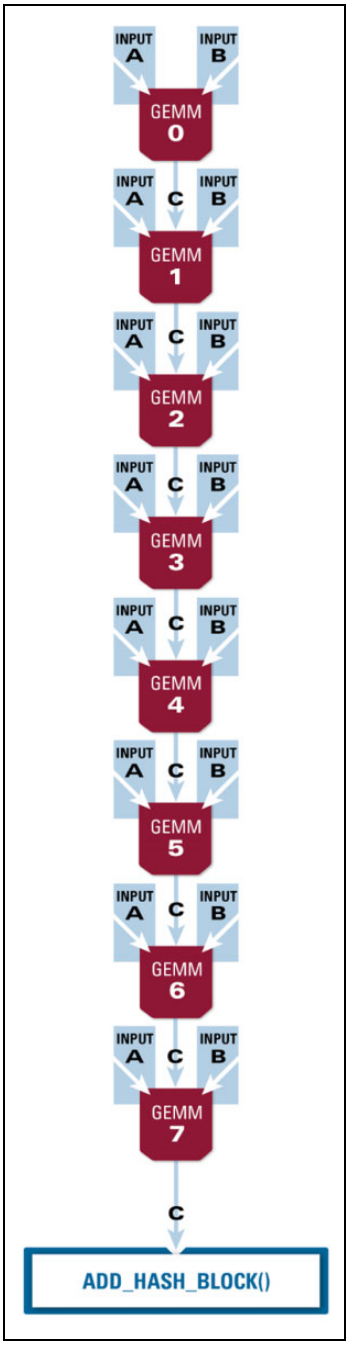

Figure 4. Chain of GEMM operations computed sequentially.

implementation into a dataflow form has been of incremental nature in order to preserve the original behavior of the code. This allowed us to initially focus only on the most time-consuming and most significant subroutines (the 29 heavy GEMM routines), and more importantly, execute them over PaRSEC without having to convert the entire CCSD module. The successful conversion of these 29 kernels has proven to be very beneficial, resulting in a performance improvement of more than a factor of two in the execution of the entire CC component of NWChem. This result justifies our

conclusion that the utilization of dataflow-based execution of CC methods enabled more efficient and scalable computation.

After completion of the dataflow implementation within each of the four levels, the next increment of work that we are currently pursuing focuses on implementing dataflow between the levels. From the DAG in Figure 2 it becomes very apparent that not every subroutine in the upper levels has to wait for the completion of all subroutines in the lower levels. For instance, `t2_2_2()` in Level 3 only depends on the data coming out of `t2_2_4()`, `t2_2_5()`, and `t2_2_2_2()` in Level 2; while these three only depend on 10 (of the 29) subroutines[2] in Level 1. Instead of putting the execution of the tasks that comprise `t2_2_2()` on hold until all subroutines in Level 1 and Level 2 are completed, the output of the tasks that flow into `t2_2_2()` can be passed as soon as it becomes available. The return of enabling dataflow between levels is twofold. First, it increases the level of parallelism even more; and second, it enabled the freedom to choose a certain placement of tasks. For example, tasks of subroutine `t2_2_2()` in Level 3 can be computed where the Level 2 subroutines, whose output flows into `t2_2_2()`, store their data.

Another advantage of enabling dataflow between the levels, in addition to the benefits resulting from increased levels of parallelism, is that part of the aforementioned work that is necessary in the current version of the code, will become unnecessary as soon as the complete CCSD code has been converted to PaRSEC. Specifically, data will not need to be pulled and pushed into the GA at the beginning and end of each subroutine if all subroutines execute over PaRSEC. Instead, the different PaRSEC tasks that comprise a subroutine will pass their output to the tasks that comprise another subroutine using the internal communication engine of PaRSEC. This will be done implicitly, without user involvement, since PaRSEC handles communication internally.

5. Dataflow as a programming paradigm

In this section we will discuss the reasons why the PaRSEC implementation of CC is faster than the original code by contrasting the corresponding programming paradigms.

