Evaluation of Dataflow Programming Models for Electronic Structure Theory

Heike Jagode, Anthony Danalis, Reazul Hoque, Mathieu Faverge, Jack Dongarra

Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

ARTICLE TYPE

Evaluation of Dataflow Programming Models for Electronic Structure Theory

Heike Jagode*[1] | Anthony Danalis[1] | Reazul Hoque[1] | Mathieu Faverge[2] | Jack Dongarra[1]

[1]Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA

[2]Bordeaux INP, CNRS, Inria, Université de Bordeaux, Talence, France

Correspondence
*Heike Jagode
http://icl.utk.edu/ jagode/
Email: jagode@icl.utk.edu

Present Address
Innovative Computing Laboratory
University of Tennessee
Suite 203 Claxton
1122 Volunteer Blvd
Knoxville, TN 37996

Abstract

Dataflow programming models have been growing in popularity as a means to deliver a good balance between performance and portability in the post-petascale era. In this paper we evaluate different dataflow programming models for electronic structure methods and compare them in terms of programmability, resource utilization, and scalability. In particular, we evaluate two programming paradigms for expressing scientific applications in a dataflow form: (1) *explicit dataflow*, where the dataflow is specified explicitly by the developer, and (2) *implicit dataflow*, where a task scheduling runtime derives the dataflow using per-task data-access information embedded in a serial program. We discuss our findings and present a thorough experimental analysis using methods from the NWCHEM quantum chemistry application as our case study, and OPENMP, STARPU and PARSEC as the task-based runtimes that enable the different forms of dataflow execution. Furthermore, we derive an abstract model to explore the limits of the different dataflow programming paradigms.

KEYWORDS:
dataflow; task-based runtime; PaRSEC; StarPU; OpenMP; NWChem; coupled cluster methods; CCSD

1 | INTRODUCTION

To this day, the most popular programming model for large-scale computing is coarse grain parallelism (CGP) with explicit message passing, where every exchange of data has to be implemented with carefully hand-crafted send and receive calls to the communication layer. With the profound changes of the hardware landscape, featuring increasing complexity and heterogeneity of today's and projected future high-performance computing (HPC) systems, one of the challenges for the CGP programming model is to sustain the expected performance and scalability of real-world applications on these emerging systems.

Focusing on one such real-world application, this paper targets the field of electronic structure theory—a significant example worth exploring because most computational chemistry methods are already unable to take full advantage of current computer resources at leadership-class computing facilities and are bound to fall behind even further on future post-petascale systems. Here we provide evidence that *dataflow programming models* are well suited for computational chemistry applications and provide a viable way to *achieve and maintain* effective computation and performance at scale—especially on distributed, many-core, heterogeneous architectures.

The main objective is to move away from the concept of developing scientific applications for specific architectures or platforms and instead use dataflow programming models to represent the algorithms in a way that enables us to observe and capture essential properties of the algorithms (e.g., data dependencies). A task-scheduling runtime can then map the algorithms from the directed acyclic graph (DAG) representation to the hardware platforms. It is then the runtime's responsibility to manage

architecture-aware scheduling of the tasks on a broad range of computing platforms. Ultimately, this approach lifts the burden of traditional programming models—namely, "programming at the communication layer"—from the domain scientists, which enables them to focus on the efficiency of the *numerical algorithms* and not *individual implementations* of the algorithms.

In the rest of this paper, we use the term "*tasks*" to refer to non-blocking, pure functions that can be executed by the runtime. That is, functions which have no side-effects to any memory other than arguments passed to the function itself, and whose execution does not block after it has started. Additionally, we use the term "*dataflow*" for the description of how the data is flowing between tasks. The dataflow of a program is used by the runtime to define the ordering of tasks, and can be either implicitly detected by the runtime, or explicitly provided by the developer through a specialized notation.

This work adopts dataflow programming models to the current state-of-the-art NWCHEM coupled cluster methods (CC) (1, 2); where the "explicit" dataflow model takes advantage of the parameterized task graph (PTG) representation (3, 4, 5); and the alternative "implicit" dataflow model uses a task-based representation, where the dataflow is derived by the runtime, and the user does not explicitly define the dataflow as they would for the PTG representation. In this paper, we present various implicit dataflow versions of NWCHEM CC and compare the performance of each runtime. For our detailed analysis of the explicit PTG method, please see (5). We also analyze implicit and explicit dataflow in terms of programmability and performance scalability. Additionally, we derive a theoretical model that investigates the limits of the different dataflow paradigms.

