Utilizing Dataflow-based Execution for Coupled Cluster Methods

Heike McCraw*, Anthony Danalis*, Thomas Herault*, George Bosilca*, Jack Dongarra*,
Karol Kowalski† and Theresa L. Windus‡
*Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville
†Pacific Northwest National Laboratory, Richland, WA
‡Iowa State University, Ames, IA

***Abstract*—Computational chemistry comprises one of the driving forces of High Performance Computing. In particular, many-body methods, such as Coupled Cluster (CC) methods of the quantum chemistry package NWCHEM, are of particular interest for the applied chemistry community.**

Harnessing large fractions of the processing power of modern large scale computing platforms has become increasingly difficult. With the increase in scale, complexity, and heterogeneity of modern platforms, traditional programming models fail to deliver the expected performance scalability. On our way to Exascale and with these extremely hybrid platforms, dataflow-based programming models may be the only viable way for achieving and maintaining computation at scale.

In this paper, we discuss a dataflow-based programming model and its applicability to NWCHEM's CC methods. Our dataflow version of the CC kernels breaks down the algorithm into fine-grained tasks with explicitly defined data dependencies. As a result, many of the traditional synchronization points can be eliminated, allowing for a dynamic reshaping of the execution based on the ongoing availability of computational resources. We build this experiment using PARSEC – a task-based dataflow-driven execution engine – that enables efficient task scheduling on distributed systems, providing a desirable portability layer for application developers.

I. INTRODUCTION

Computational chemistry, aiming to simulate non-trivial physical systems, 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 methods [1] (CC) of the quantum chemistry package NWCHEM [2] are both computationally intensive and of interest to the computational chemistry community.

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 last couple of decades. This is true due to both the increasing scale and the increasing complexity and heterogeneity of modern (and projected future) platforms. We believe that dataflow-driven task-based programming models may be the only viable way to achieve computation at scale, especially on distributed heterogeneous architectures.

The Parallel Runtime Scheduling and Execution Control (PARSEC) framework [3] is a task-based dataflow-driven runtime that enables high performance computing at scale. 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 of supercomputers from the application developer, while not hindering the achievable performance. The main difference between PARSEC and other task engines is the way tasks, and their data dependencies, are represented, enabling PARSEC to employ a unique, symbolic way of discovering and processing the graph of tasks. Namely, PARSEC uses a symbolic Parameterized Task Graph (PTG) [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 to 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 entire DAG that represents the whole execution of a parallel application.

However, utilizing a task scheduling system such as PARSEC to execute the Coupled Cluster code of NWCHEM is not trivial. The CC code – generated by the Tensor Contraction Engine (TCE) [5] – is neither organized in pure tasks, nor is the control flow affine, as required by the front-end compiler of PARSEC for automatic PTG generation. For example, the CC code contains branches whose predicates depend on program data. Nevertheless, the program data that controls the behavior of these branches is set only once, upon the initialization stages of the code, and does not change during the execution. Capitalizing on this fact, we have created a dataflow version of a subset of CC code subroutines by storing the data that affects the control flow of the program into meta-data structures that PARSEC examines during execution. This enables PARSEC to take full control of the communication and task scheduling, while preserving the semantics of the original CC code.

In this paper we outline PARSEC and NWCHEM, and describe the transformations we performed to a subset of the CC code so that it can run over PARSEC. We explain why PARSEC's execution model is (a) a good fit for the challenges posed by running CC at scale; and (b) more promising than the alternatives. Finally, we present experimental evidence supporting the expectation that a deeper integration of PARSEC and CC will enable more efficient execution at scale.

II. Overview of NWChem and PaRSEC

A. NWChem

Computational modeling has become an integral part of many research efforts in key application areas in chemical, physical, and biological sciences. NWChem [2] 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 Coupled Cluster theory [1] (CC) 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.

Tensor Contraction Engine: An important role in designing the optimum memory vs. cost strategies in coupled cluster implementations is played by the automatic code generator, TCE [5], 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) [6] functionalities, which supports the distributed memory programming model.

CC Single Double (CCSD): Especially important in the hierarchy of the CC formalism is the iterative CC model with single and double excitations (CCSD) [7], which is a starting point for many accurate perturbative CC formalisms including the ubiquitous CCSD(T) approach [8]. 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 [9]. 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.

B. PaRSEC

The natural decomposition of NWChem into tasks makes it a good match for PaRSEC. The PaRSEC framework [3] is a task-based dataflow-driven system designed as a dynamic platform that can address the challenges posed by distributed heterogeneous hardware resources. The central component of the system, the *runtime*, orchestrates the execution of the tasks on the available hardware. Choices regarding the execution of the tasks are based on information provided by the user regarding the tasks that comprise the user application, and the dataflow between those tasks. This information is provided in the form of a compact, symbolic representation of the tasks and their dependencies known as a Parameterized Task Graph (PTG) [4]. The runtime combines the information contained in the PTG with supplementary information provided by the user – such as the distribution of data onto nodes, or hints about the relative importance of different tasks – in order to make efficient scheduling decisions.