5.1. Communication computation overlapping

The original code is written in FORTRAN[3] and makes calls to the GA toolkit for communication and synchronization purposes. Global Arrays enables applications to adopt the Partitioned Global Address Space (PGAS) programming model and provides primitives for one-sided communication and atomic operations. However, the structure of the CC code that makes the communication calls does not take advantage of these more advanced concepts and rather follows a more traditional Corse Grain Parallelism (CGP) programming paradigm. Let us consider again the pseudocode shown in Figure 1. This figure abstracts away many of the details of the actual code but captures the structure very accurately. Namely, the work is organized in subroutines (that are further organized in logical steps) and inside each subroutine there are multiple nested loops with the call to the most computationally intensive functions (i.e. `DGEMM` and `TCE_SORT_*`) contained in the innermost loop. Interestingly, the calls to the functions that fetch the data to be processed by these calls (i.e.`GET_HASH_BLOCK`) are also in the innermost loop, immediately preceding the computation. In other words, the program contains no additional work that is available between the call to the data transfer function and the call to the computation function that uses the data. This means no matter how sophisticated and efficient the communication library and the network hardware is, this data exchange will not be overlapped with computation. This behavior leads to a major waste of efficiency, since at almost any given point in the program there is additional work (in other subroutines) that is not semantically dependent on the work currently being performed. However, the coarse-grain structure of the program, with work and communication contained in deep loop nests inside subroutines, does not allow for different units of work, that are independent to one another, to be utilized for communication–computation overlapping.

This missed opportunity for overlapping can be witnessed in the execution trace shown in Figure 5a. The figure shows an enlarged segment of the execution trace to improve readability. The shown segment is representative of the whole execution and does not exhibit a unique behavior. In this trace, the different colors represent different operations (i.e.`GEMM`, `SORT`, data transfer), the x-axis represents time (i.e. longer boxes signify operations that took longer to finish), and each row corresponds to one MPI rank (i.e. one instance of the parallel application) out of the total 224 used for this run. The red color represents the execution of a matrix multiply (`GEMM`), which is the most computationally expensive operation in this code. The blue and purple colors represent data transfers (matrices A and B needed to perform the $C+=A*B$ operation that a `GEMM` performs). As can be easily seen by the prominence of the blue color in the trace, the communication imposes an unacceptably high overhead in the execution of the original code.

This behavior is in stark contrast with the dataflow-based execution model that the PaRSEC-enabled version of the code follows. In the latter, computation load is organized in tasks (not loops, or subroutines), and tasks declare to the runtime their dependencies to other tasks. When a task completes, the runtime can

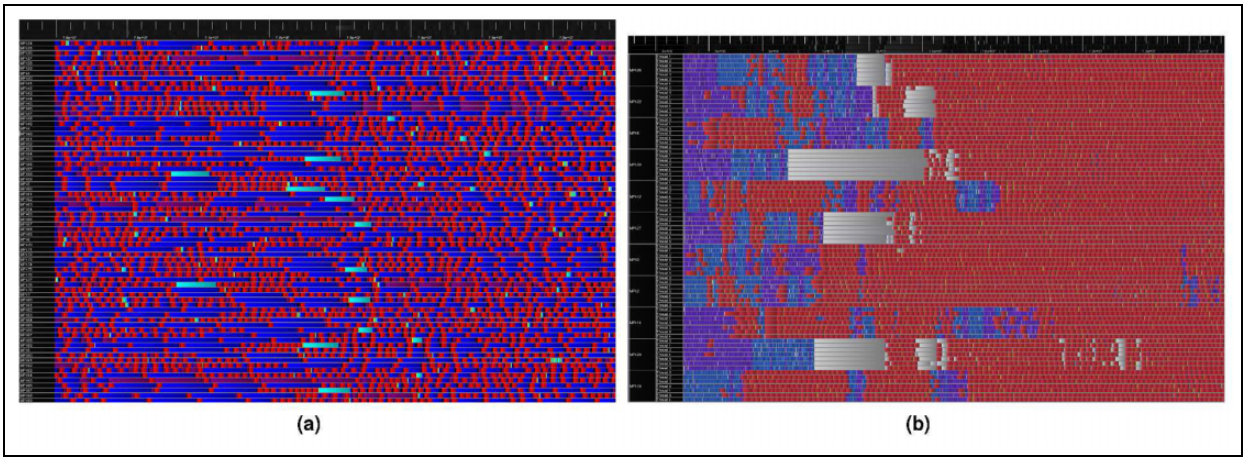

Figure 5. Execution traces: (a) trace of original NWChem code; (b) trace of dataflow-based NWChem code.