2 | BACKGROUND

The dataflow versions are executed with three runtimes: OPENMP for task-level parallelism that is limited to shared memory, and PARSEC and STARPU, both of which enable task-level parallelism in distributed memory.

The PARSEC framework (6) is a task-based runtime for heterogeneous architectures that employs dataflow models to create dependencies between tasks and exploits the available parallelism present in an application. Here, a task represents any sort of computation. PARSEC builds a DAG of tasks to determine which tasks can run in parallel. PARSEC currently provides two interfaces to express the DAG: the PTG abstraction (the explicit dataflow) and the dynamic-task-discovery (DTD) via the task-insertion interface (the implicit dataflow). In PTG, users have to express the dependency between tasks in a parameterized fashion, and the dataflow is explicit. The task graph is symbolic and can be traversed at runtime without the need to build and store it in memory. The PTG abstraction is the main difference between PARSEC and other task engines (e.g., STARPU), differentiating the way tasks and their data dependencies are represented. In DTD, users insert tasks sequentially, and the runtime infers the dataflow implicitly. This paradigm builds the task graph dynamically during runtime and stores it in memory during the execution. For the end user, PARSEC's consolidation of multiple interfaces into a single runtime makes the challenging job of expressing a parallel application easier and less time consuming.

The STARPU runtime system (7) supports the implicit dataflow programming model. Tasks are submitted to the system serially through `insert_task()` calls that include data-access information that enables the runtime discover the dependencies. Tasks are then automatically executed on the best resource available. One of STARPU's strengths is that the system relies on the variety of scheduling strategies available to adapt to applications and platforms with both centralized and distributed solutions. Recently, support for automatically inferring data communication was added to STARPU to help users moving toward distributed architectures (8).

3 | EXPLICIT DATAFLOW EXECUTION OF COUPLED CLUSTER METHODS

This section describes various design decisions of the *explicit dataflow* version of CC and the associated levels of parallelism and optimizations that have been studied for the PTG-enabled CC implementation.

3.1 | Design Decisions

The original code of the CC subroutines consists of deep loop nests that contain the memory access routines as well as the main computation, namely `SORT` and `GEMM`. In addition to the loops, the code contains several `IF` statements. 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, we decomposed the code into two steps—the discovery phase and the PTG execution—which are described in detail below.

```
1  !
2  DO i
3    DO j
4  !   Validation of loop index values
5      IF (int_mb(k_spin+i) .eq. int_mb(k_spin+j))
6      IF (int_mb(k_sym+j) ...
7        DFILL(C)
8        DO k
9          DO l
10 !         Validation of loop index values
11           IF (int_mb(k_spin+k) .eq. ...
12              GET_HASH_BLOCK_MA(GA,A)
13              SORT(A)
14              GET_HASH_BLOCK(GA,B)
15              SORT(B)
16              DGEMM()
17            END IF
18         END DO
19       END DO
20       SORT(C)
21       ADD_HASH_BLOCK(GA,C)
22
23     END IF
24   END DO
25 END DO
```

```
1  L1_idx = 1
2  DO i
3    DO j
4  !   Validation of loop index values
5      IF (int_mb(k_spin+i) .eq. int_mb(k_spin+j))
6      IF (int_mb(k_sym+j) ...
7        DFILL(C)
8        DO k
9          DO m
10 !         Validation of loop index values
11           IF (int_mb(k_spin+k) .eq. ...
12              task_meta_data(1,L2_idx,L1_idx)=i
13              task_meta_data(2,L2_idx,L1_idx)=j
14              task_meta_data(3,L2_idx,L1_idx)=k
15              task_meta_data(4,L2_idx,L1_idx)=m
16              L2_idx = L2_idx + 1
17            END IF
18         END DO
19       END DO
20       L2_cnt = L2_idx - 1
21       inner_task_count(L1_idx) = L2_cnt
22       L1_idx = L1_idx + 1
23     END IF
24   END DO
25 END DO
```

Figure 1: (a) NWChem CC. **(b)** Populate metadata for PaRSEC version.

Step 1: The Discovery Phase

The first step is the discovery phase, which bridges the original legacy code with PARSEC, 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+i)`) involved in the `IF` branches (lines 4–6 and 10–11 in Figure 1) and then stores the results in custom metadata vectors that we defined. In addition, this step probes the Global Arrays (GA) communication library to discover the physical location of the program data on which the `GEMM`s will operate and stores these addresses in the metadata structures as well. Note that this code is not shown in the figure in an effort to keep the figure small and readable. Since the data of NWCHEM that affects the control flow is immutable at execution time, this first step only needs to be performed once per execution and not for each iteration.