The PTG can be understood as a compressed representation of the DAG that describes the execution of a task-based application. As an example, consider the code shown in Figure 1 and consider that the two lines in the body of the loop correspond to the two different tasks `PING` and `PONG`. An abstract form of the PTG that describes this program is

```
for(step=0; step<max_steps; step++){
    A[0] = A[0] + A[1]; // PING( IN: A[1], INOUT: A[0] )
    A[1] = A[1] + A[0]; // PONG( IN: A[0], INOUT: A[1] )
}
```

Fig. 1. Serial code for ping-pong.

shown in Figure 2. As can be seen in the figure, the PTG has an entry for each class of tasks, as opposed to one entry for each individual task that will execute at run-time. Task classes are parameterized (using parameter "`s`" in this example) and their dependencies to other task classes depend only on their local parameters. Each unique value of the parameters corresponds to a particular task instance. Clearly, for each task instance, simple evaluation of the algebraic expressions in the PTG reveal the predecessors and the descendants of the task, or in other words, discovering the communication peers of each task requires nothing more than the evaluation of the expressions that appear in the PTG, given particular values for the parameters.

```
PING(s)
  s = 0..max_steps-1

  READ A1  <- A1 PONG(s-1)
  WRITE A0 -> A0 PONG(s)
END

PONG(s)
  s = 0..max_steps-1

  READ A0  <- A0 PING(s)
  WRITE A1 -> A1 PING(s+1)
END
```

Fig. 2. Abstract PTG for ping-pong.

PaRSEC is an event driven system. When an event occurs (i.e., a task completes), the runtime reacts by examining the dataflow from this task. This reveals what future tasks can be executed based on the data generated by the completed task. Since the tasks and their dataflow are described in the PTG, discovering the future tasks, given the task that just completed, does not involve expensive traversals of DAG structures stored in program memory. This contrasts to other task scheduling systems which rely on building the whole DAG of execution

in memory at run-time and traversing it in order to make scheduling decisions. Beyond scheduling tasks, the runtime also handles the data exchange between distributed nodes, thus it reacts to the events triggered by the completion of data transfers as well. When the hardware is busy executing application code – and thus no events are triggered – the runtime does not incur overhead.

Due to the PTG representation, all communication becomes implicit and thus is handled automatically by the runtime. In MPI (or other Bulk Synchronous programming models), the developer has to explicitly specify the point during the execution at which every message should be sent or received*. In PARSEC the runtime performs all necessary data exchanges without user intervention. This is possible because the PTG provides the necessary information regarding the data that each task needs in order to execute, and the runtime is aware of the mapping of tasks onto compute nodes. This approach provides multiple benefits.

1) Application development is simplified, since communication management does not need to be handled by the developer.
2) It allows the runtime to automatically make use of efficient non-blocking communication and advanced collective communication algorithms to achieve communication-computation overlapping and hide significant parts of the communication overhead.
3) The decoupling of the computation and communication code allows for easy experimentation with alternative communication strategies.

As an example, in the work presented in this paper, we modified the ordering and the communication of the computation kernels from the original serial chain pattern to an embarrassingly parallel pattern followed by a binary tree reduction. Performing this change required minimal effort, which included little more than changing a few dataflow edges in the PTG of our application and incorporating the PTG for a generic binary tree reduction provided by PARSEC.

The runtime is also responsible for scheduling tasks within each node. Specifically, every task that completes generates data that enables the execution of other tasks. The runtime keeps track of all completed tasks and uses the PTG to discover the tasks that can execute next, without performing expensive traversals of DAG structures in memory.

In summary, PARSEC provides the opportunity for parallel applications to enjoy high efficiency at scale, without putting the burden of micromanaging data-transfers, processes, threads, and other exotic low level library primitives and interfaces on the application developer.

*Non-blocking communication seemingly allows for some flexibility, but in reality its use is limited, managing large numbers of outstanding messages quickly becomes a logistical overhead, and many non-blocking data transfers end up happening when the `wait()` operation is called instead of truly asynchronously.