initiate the asynchronous transfer of its output data while scheduling for execution the next task whose dependencies have been satisfied. This opportunistic execution, typical of dataflow-based systems, allows for maximum communication computation overlapping. This is true because at any given time during the life of the program, if there is available computation, it will be performed without unnecessary waiting for some predetermined order of loops or subroutines to be satisfied. The opportunistic execution of the dataflow-based code is demonstrated in the trace shown in Figure 5b. As can be seen in this figure, which is an enlarged view of the beginning of the execution, many of the tasks, responsible for reading input data (blue and purple boxes) and sending it to the tasks that will perform the computation (red boxes), are scheduled for execution early on. However, as soon as some data transfers complete, computation tasks become ready and, from that point on, communication and computation are overlapped. It is worth noting that, in contrast with the trace of the original code (Figure 5a), the length of the blue and purple boxes in Figure 5b do not represent the communication cost of the data transfers, since tasks in PaRSEC do not perform explicit communication. Rather, these boxes represent the tasks that merely express their communication needs to the runtime by specifying their dependencies to other tasks. The actual data transfers are scheduled and managed by the runtime and are fully overlapped with the computation. As a result, they do not appear in the trace, but they are responsible for the light gray gaps (Figure 5b), which are periods of time where no work can be executed because the corresponding data has not been transferred yet (a phenomenon that only happens at the beginning of the execution). To summarize, by comparing the two traces we can clearly see that the original NWChem code faces significant communication delays which can be largely addressed through the use of communication–computation overlapping.

One could argue that the original FORTRAN code can be modified to allow for more communication–computation overlapping without resorting to dataflow-based programming. A developer could reorganize the loop nests into a form of a pipeline, so that each iteration "prefetches" the data needed for the computation of a future iteration. This can be achieved if every iteration initialized an asynchronous transfer for data needed by future iterations and then proceeded to execute work whose data was prefetched by a previous iteration. This would probably increase communication-computation overlapping and decrease waiting time, however, it would not be sufficient to achieve the performance improvements gained in the dataflow-based execution. The reason can be seen in the trace. In each row (i.e. for every MPI rank) there are time periods where communication (blue) takes a small amount of time in comparison with useful work (red). However, in each row there are also time periods where a few communication operations take significantly longer than the following computation. As a result, pipelining work and data transfer would only remove a small part of the communication overhead, unless a very deep pipeline is used, which would lead to significant temporary storage overhead (because all of the incoming data that correspond to future iterations have to be stored in temporary buffers). In the case of our dataflow-based version of CC, the runtime does not merely pipeline loop iterations, but overlaps communication with completely independent computation. Work that in the original code resides in completely different subroutines and can not be utilized for overlapping, unless a major restructuring of the code is performed.

5.2. Freedom from control flow

As we mentioned earlier, the original code executes all GEMM operations in serial chains and only allows for parallelism between different chains. This structure

defines both the order in which operations (that are semantically independent) will execute and the granularity of parallelism. Changing either while preserving the straight forward structure of the original code is not a trivial exercise. The dataflow-based PaRSEC form of the code that we have created departs from the simplicity of the original code, but it is not subject to either limitation mentioned above. Namely, as we discussed earlier, in PaRSEC we execute all GEMM operations in parallel and perform a reduction on the output data of all GEMM operations that belonged to a chain in the original code. In this way, we preserve the semantics of the code, but liberate the execution from the unnecessary limitations on the granularity of parallelism and strict ordering that were not dictated by the semantics of the algorithm, but rather by inherent limitations of control-flow-based programs and coarse grain parallelism.