The code that performs this discovery is derived from the original code of each subroutine. Figure 1 (a) shows the pseudocode, generated by the Tensor Contraction Engine (TCE), for one CC sub-kernel, while the pseudocode shown in (b) is derived from the original TCE code to populate all the metadata that PARSEC will use to dynamically derive the data dependencies between the real tasks. The derived code is generated using automatic code transformation tools that we developed. Specifically, in the version depicted in Figure 1 (b), we create a slice of the original code that contains all the control flow statements (i.e., `DO` loops and `IF-THEN-ELSE` branches) but none of the subroutine calls (i.e., `GEMM()`, `SORT()`, `GET_HASH_BLOCK()`, `MA_PUSH_GET()`). Instead, in the place of the original subroutine calls, we insert operations that store the status of the execution into custom metadata arrays that we introduced.

The custom metadata vectors merely hold information about the actual loop iterations that will execute the computational kernels at execution time (i.e., iterations where all the `IF` statements evaluate to `true`). This step significantly reduces the execution space of the loop nests by eliminating all entries that would not have executed. After the first step is finished, the metadata arrays contain all the information necessary for PARSEC to determine which `GEMM` tasks are connected into a chain, the length of each chain, the number of chains, the node that holds the data needed for each `GEMM` task, and the actual pointers for these data.

Below is a list of steps that have been applied for the metadata population:

1. Determine what the tasks are for the CC code:
   `DFILL()`: initialization of an array;
   `SORT_2()`, `SORT_4()`: remapping of the array elements; and
   `DGEMM()`: double precision matrix-matrix multiplication.

2. Dimensions of metadata: the number of dimensions of metadata is defined by ("number of loop levels in which we have tasks"+1). For instance, in `t1_2_2_2`, the number of dimensions is `2+1` because `DIFLL` and `DGEMM` are in different levels.

3. Task metadata:
   `C: task_meta_data[L1_size][L2_size][loop_depth]`
   `FORTRAN: task_meta_data(loop_depth,L2_size,L1_size)`
   For each dimension—except for the `loop_depth`—we have a counter that is incremented at the corresponding level. For example, at the `DFILL`, level we increment `L1`, and at the `GEMM` level we increment `L2`.

The existence of conditionals in the code results in "holes" in the execution space. In other words, not all possible tasks will actually execute, and—as a matter of fact—only a small subset of tasks are truly executed. Since the "real tasks" are a subset of the entire execution space, the metadata-population step is necessary to identify only these real tasks—excluding all remaining tasks. However, this step adds overhead to the execution time, which in Section 6—where we model this behavior—is referred to as $T_O^{(2)}$.

```
GEMM(L1, L2)
  L1 = 0..(mtdata->size_L1-1)
  L2 = 0..(mtdata->size_L2-1)

  A_reader = find_last_segment_owner(mtdata, 0, L2, L1)
  B_reader = find_last_segment_owner(mtdata, 1, L2, L1)

  : descRR(L1)

  READ   A <- A input_A(A_reader, L2, L1)
  READ   B <- B input_B(B_reader, L2, L1)

  RW C <- (L2 == 0) ? C DFILL(L1)
       <- (L2 != 0) ? C GEMM(L1, L2-1)
       -> (L2 < (mtdata->size_L2-1))  ? C GEMM(L1, L2+1)
       -> (L2 == (mtdata->size_L2-1)) ? C SORT(L1)

; mtdata->size_L1-L1 + P
BODY {
  dgemm('T', 'N', ...
}
```

Figure 2: PTG for `GEMM` tasks organized in a chain.

Step 2: The Parameterized Task Graph

The second step is the execution of the PTG representation of the subroutines, which can be understood as a compressed representation of the DAG that describes the execution of a task-based application. Since the control flow depends on the program data, the PTG examines the custom metadata vectors populated by the first step; this allows the execution space of the modified subroutines over PARSEC to match the original execution space of these subroutines. Also, using the metadata structures,

which were populated by probing the communication library (GA), PARSEC accesses the program data directly from memory without making GA communication calls to transfer the data during execution.