III. Implementation of Coupled Cluster Theory

A. Coupled Cluster Theory through TCE

In NWCHEM, the CC methods, among other kernels, are generated through the TCE into multiple sub-kernels that are divided into so-called "T1" and "T2" subroutines for equations determining `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` 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 [6] – through the TCE. For instance, for the CCSD generated code, there exist 19 `T1` and 41 `T2` subroutines, and all of them highlight a very similar code structure and patterns. The FORTRAN code that is generated for the `T1` and `T2` subroutines includes most work in deep loop nests. In these loop nests there are three types of code. These are:

- Local memory management (i.e., `MA_PUSH_GET()`, `MA_POP_STACK()`),
- Calls to GA functions that transfer data over the network (i.e., `GET_HASH_BLOCK()`, `ADD_HASH_BLOCK()`), and
- Calls to the subroutines that perform the actual computation on the data (i.e., `SORT()`, `GEMM()`).

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.

Parallelism of the TCE generated CC code follows a task-stealing model. The work inside each `T1` and `T2` subroutine is grouped into chains of multiple matrix-multiply kernels (GEMM) and those chains are executed in a parallel fashion. However, multiple subroutines are divided into different levels (e.g., the 19 `T1` subroutines are divided into 3 different levels). The task-stealing model is only executed within each level, and there is a synchronization step between the levels. Load balancing within each level of subroutines is achieved through shared variables 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. Additionally, 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. Finally, the notion of *task* in the current CC implementation in NWCHEM and the notion of *task* in PARSEC are not identical. 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 by breaking down the algorithms into finer grained tasks with explicitly defined dependencies.

B. 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. 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).eq.int_mb(k_spin+p3b-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 CC code is neither affine, nor statically decidable, all the program data that affects the behavior of CC is constant during a given execution of the code. Therefore, the code can be expressed as a parameterized DAG, by using lookups into the data of the program, or similar meta-data structures constructed from the program data as soon as the latter is known.

In the work described in this paper, we hand-modified the body of one of the `T1` subroutines – namely `icsd_t1_2_2_2()`. The original code of `icsd_t1_2_2_2()` consists of a loop-nest of depth four that contains 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 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 a PARSEC-enabled version of the subroutine we decomposed it 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 performing all the lookups into the program data (i.e., `int_mb(k_spin+h7b-1)`) that is involved in the `IF` branches prior to the computation, and stores the results in new custom meta-data vectors. Since the data of NWCHEM that affects the control flow is immutable at run-time, this first step only needs to be performed once.

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 resulted in the execution of the computational kernels.

In addition to the first step, we created a PTG representation of the subroutine. Since the control flow depends on the program data that was examined by the first step, the PTG of the subroutine cannot contain only algebraic expressions for defining the dataflow between tasks. Instead, the PTG includes lookups into our custom meta-data vectors populated by the first step, so that the execution of the modified subroutine over PARSEC perfectly matches the original execution of `icsd_t1_2_2_2()`. Figure 3 shows one chain (out of a total of twelve) from the DAG generated by executing the PARSEC version of the subroutine using *uracil-dimer*, in 6-31G basis set composed of 160 basis set functions, as the input molecule. It is clear that the execution forms a chain, where each task

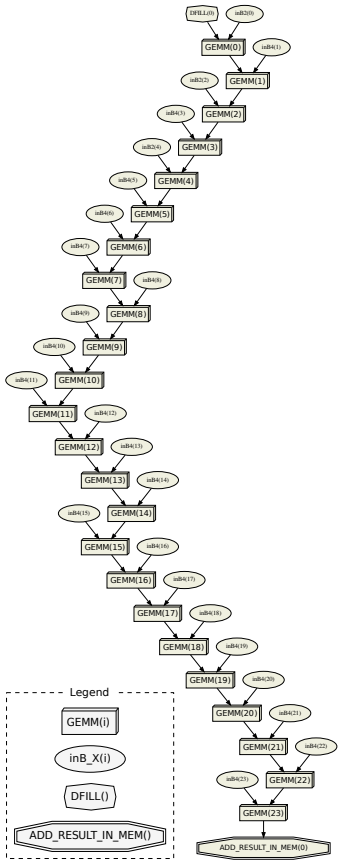

Fig. 3. Chain of GEMMs

(a GEMM in particular) has to wait for the completion of the previous one (as well as the task that reads the necessary input data). In terms of parallelism and load balancing, this perfectly matches the execution of the original NWChem code, where a series of GEMM operations, executed serially in a loop, constitutes a single task. In the case of *uracil-dimer* there are twelve such independent chains, and each one performs twenty four GEMM operations, as shown in the figure.