5.3. Multi-threading and accelerators

Another artifact of the way the original code is structured is that taking advantage of multi-threading to utilize multiple cores on each node is not easy to implement efficiently. As a result, the NWChem application does not use threads and rather relies on multiple MPI ranks per node in order to utilize multiple cores. In PaRSEC, multi-threading comes at no additional cost for the developer. Once the developer has defined the tasks and the dataflow between them, PaRSEC will automatically use threads to accomplish the work, utilizing multiple hardware resources. Furthermore, PaRSEC uses a combination of thread local queues and global queues to store available tasks during the execution. When a task T_i completes, the tasks T_j that were waiting for the output of T_i will be placed in the thread local queue of the thread that executed T_i. As a result, work that should naturally execute as a chain (because the output of one task is the input of another) has a high probability of executing as a chain in PaRSEC taking advantage of cache locality and increasing execution efficiency. At the same time, the existence of global queues and work stealing guarantees that PaRSEC will also exhibit good load balance.

While outside the scope of this paper, PaRSEC also enables use of accelerators without too much complexity overhead for the developer. If the developer provides a kernel that can execute on an accelerator, and specifies the availability in the PTG representation of the code, then PaRSEC will execute this kernel on the accelerator. In the work we are current pursuing, we are experimenting with the execution of some of the GEMM operations on an Intel Xeon Phi aiming to maximize performance, given the tradeoffs between using that additional computing power of the accelerator and paying the overhead of transferring the necessary data to it.

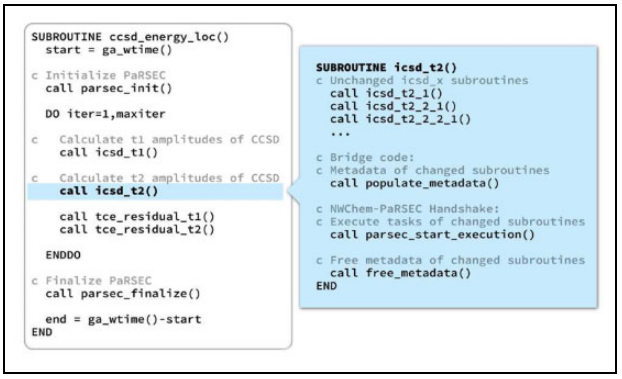

Figure 6. High-level view of PaRSEC code in NWChem.

6. 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 6 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 Figure 6) 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 earlier; 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.

6.1. 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 7a 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). In addition, Figure 7b 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 approximately 91% of the execution time of all 41 T2 subroutines in the original NWChem TCE CCSD execution.

The scalability tests for the original TCE-generated code and the dataflow version of PaRSEC:T2 were

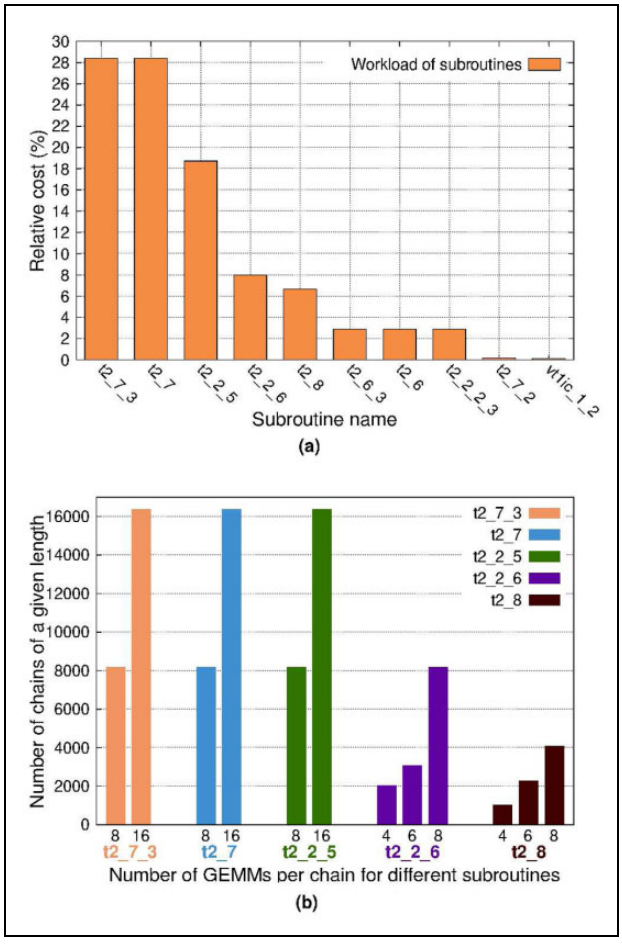

Figure 7. CCSD statistics for beta-carotene and tilesize of 45: (a) relative load of heaviest subroutines; (b) chain length distribution.