The example in Figure 2 will be used to outline some important aspects of the PTG representation. This code snippet defines tasks as a chain of GEMMs. These tasks are parameterized using the parameters L1 and L2. In fact, this PTG mimics the original CC code, which has the GEMM operations organized into multiple parallel chains, with each chain containing multiple GEMMs that execute sequentially. In this PTG, L1 corresponds to the chain number, and L2 corresponds to the position of a GEMM inside the chain to which it belongs. The number of chains and the length of each chain do not have to be fixed numerical constants. PARSEC will dynamically look them up from the metadata vectors that have been filled by the discovery phase during the first step. Also, the PTG allows for calls to arbitrary C functions for dynamically discovering information, such as the nodes from which the input data must be received.

By looking at the dataflow information of matrix C (lines 13-16), one can see the chain structure. The first GEMM task (L2==0) receives matrix C from the task DFILL (which initializes matrix C), and all other GEMM tasks receive the C matrix from the previous GEMM task in the chain (GEMM(L1, L2-1)) and send it to the next GEMM task in the chain (GEMM(L1, L2+1)), except for the last GEMM, which sends the C matrix to the SORT task.

This representation of the algorithm does not resemble the form of familiar CGP programs, but the learning curve that must be climbed comes with rewards for those who climb it. In order to change the organization of the GEMMs from a serial chain to a parallel execution followed by a reduction, the four lines that define the dataflow of matrix C (13–16) would have to be replaced by the following single line depicted in Figure 3. In addition, PTG code that implements a reduction must be included in the PTG of the program, but a binary tree reduction is a standard operation, and so the code can be copied from another PARSEC program or example code.

```
1  WRITE C -> A REDUCTION(L1, L2)
```

Figure 3: PTG snippet for parallel GEMM tasks.

Owing to the nature of the PTG representation, all communication becomes implicit and is handled automatically by the runtime without user intervention. This is in contrast with more traditional message passing programming paradigms. In an MPI program, for instance, the developer has to explicitly insert, in the source code, a call to a function that performs each data transfer, and when non-blocking communication is used, the developer has to explicitly preallocate buffers for the data of future operations, pre-post non-blocking communication operations that populate these buffers (i.e., MPI_Irecv() and MPI_Isend()), and manage these non-blocking operations (i.e., test or wait for their completion), which quickly becomes logistical overhead. Even if abstraction layers are used over the communication library, as is the case in NWCHEM, the developer still has to explicitly insert, in the source code, calls like GET_HASH_BLOCK() (see Figure 1[a]).

3.2 | 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. PARSEC also has the ability to decouple work tasks and communication operations, which enables us to experiment with more advanced communication patterns than the serial chains. A GEMM kernel performs the operation:

$$C \leftarrow \alpha \times A \times B + \beta \times C,$$

where A,B, and C are matrices, and α and β are scalar constants. In the chain of GEMMs performed by the original code, the result of each matrix multiply is added to the result of the previous matrix multiply, since $\beta = 1$. Also, since matrix addition is an associative and commutative operation, the order in which the GEMMs are performed does not bear great significance,[1] as long as the results are added atomically. This method enables us to perform all GEMM operations in parallel and sum the results using a binary reduction tree. Figure 4 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 (9). For the sake of completeness, Figure 5 depicts such a chain where eight GEMM operations are computed sequentially.

[1]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.

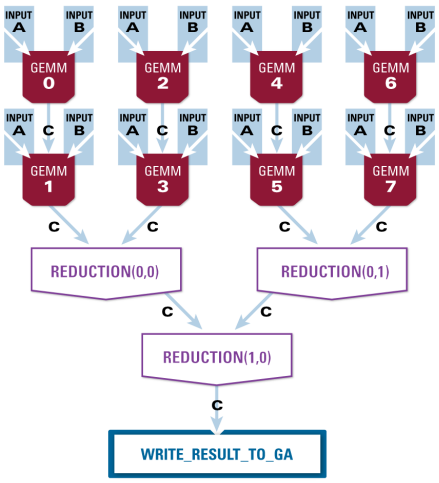

Figure 4: Parallel GEMM operations followed by reduction.

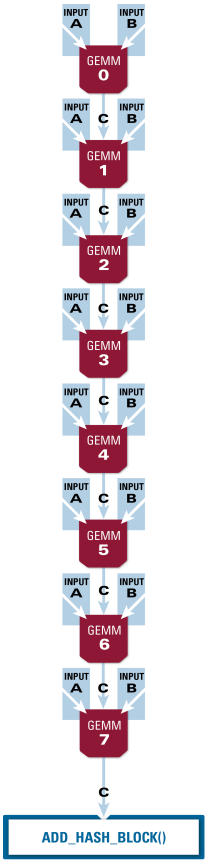

Figure 5: Chain of GEMM operations computed sequentially.