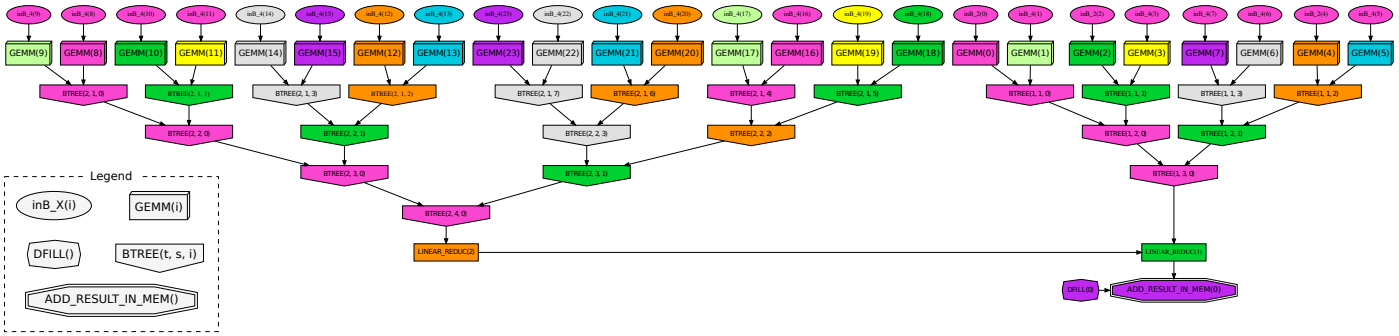

Fig. 4. Parallel GEMMS and binary reduction of results

However, 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. A GEMM kernel performs the operation:

$$C = \alpha * A * B + \beta * C$$

where A,B, and C are matrices and α and β are scalar constants. In the chain of GEMMs performed by the `t1_2_2_2()` subroutine, the result of each matrix multiply is added to the result of the previous matrix multiply (since $\beta = 1$). Since matrix addition is an associative and commutative operation, the order in which the GEMMs are performed does not bare 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. Figure 4 shows the DAG of one of the twelve "chains" of GEMMs for the *uracil-dimer* input, implemented with a binary reduction, using PARSEC.

Clearly, in this implementation there are significantly fewer sequential steps than in the original chain. In addition, the sequential steps are matrix additions, not GEMM operations, so they are significantly faster, especially for larger matrices. Furthermore, since the depth of the binary reduction trees grows with the logarithm of the total count of GEMM operations, as the problem size gets bigger, the difference in number of sequential steps performed by the chain and the binary tree will grow fast.

It is important to note that the original version of the code performs an atomic accumulate-write operation (via GA functionality `ADD_HASH_BLOCK()`) at the end of each chain. Since our dataflow version of the code computes the `GEMM`'s for each chain in parallel, we eliminate the atomic GA functionality and perform direct memory access instead.

The colors in Figure 4 were used to demonstrate the utilization of resources when this approach is used. Namely, each color corresponds to one of the eight nodes that participated in this run. Where the original CC implementation treats the entire chain of GEMM operations as one "task" and therefore assigns it to one node, our new implementation of `t1_2_2_2()` over PARSEC distributes the work onto different hardware resources leading to better load balancing and the ability to utilize additional resources, if available. That is, the PARSEC version is by design able to achieve better strong scaling (constant problem size, increasing hardware resources) than the original code.

IV. COMPUTATION MODELS FOR EXASCALE

Pursuing novel programming paradigms for high performance computing is becoming an increasingly popular trend as more and more people in the field are coming to realize that the Bulk Synchronous Programming model embodied by MPI and other similar explicit communication paradigms is unlikely to satisfy the needs of computing at exascale and beyond.

In particular, task-based dataflow programming has gained a lot of attention as is evident by the increasing number of task execution runtimes [10], [11], [12], [13] that are being pursued by research groups all around the globe. At the time of this writing, PARSEC stands unique among them in its use of the PTG representation of parallel programs. Constructing a purely algebraic PTG to describe a parallel program poses some challenges, but once this is achieved, PARSEC can utilize the PTG to deliver unparalleled performance on large scale distributed memory executions. The multiple reasons that make this true can be easily demonstrated in theory. For example, unlike runtime systems where the whole DAG is traversed and stored in memory: (a) the PTG is not problem size dependent, (b) all tasks can be examined at any time during execution (i.e., there is no "window" of visible future tasks), and (c) there is no redundant part of the DAG that needs to be traversed at every node due to tasks that some other node will execute. In addition to the theoretical evidence about the positive effects of the PTG, the performance of the DPLASMA [14] library, built on top of PARSEC, offers a concrete experimental demonstration of the performance scalability that the PTG approach offers.

However, the implementation of the Coupled Cluster code of NWCHEM, described in this paper, does not rely on a purely algebraic PTG. Instead, as we mentioned earlier, the PTG performs lookups into meta-data vectors that are populated by running a skeleton program which is based on the control flow of the original application. This step is similar to the skeleton program executed by runtimes that store the entire DAG in memory, in several ways, but the two models are not equivalent for the reasons we analyze below.