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.

6.2. Discussion

Figure 8 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) solid 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.

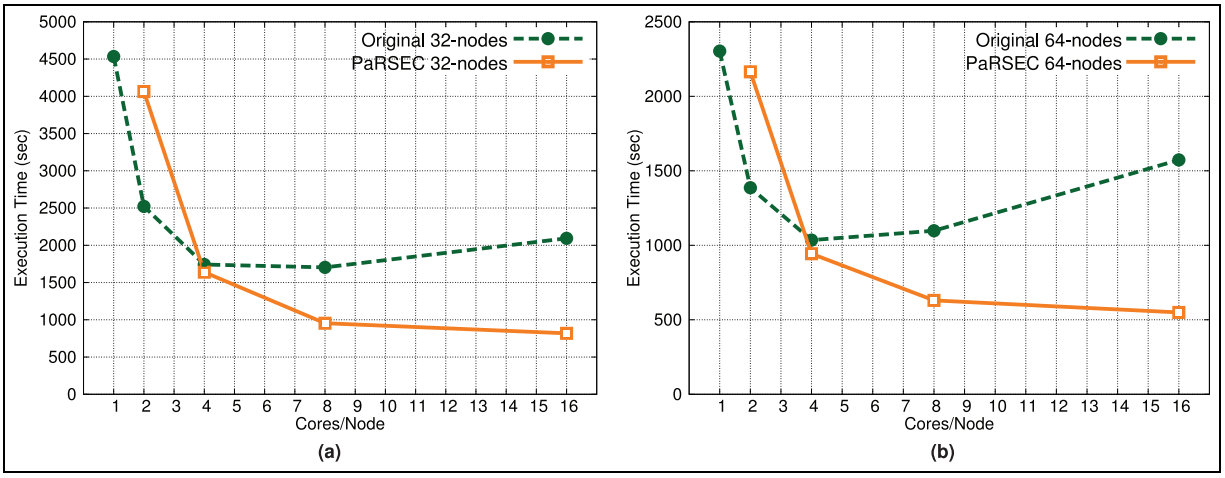

Figure 8. Execution time comparison using beta-carotene on EMSL/PNNL Cascade: (a) performance on 32 nodes; (b) performance on 64 nodes.

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 with 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.

7. Related work

An alternate approach for achieving better load balancing in the TCE CC code is the Inspector-Executor methods (Ozog et al., 2013). 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 (Lotrich et al., 2008) 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) (Deumens et al., 2011). 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 (Lai et al., 2013) has been designed with the goal of providing 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 (Solomonik et al., 2013).

8. 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 CC methods enables more efficient and scalable computation, as our performance evaluation reveals a performance boost of a factor of two for the entire CCSD kernel.

This strategy with PaRSEC offers many advantages since communication becomes implicit (and can be overlapped with computation), finer-grained tasks can be executed in more efficient orderings than sequential chains (i.e. binary trees) and each of these finer-grained parallel tasks are able to run on different cores of multicore systems, or even different parts of heterogeneous platforms. This will enable computation at extreme scale in the era of many-core, highly heterogeneous platforms, utilizing the components (e.g. CPU, GPU accelerators, and/or Xeon Phi coprocessors) that perform best for the type of task under consideration.

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.

Acknowledgements

We thank the anonymous reviewers for their improvement suggestions. 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.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This material is based upon work supported in part by the Air Force Office of Scientific Research under AFOSR Award Number FA9550-12-1-0476, and the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-SC0006733 "SUPER - Institute for Sustained Performance, Energy and Resilience".

Notes

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()
2. The 10 subroutines are t2_2_1(), t2_2_6(), t2_2_4_1(), t2_2_4_2(), t2_2_5_1(), t2_2_5_2(), t2_2_2_1(), t2_2_2_3(), t2_2_2_2_1(), t2_2_2_2_2().

3. NWChem uses a mixture of FORTRAN dialects, including 77 and more modern ones.

References

Bartlett RJ and Musial M (2007) Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics* 79(1): 291–352.