In addition to the number of sequential steps being lower when a binary tree reduction is used, the steps themselves 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 the number of sequential steps performed by the chain and the binary tree grows fast. We should note that the reductions discussed here only apply to `GEMM` operations that execute on the same node, thus no additional communication is necessary to implement this optimization.

It is also important to note that the original version of the code uses functionality from the GA library to perform a **global** (i.e., across the whole distributed memory allocation) atomic accumulate-write operation (via calls to `ADD_HASH_BLOCK()`) at the end of each chain. Since our dataflow version of the code has knowledge of the memory location where the data resides (through the discovery phase that we discussed earlier), we eliminate the **global** atomic GA functionality and perform direct memory access instead using **local** atomic locks (i.e., pthread mutexes and low-level atomic memory operations) within each node to prevent race conditions. The choices made in the implementation discussed in this chapter are based on earlier studies presented in (10, 11). In these earlier investigations we explored multiple different options for the explicit dataflow version of the code. In the work presented in this paper we selected the best explicit dataflow version from our previous work and compared it against multiple different options for the implicit dataflow version of the code.

The original CC implementation treats the entire chain of `GEMM` operations as one "task" and therefore assigns it to one node. Our new implementation over PARSEC distributes the work onto different hardware resources, which leads 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 (9).

4 | IMPLICIT DATAFLOW EXECUTION OF COUPLED CLUSTER METHODS

This section describes the design decisions of the *implicit dataflow* version of CC and the associated levels of parallelism and optimizations that have been studied for the `Insert_Task()`-enabled CC implementation. While the *explicit dataflow* model with its PTG-enabled CC implementations is only supported by the PARSEC runtime, the *implicit dataflow* model can be used by any runtime a user chooses. That is, any runtime that supports the serial task-insertion interface can be use to submit tasks and generate the DAG of the execution, thereby not requiring developers to use the PARSEC.

For this study, the `Insert_Task()`-enabled CC implementations have been executed with three runtimes: PARSEC, STARPU, and OPENMP. The performance results of these experiments are compared against the performance results from the original NWCHEM CC code and the PTG-enabled PARSEC implementation.

4.1 | Implicit Dataflow and Programmability

The development of an *implicit dataflow* version of CC is less cumbersome on the application developer, in the sense that it is more accessible with the natural control flow of the original CC implementation.

The implicit programming model expresses parallelism by submitting tasks in the sequential flow of the program. The task-based runtime schedules the tasks asynchronously at execution time, derives the data dependencies between tasks, builds the DAG of the tasks in memory, and then executes the graph of tasks. Such a programming model, which is based on sequential task submission, is particularly attractive for application scientists, because they can use the long-established features from the original control flow of the program without diverging too much into a different representation of the algorithm. However, one negative aspect of this approach is that overheads of the original code—such as unnecessary delays due to suboptimal blocking communication, or overheads due to load balancing—are inherited by the *implicit* dataflow versions. Later on, in Section 6, where we model this behavior, we refer to these overheads as $T_O^{(3)}$.

In terms of programmability, the implicit dataflow enables domain scientists to take an extensive, multi-million line application—like NWCHEM, which has a user base of more than 150,000 users—and express additional parallelism by simply submitting tasks in the sequential flow of the program. This is not possible for explicit dataflow. For instance, as we mentioned earlier, our explicit dataflow version of CC required a human developer to provide a PTG representation of the dataflow of the code and also required a metadata discovery phase to execute before the execution of the actual program. While the metadata discovery phase is akin to the task submission process that takes place in implicit dataflow programming models, the development of the PTG representation of the dataflow increases the complexity of code development significantly and diverges from the original program. In some cases, even the PTG creation can be automated (by a polyhedral analysis compiler tool (12)), but these cases are limited to highly-regular affine codes, and NWCHEM includes functions with side effects and statically undecidable branches, and so it is far from being affine and out of reach for current compiler technologies.

4.2 | Insert_Task() Version

The implicit dataflow version of CC that is based on the serial task-insertion application programming interface (API) does not require a discovery phase. Instead, `Insert_Task()` calls can be placed directly in the appropriate place in the original CC FORTRAN code. Figure 6 shows the code snippet of one original CC subroutine (left) and compares it against the modified version (right).