In terms of similarities between our meta-data building step and DAG building steps of other runtimes, they both need to execute all iterations of the loops of the original program and

†Changing the ordering of GEMM operations leads to results that are not bitwise equal to the original, but this level of accuracy is very rarely required, and is lost anyway when transitioning to different compilers and/or math libraries.

evaluate all IF statements of the original program, so they both have the same algorithmic complexity. However, the amount of work and the amount of memory overhead at every step is wildly different.

In the case of PARSEC, the task classes are known through static analysis of the code, whether this analysis is performed automatically by a compiler tool, or manually by a human developer, as is the case for the work presented in this paper. This static analysis reveals the patterns of dependencies in the code. For example, in the case of `icsd_t1_2_2_2()`, static code analysis revealed that there is always a `GET_HASH_BLOCK` and a `DFILL` operation, initializing the matrices, followed by a chain of `GEMM` operations, followed by an `ADD_HASH_BLOCK` operation that pushes the result of the chain back into the *global array*. Therefore, our skeleton program that populates the custom meta-data vectors only needs to discover information such as the length of each chain of GEMMs, and only store a handful of variables into the vectors at every step.

A DAG building skeleton program, in contrast, assumes nothing about the shape of the DAG. Instead, it needs to record the pointers to the program data that is read and modified by each task. Then it must use these pointers, for each newly discovered task, to identify the previous task that read or modified the same data, in order to build the DAG that represents the execution of the program. This is clearly a much more time-consuming operation.

Also, the memory requirements in the case of PARSEC are lower than in the case of a runtime that builds the whole DAG in memory. This is true because the meta-data vectors used by PARSEC are custom per application and are crafted to store the minimum number of elements needed to reproduce the behavior of the original program. In the case of DAG build runtimes, the structures that hold the nodes and the edges of the DAG have to be generic enough to accommodate any type of program and thus they must employ expensive lists and other memory consuming data structures.

The second major difference between PARSEC's meta-data driven PTG and alternative approaches that build the program's DAG, is the adherence to the original program's control flow. As we described earlier in this paper, we have knowledge of the type of operation performed by the chained tasks (i.e., matrix-matrix multiply) and the mathematical properties that govern it (i.e., matrix addition being associative and commutative). As a result, we can modify our PTG to sum the resulting matrices using a binary tree instead of a linear chain. It is important to note that performing this optimization was not only technically easy, but it did not require any changes in the meta-data, or the skeleton code that populates the meta-data. It only involved modifying some dataflow edges in the PTG. In contrast, a runtime that builds the DAG by following the execution of the original program will construct a chain of tasks, and it will have no way of identifing that this chain could be rearranged as a binary tree. In other words, the DAG building approach is forced to adhere to the (potentially suboptimal) control flow of the original program.

While for small examples the differences between these two models might be masked by other factors and design choices between different runtimes, we assert that at exascale, even when using skeleton programs to populate custom meta-data vectors that will be used by the PTG, the PARSEC approach has the potential to deliver higher performance than approaches that build the entire execution DAG in memory.

V. Related Work

The Cyclops Tensor Framework [15] uses an orthogonal approach to TCE. It uses cyclic data decomposition to compute (mainly dense) tensor contractions, with the objective to eliminate scalability issues of load imbalance and irregular communication.

An alternate approach for achieving better load balancing in the TCE CC code is the Inspector-Executor methods [16]. The Inspector phase loops through the subroutines and creates an informative list of tasks as it uncovers sparsity information. The Executor phase loops through this list of tasks and aggregates the computations into a single task. Prior to the Executor phase, performance model based cost estimation techniques for the computations (e.g., GEMM and SORT) are applied to assign the aggregated tasks to processors. This technique focuses on balancing the computational cost without taking data locality into consideration.

Another method that has been effectively used to parallelize CC codes is through ACES III [17]. In this work, the CC algorithms are designed in a domain specific language named super instruction assembly language (SIAL) [18]. 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, communicated information between tasks, for retrieving data, and then for execution.

The Dynamic Load-balanced Tensor Contractions framework [19] 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 [15].

In an effort to offer alternative programming paradigms to the Bulk Synchronous Parallel model offered by MPI there has been a significant body of work on languages, or language extensions, such as the PGAS languages [20], [21], [22], [23], [24], where the compiler is expected to perform the parallelization of the input program. Habanero [25] combines a compiler and a runtime to achieve parallelism and schedule tasks, and relies on language extensions that a human developer must place into his or her application to guide task creation and synchronization. Bamboo [26] is another compiler tool that utilizes the prototype runtime system Tarragon [13] for scheduling tasks extracted from annotated MPI code. Bamboo's execution model is a form of fork-join parallelism, since it preserves the execution order of overlap regions, which run sequentially, one after the other. Also, the more mature Charm++ solution offers a combination of a programming language and a task scheduling backend. All of these solutions offer new languages, or extensions to existing languages that require specialized compilers and expect the developer to adopt

them as the programming paradigm of choice. In the work presented in this paper, we did not require the developers of NWCHEM to change the programming language they use, but rather adapted their FORTRAN 77 code to use our task scheduling runtime.