Valiev M, Bylaska EJ, Govind N, et al. (2010) NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181(9): 1477–1489.

Bosilca G, Bouteiller A, Danalis A, et al. (2012) DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing* 38(12): 37–51.

Cosnard M and Loi M. (1995). Automatic task graph generation techniques. In: Proceedings of the 28th Hawaii International Conference on System Sciences, pp. 113–122. Washington, DC: IEEE.

Danalis A, Bosilca G, Bouteiller A, et al. (2014) PTG: an abstraction for unhindered parallelism. In Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on, pp. 21–30. Washington, DC: IEEE.

Hirata S (2003) Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A* 107(46): 9887–9897.

Nieplocha J, Palmer B, Tipparaju V, et al. (2006) Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications* 20(2): 203–231.

Purvis G and Bartlett R. (1982) A full coupled-cluster singles and doubles model - the inclusion of disconnected triples. *The Journal of Chemical Physics* 76(4): 1910–1918.

Raghavachari K, Trucks GW, Pople JA, et al. (1989) A 5th-order perturbation comparison of electron correlation theories. *Chemical Physics Letters* 157(6): 479–483.

Kowalski K, Krishnamoorthy S, Olson R, et al. 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. Washington, DC: IEEE.

McCraw H, Danalis A, Herault T, et al. (2014) Utilizing dataflow-based execution for coupled cluster methods. In: *Proceedings of IEEE Cluster 2014*, pp. 296–297. Washington, DC: IEEE.

Danalis A, Jagode H, Bosilca G, et al. (2015) PaRSEC in practice: Optimizing a legacy chemistry application through distributed task-based execution. In: 2015 IEEE International Conference on Cluster Computing, pp. 304–313. Washington, DC: IEEE.

Jagode H, Danalis A, Bosilca G, et al. (2015) *Parallel processing and applied mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6–9, 2015.* Revised Selected Papers, Part I, chapter Accelerating NWChem Coupled Cluster Through Dataflow-Based Execution, pp. 366–376. Cham: Springer International Publishing.

Ozog D, Shende S, Malony A, et al. (2013) Inspector/executor load balancing algorithms for block-sparse tensor contractions. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, pp. 483–484. New York, NY: ICS '13, ACM.

Lotrich V, Flocke N, Ponton M, et al. (2008) Parallel implementation of electronic structure energy, gradient and hessian calculations. *The Journal of Chemical Physics* 128: 194104.

Deumens E, Lotrich VF, Perera A, et al. (2011) Software design of aces iii with the super instruction architecture. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 1(6): 895–901.

Lai PW, Stock K, Rajbhandari S, et al. (2013) 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. New York, NY: ACM.

Solomonik E, Matthews D, Hammond J, et al. (2013) Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In: Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 813–824. Washington, DC: IEEE.

Author biographies

Heike Jagode is a Research Director with the Innovative Computing Laboratory at the University of Tennessee, Knoxville. She specializes in high-performance computing (HPC) and the efficient use of advanced computer architectures; focusing primarily on developing methods and tools for performance analysis and tuning of parallel scientific applications. Her research interests include the multi-disciplinary effort to convert computational chemistry algorithms into a dataflow-based form to make them compatible with next-generation task-scheduling systems, such as PaRSEC. She is currently pursuing a PhD in Computer Science from the University of Tennessee, Knoxville. Previously, she received an MSc in High-Performance Computing from The University of Edinburgh, Scotland, UK; an MSc in Applied Techno-Mathematics and a BSc in Applied Mathematics from the University of Applied Sciences Mittweida, Germany.

Anthony Danalis is currently a Research Scientist II with the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests come from the area of HPC. Recently, his work has been focused on the subjects of performance analysis, system benchmarking, compiler analysis and optimization, dataflow programming, and accelerators. He

received his PhD in Computer Science from the University of Delaware on compiler optimizations for HPC. Previously, he received an MSc from the University of Delaware and an MSc from the University of Crete, Greece, both on Computer Networks, and a BSc in Physics from the University of Crete, Greece.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced- computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE IPDPS Charles Babbage Award; and in 2013 he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the US National Academy of Engineering.