The new version replaces the original call to `GEMM` (line 13) with a call to an `Insert_task_core()` subroutine (the body of which is shown in Figure 7) that takes mostly the same parameters as the original `GEMM` function. Note that the parameter change for the scalar *beta* (highlighted in orange on line 13) from `1.0d0` in the original code to `0.0d0` for the `Insert_Task()` version. In the original code, each output from a `GEMM` in a chain is added to the previous $C \leftarrow alpha \times A \times B + beta \times C$, which requires the C matrix to be set on input. However, since there is no other data dependency between `GEMM`s that belong to the same chain, they can be computed independently from each other. For the implicit dataflow version, each `Insert_task_core()` submits an individual `GEMM` task with its own C, and the runtime can schedule and execute these tasks in parallel. The code snippet of the `Insert_task_core()` subroutine is shown in Figure 7.

```
1  my_next_task = SharedCounter()
2  DO i
3   DO j
4  ! Validation of loop index values
5    IF ...
6     DFILL(dimc, dbl_mb(k_c),...)
7     DO k
8      DO l
9  !     Validation of loop index values
10      IF ...
11       GET_HASH_BLOCK(GA,A)
12       GET_HASH_BLOCK(GA,B)
13       call DGEMM('T','N',...,1.0d0,
          ↪  dbl_mb(k_c),...)
14      END IF
15     END DO
16    END DO
17
18    ADD_HASH_BLOCK( dbl_mb(k_c),...)
19    my_next_task = SharedCounter()
20   END IF
21  END DO
22 END DO
```

```
1  my_next_task = SharedCounter()
2  DO i
3   DO j
4  ! Validation of loop index values
5    IF ...
6     DFILL(dimc, dbl_mb(k_c),...)
7     DO k
8      DO l
9  !     Validation of loop index values
10      IF ...
11       GET_HASH_BLOCK(GA,A)
12       GET_HASH_BLOCK(GA,B)
13       call INSERT_TASK_CORE(...,0.0d0,
          ↪  k_c,...)
14      END IF
15     END DO
16    END DO
17    call INSERT_TASK_WAIT()
18    ADD_HASH_BLOCK( dbl_mb(k_c),...)
19    my_next_task = SharedCounter()
20   END IF
21  END DO
22 END DO
```

Figure 6: Original CC subroutine code **(left)** compared to the Insert_Task() version **(right)**.

Once the GEMM operation is completed, each task adds its local computed array, c_tmp, to the array that holds the final result for the whole chain, dbl_mb(k_c),[2] which belongs to the main thread. This update is shown in lines 6–10 of Figure 7. The update uses mutual exclusion (a mutex) to protect simultaneous access to the dbl_mb(k_c) array when multiple threads try to add their local c_tmp array to it.

The dbl_mb(k_c) array (highlighted in green in the code snippets) is initialized by the main thread at the beginning of each chain of GEMMs (line 6 in Figure 6). Once all GEMMs that belong to the same chain in the original code are submitted, there is a barrier placed at the end of each chain (line 17 in Figure 6). Even though the runtime can schedule and execute all GEMMs in parallel, because of the ADD_HASH_BLOCK() operation (in line 18 in Figure 6), all GEMM tasks need to be completed so that the final dbl_mb(k_c) array can be added to the GA.

At this point, it is important to point out that a significant amount of time was spent on an attempt to move the GET_HASH_BLOCK() calls, which transfer the matrix A and B data over the network using the global arrays (13) library (line 11–12 in Figure 6), into the task together with the GEMM operation. This would result in a more efficient execution. First, it would put less burden on the main thread since, in the current implementation, it is the main thread's sole responsibility to fetch data from the GAs into local memory. Second, it would add additional overlap of communication and computation as individual tasks do not depend on GA data from other tasks that belong to the same chain. Hence fetching data from GA for each GEMM task could (in theory) be performed in parallel. However, it is currently not possible to implement this design decision since the implementation of the GET_HASH_BLOCK subroutine is currently not thread safe.

[2]The syntax "dbl_mb(k_c)" might suggest an array element instead of an array, but it is an actual array, and the uncommon form is due to the idiosyncratic memory management of NWChem, of which we have no control.

```
recursive SUBROUTINE insert_task_core(...)
  ...
  double precision, dimension(:), pointer :: c_tmp
  allocate(c_tmp(dimc))
  call DGEMM('T','N',..., c_tmp , ...)
  call lock_mutex()
  DO i = 0, dimc-1
     dbl_mb(k_c+i) = dbl_mb(k_c+i) + c_tmp(1+i)
  END DO
  call unlock_mutex()
  deallocate( c_tmp )
  RETURN
END
```

Figure 7: Insert_task_core() subroutine that is submitted to the runtime.