In terms of dataflow environments, several groups have studied parallel execution models since the early $1990's$ that (a) allowed the same code to run on shared memory and distributed memory systems, and (b) provided load balancing features for irregular applications [27], [28], [29]. Unfortunately, most of these systems are impossible to use and evaluate today. Newer approaches, such as PM2 [30], SMARTS [31], Uintah [32], and Mentat [27] exist, but do not satisfy the requirement for decentralized execution of medium grain tasks ($\approx 10\mu s - 1ms$) in distributed-memory environments.

Finally, there are several task scheduling systems that employ "Dynamic Task Discovery (DTD)", or in other words building the entire DAG of execution in memory using skeleton programs. Several projects are embracing this principle on shared memory (SMPSs [10], Cilk [33], Thread Building Blocks TBB [34]), or accelerator based systems (StarPU [11], CellSs, GPUSs [35], [36]). Some have support for medium size distributed memory machines, but with the introduction of synchronization at both endpoints. In this work we use PARSEC as our runtime system and take advantage of the PTG representation, in order to avoid the unnecessary overheads accosiated with DTD, as we explain further in Section IV.

VI. EXPERIMENTAL EVALUATION

In this section we illustrate the performance of one CC subroutine, `icsd_t1_2_2_2()`, that represents single excitations. For the purpose of this experiment we modified both the original code and our implementation so that the subroutine runs in isolation, without the other icsd subroutines that normally surround it. Also, we allowed for only one iteration of the iterative CC code. This reduced the noise in the measurements and made the test less "forgiving" since there is no additional workload to alleviate scalability shortcomings, or amortize overheads.

A. Methodology

As input, we used the beta carotene molecule in 6-31G basis set composed of 472 basis set functions. In our tests we kept all core electrons frozen and correlated 296 electrons. For the original version of the `icsd_t1_2_2_2()`, this results in 48 chains, each computing 48 sequential `GEMM`'s.

The scalability tests for the original TCE generated code and the dataflow version of `icsd_t1_2_2_2()` were performed on the Titan Cray-XK7 computer system of the National Center for Computational Sciences at Oak Ridge National Laboratory. Each node has 32 GB of RAM and one 16-core AMD Opteron (Interlagos) processor running at 2.2 GHz. We performed various performance tests utilizing 1, 2, 4, 8, and 16 cores per node. NWCHEM was compiled with the Intel compiler icc-13.1.3, and instead of using NWCHEM's internal reference BLAS implementation, we employed the optimized BLAS library ACML 5.3.1, provided on titan.

B. Discussion

Figure 5 shows the execution time of the subroutine `icsd_t1_2_2_2()` when the implementation found in the original NWCHEM code is used, and when our PARSEC based dataflow implementation is used.

In the graph we depict the behavior of the original code using the blue boxes with whiskers – light gray in grayscale – and the behavior of the PARSEC implementation using the red – dark gray – boxes. Note that the Y axis is logarithmic and we have "removed" the middle part of the axis to improve readability, since there is nothing but straight lines between $1s$ and $10^{-4}s$.

Since the beta carotene input results in only 48 chains of `GEMM` operations, and the original code uses entire chains as the unit of parallelism, it is expected that the code will only utilize up to 48 parallel processes[‡]. Each (light) blue box represents the execution time range observed for the 48 slowest processes of the original code (i.e., only the 48 processes involved in the computation of the chains). The whiskers show the execution time of the fastest process. These results clearly demonstrate that there are indeed 48 processes that perform the work and all other processes are idle (since their execution time is on the order of $100\mu s$). Interestingly however, in the case of the original NWCHEM code, as we increase the number of processes per node we observe a dramatic performance degradation. Indeed, the execution time of the 48 working processes (i.e., the boxes in the graph) goes from the order of $1s$ when using one core per node, to $5s$ with two cores, to $20s$ with eight and finally to over $50s$ when all the cores of each node are running NWCHEM processes. This is an unexpected outcome, since the additional processes are not involved in the computation. Furthermore, this behavior demonstrates the inability of the original code to utilize additional resources to speed up a fixed problem, i.e., lack of strong scaling.