4.3 | Design Variations for OpenMP Version

The OpenMP specification (14) is a well-known standard for developing parallel shared-memory applications. OpenMP 3.0 (15) introduced a feature called *tasking*, which adds support for task-level parallelism. Three different design variations for the OpenMP task version of the CC code were implemented and compared. Before describing the differences between these alternative versions, their common code blocks are described.

In OpenMP, an *explicit task* is a unit of work that may be executed at some point in time, either immediately or later. The decision of when a task runs is taken up by the OpenMP runtime. A task is composed of three parts: (1) the code to be executed, (2) the data environment associated with the task, and (3) the internal control variables. Additionally, an OpenMP `task` is specified using the `task` construct with its associated clauses (`shared`, `private`, `firstprivate`). For tasks, the default data-sharing attribute is `firstprivate`, but the `shared` attribute is lexically inherited. This is different from other OpenMP constructs like `parallel` or `single`. In the majority of cases, the default data-sharing attribute is `shared`. The `firstprivate` clause captures the values of the variable at the time of a task's creation. This is particularly important for tasks, because the OpenMP runtime may choose to run a task at a later point in time. However, the values that are captured when the task is created are actually the values that we want for the execution of the task, regardless of when the task executes.

Figure 8 compares the pseudocode for two of the three OpenMP CC versions. The `task` construct can be placed anywhere in the program, and whenever a thread encounters a task construct, a new task is generated. In the OpenMP task version of CC, the call to a `GEMM` is wrapped with `!OMP TASK` and `!OMP END TASK`, which specify a single `GEMM` operation (and corresponding memory management) as one task (lines 15–24 in Figure 8).

However, the `task` directive by itself does not enable the parallel execution of tasks. Instead, a `parallel` region needs to be created first, which creates a team of threads (line 7). Additionally, the `single` construct is used (line 8) to guarantee that only one thread of the team creates the tasks that are then added to the task pool. Any thread from the team can then execute tasks, and with multiple threads, these tasks are computed in parallel. Having the `parallel` region outside each chain of `GEMMs` (but inside the `DO` loops that execute over multiple chains), means that the thread that enters the `single` region generates work while others might execute tasks from the task pool.

Once all `GEMM` tasks for a chain are generated, the code exits the `single` and `parallel` region (lines 28–29) in order to move on to the next chain of `GEMMs`. The end of a `single`, and also the end of a `parallel` region, features an implicit barrier. This is the heaviest form of synchronization that waits for all threads to be completed before moving on. This implicit barrier—which is equivalent to the `INSERT_TASK_WAIT()` calls in Figure 6—is necessary for the same reason as mentioned in the previous section. All `GEMM` tasks, belonging to the same chain, need to be completed so that the final `dbl_mb(k_c)` array can to be added to the GA using the `ADD_HASH_BLOCK()` operation (line 30 in Figure 8).

```
1  my_next_task = SharedCounter()
2  DO i
3   DO j
4  ! Validation of loop index values
5    IF ...
6     DFILL(dimc, dbl_mb(k_c) ,...)
7  !OMP PARALLEL PRIVATE(...)
8  !OMP SINGLE
9     DO k
10     DO l
11 !     Validation of loop index values
12       IF ...
13        GET_HASH_BLOCK(GA,A)
14        GET_HASH_BLOCK(GA,B)
15 !OMP TASK PRIVATE(c_tmp,h)
16        allocate(c_tmp(dimc))
17        call DGEMM('T','N',..., 0.0d0 , c_tmp ,...)
18        DO h=0,dimc-1
19 !OMP ATOMIC
20        dbl_mb(k_c+h)=dbl_mb(k_c+h)+c_tmp(h+1)
21        END DO
22
23        deallocate(c_tmp)
24 !OMP END TASK
25       END IF
26      END DO
27     END DO
28 !OMP END SINGLE
29 !OMP END PARALLEL
30     ADD_HASH_BLOCK( dbl_mb(k_c) ,...)
31     my_next_task = SharedCounter()
32    END IF
33   END DO
34 END DO
```

```
1  my_next_task = SharedCounter()
2  DO i
3   DO j
4  ! Validation of loop index values
5    IF ...
6     DFILL(dimc, dbl_mb(k_c) ,...)
7  !OMP PARALLEL PRIVATE(...)
8  !OMP SINGLE
9     DO k
10     DO l
11 !     Validation of loop index values
12       IF ...
13        GET_HASH_BLOCK(GA,A)
14        GET_HASH_BLOCK(GA,B)
15 !OMP TASK PRIVATE(c_tmp,h)
16        allocate(c_tmp(dimc))
17        call DGEMM('T','N',..., 0.0d0 , c_tmp ,...)
18 !OMP CRITICAL
19        DO h=0,dimc-1
20        dbl_mb(k_c+h)=dbl_mb(k_c+h)+c_tmp(h+1)
21        END DO
22 !OMP END CRITICAL
23        deallocate(c_tmp)
24 !OMP END TASK
25       END IF
26      END DO
27     END DO
28 !OMP END SINGLE
29 !OMP END PARALLEL
30     ADD_HASH_BLOCK( dbl_mb(k_c) ,...)
31     my_next_task = SharedCounter()
32    END IF
33   END DO
34 END DO
```