The same graph depicts the behavior of our implementation of the subroutine over PARSEC using the (dark) red boxes. PARSEC uses one process per node and an increased number of threads per node when additional cores are being used. Since PARSEC causes an implicit synchronization between tasks at the beginning and end of execution, we could not differentiate between busy and idle threads. Therefore the (dark) red boxes depict the whole range from minimum to maximum. However, by design the PARSEC version of the code uses individual `GEMM` operations as the unit of parallelism (as opposed to a whole chain) and thus it is able to utilize more hardware resources. Also, PARSEC internally uses an additional thread for performing the communication, so the minimum recommended core count per node is two.

In terms of absolute time, the PARSEC version of the code is at best (for 16 cores/node) on a par with the best performance achieved by the original code (for 1 core/node). These results support the claim that a dataflow representation of CC has greater scalability potential than the current design, for the following reasons:

1) As discussed in section III-B, in order to create the dataflow version of CC we needed to create a skeleton

[‡]However, more than 48 nodes are needed due to memory requirements.

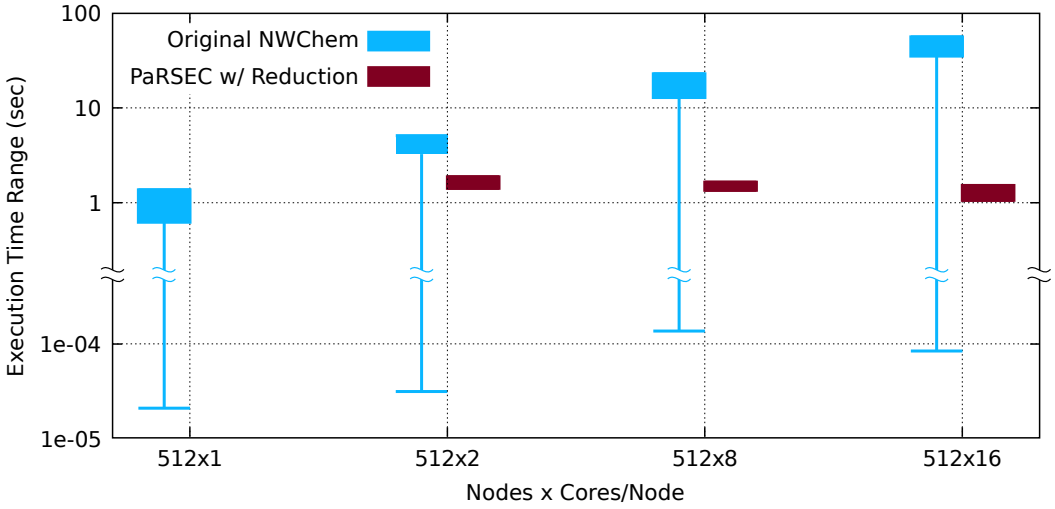

Fig. 5. Performance comparison of NWChem versions on titan using beta carotene

function that populates the meta-data vectors. This function incurs overhead in the computation. This overhead will be amortized when a larger number of CC subroutines are ported over PaRSEC, but in the experiments presented in this paper we timed a single subroutine and thus there is no amortization of the cost of the meta-data population step.

2) The cost of the meta-data population does not increase as the size of the GEMM operation increases. Thus, for larger input problems, where the GEMM operations will be more time consuming, the overhead plays a less significant role.
3) As the input problem increases, and the size of the chains of GEMM operations increases, the difference between the original chain implementation and the modified binary tree implementation becomes more important.
4) As can be seen in the graph, while the performance of the original code deteriorates with the increasing number of resources, the performance of the PaRSEC version improves as the number of cores per node increases. If we compare the performance of the original versus the modified code when all the cores per node (or half the cores per node) are used then our dataflow version of CC outperforms the original by more than an order of magnitude.

VII. 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. We performed this conversion manually, and thus we were limited to a small scope (the subroutine `icsd_t1_2_2_2`), yet this process provided a strong indication that utilizing dataflow-based execution for Coupled Cluster methods will enable more efficient computation at scale.

As a next step, we will automate the conversion of the entire NWChem TCE Coupled Cluster 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 may be adopted by the TCE engine.

With this new dataflow version of the *CC* kernel promoting much finer grained parallelism, most of the traditional synchronization points throughout each cycle of its iterative process are eliminated. 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 or GPU) that perform best for the type of task under consideration.

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”. This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] R. J. Bartlett and M. Musial, “Coupled-cluster theory in quantum chemistry,” *Reviews of Modern Physics*, vol. 79, no. 1, pp. 291–352, JAN-MAR 2007.

[2] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Aprà, T. L. Windus, and W. de Jong, “NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations,” *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, SEP 2010.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, “DAGuE: A generic distributed DAG engine for high performance computing,” *Parallel Computing*, vol. 38, no. 12, pp. 37 – 51, 2012, extensions for Next-Generation Parallel Programming Models. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001347

[4] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*. Washington, DC: IEEE Computer Society, 1995.

[5] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *Journal of Physical Chemistry A*, vol. 107, no. 46, pp. 9887–9897, NOV 20 2003.

[6] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006. [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/20/2/203

[7] G. Purvis and R. Bartlett, "A Full Coupled-Cluster Singles and Doubles Model - the Inclusion of Disconnected Triples," *Journal of Chemical Physics*, vol. 76, no. 4, pp. 1910–1918, 1982.

[8] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon, "A 5th-order perturbation comparison of electron correlation theories," *Chemical Physics Letters*, vol. 157, no. 6, pp. 479–483, 1989.

[9] K. Kowalski, S. Krishnamoorthy, R. Olson, V. Tipparaju, and E. Aprà, "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 International Conference for*, Nov 2011, pp. 1–10.

[10] J. M. Pérez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER)*, Tsukuba, Japan, 2008, pp. 142–151.

[11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[12] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2011.

[13] P. Cicotti, "Tarragon: a programming model for latency-hiding scientific computations," Ph. D. Dissertation, Department of Computer Science and Engineering, University of California, San Diego, 2011.

[14] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaeif, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPSW 2011)*. IEEE, 16-20 May 2011, pp. 1432–1441.

[15] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 813–824.

[16] D. Ozog, S. Shende, A. Malony, J. R. Hammond, J. Dinan, and P. Balaji, "Inspector/executor load balancing algorithms for block-sparse tensor contractions," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. ACM, 2013, pp. 483–484. [Online]. Available: http://doi.acm.org/10.1145/2464996.2467282

[17] V. Lotrich, N. Flocke, M. Ponton, A. Yau, A. Perera, E. Deumens, and R. Bartlett, "Parallel implementation of electronic structure energy, gradient and hessian calculations," *J. Chem. Phys, 128, 194104*, p. 15 pages, 2008.

[18] E. Deumens, V. F. Lotrich, A. Perera, M. J. Ponton, B. A. Sanders, and R. J. Bartlett, "Software design of aces iii with the super instruction architecture," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 1, no. 6, pp. 895–901, 2011. [Online]. Available: http://dx.doi.org/10.1002/wcms.77

[19] P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, and P. Sadayappan, "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*, ser. SC '13. ACM, 2013, pp. 13:1–13:10. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503290

[20] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, 2007.

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented approach to non-uniform Clustered Computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, October 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[22] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper, "Upc specification v. 1.1," http://upc.gwu.edu/documentation, 2003.

[23] R. W. Numrich and J. K. Reid, "Co-Array Fortran for parallel programming. ACM Fortran Forum 17, 2, 1-31," 1998.

[24] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," in *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[25] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1586640.1587670

[26] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, "Bamboo: translating mpi applications to a latency-tolerant, data-driven form," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 39:1–39:11.

[27] A. Grimshaw, W. Strayer, and P. Narayan, "Dynamic object-oriented parallel processing," in *IEEE Parallel and Distributed Technology: Systems and Applications*, May 1993, pp. 33–47, http://www.cs.virginia.edu/ mentat/.

[28] L. Kale, "Parallel programming with charm: An overview," Dept. of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 93-8, 1993.

[29] M. Rinard, "The design, implementation, and evaluation of jade, a portable, implicitly parallel programming language," Ph.D. dissertation, Stanford, CA, 1994.

[30] R. Namyst and J.-F. Méhaut, "PM^2: Parallel multithreaded machine. A computing environment for distributed architectures," in *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, ser. Advances in Parallel Computing, E. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, Eds., vol. 11. Amsterdam: Elsevier, North-Holland, Feb. 1996, pp. 279–285.

[31] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malonv, S. Shende, R. Oldehoeft, and S. Smith, "SMARTS: Exploting temporal locality and parallelism through vertical execution," in *Proceedings of the 1999 ACM SIGARCH International Conference of Supercomputing (ICS '99)*, June 1999, pp. 302–310.

[32] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *HPDC'00: Ninth IEEE International Symposium on High Performance and Distributed Computing*, August 2000.

[33] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings Symposium on Parallel Algorithms and Architectures*, July 1995.

[34] Intel, "Intel threading building blocks," http://threadingbuildingblocks.org/.

[35] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A programming model for the Cell BE architecture," in *SC'2006 Conference CD*. Tampa, FL: IEEE/ACM SIGARCH, Nov. 2006.

[36] E. Ayguade, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Orti, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Euro-Par 2009 Parallel Processing (15th Euro-Par'09)*, ser. Lecture Notes in Computer Science (LNCS), H. J. Sips, D. H. J. Epema, and H.-X. Lin, Eds. Delft, The Netherlands: Springer-Verlag (New York), Aug. 2009, vol. 5704, pp. 851–862.