Figure 8: Pseudocode of OPENMP atomic **(left)** and critical **(right)** versions.

The three different design variations that were implemented and studied for the OPENMP task version of the CC code are shown in Figure 8 and Figure 9. The difference between these alternative versions is in the way the *C* arrays from the GEMM computations are added to the `dbl_mb(k_c)` array that belongs to the main thread. Like in the `Insert_Task` version, this operation needs to be protected from simultaneous access to the `dbl_mb(k_c)` array by multiple threads. This can be accomplished in different ways with the same effect on `dbl_mb(k_c)` but with different results on the performance.

Atomic: The version in Figure 8 (left) uses an OPENMP `ATOMIC` operation (lines 19–20), which protects only the single assignment that immediately follows it. An `ATOMIC` operation provides mutual exclusion but only applies to the load/update of a memory location. More precisely, `ATOMIC` only protects the update of `dbl_mb(k_c)`. Using this directive enables different threads to update different parts of `dbl_mb(k_c)` simultaneously.

Critical: The version in Figure 8 (right) uses an OPENMP `CRITICAL` region, which surrounds the entire `DO` loop that adds the `c_tmps` array to `dbl_mb(k_c)` (lines 18–22). In OPENMP, all the unnamed critical regions are mutually exclusive. This form of exclusivity forces the update of `dbl_mb(k_c)` to be fully serial, so it was initially surprising that this implementation resulted in the best performance (see Section 5).

Buffered: The version in Figure 9 creates a temporary *C* array for each OPENMP thread. This means that there are "number_of_omp_threads" temporary *C* arrays, and they are all initialized to zero (lines 7–8). Each thread adds the result from different GEMM tasks (that it picks from the task pool) to its local *C* array (note that $\beta = 1$ for the GEMM in line 18). After the parallel region, the main thread adds all the temporary *C* arrays to dbl_mb(k_c) (lines 25–27). It also deallocates the temporary *C* arrays that were created for each thread (line 28). With this version, an OPENMP task (line 17–19) is a pure GEMM operation without any atomic operation or critical region in the task block itself.

For all three OPENMP task variations, the tasks are tied to threads, which is the default unless the UNTIED clause is specified. This prevents tasks being switched from one thread to another, which, for the two implementations, would be a formula for disaster owing to the use of thread-id and critical regions in the current tasks. In addition, threads can be bound to CPU cores for performance reasons. Forcing threads to run on a specific processor core may be beneficial in order to take advantage of local process states. The OMP_PROC_BIND environment variable provides some control over thread binding. Table 1 summarizes its different settings. All three implementations were run and compared against each of the four OMP_PROC_BIND options.

```
1  my_next_task = SharedCounter()
2  DO i
3   DO j
4  ! Validation of loop index values
5    IF ...
6     DFILL(dimc, dbl_mb(k_c),...)
7     allocate(c_tmps(dimc,my_omp_nthreads))
8     c_tmps(:,:)=0.0d0
9  !OMP PARALLEL PRIVATE(...)
10 !OMP SINGLE
11    DO k
12     DO l
13 !     Validation of loop index values
14       IF ...
15        GET_HASH_BLOCK(GA,A)
16        GET_HASH_BLOCK(GA,B)
17 !OMP TASK PRIVATE(my_omp_tid)
18       call DGEMM('T','N',...,1.0d0, c_tmps(1,my_omp_tid+1), ...)
19 !OMP END TASK
20       END IF
21     END DO
22    END DO
23 !OMP END SINGLE
24 !OMP END PARALLEL
25    DO h = 1, my_omp_nthreads
26       dbl_mb(k_c:k_c+dimc-1) = dbl_mb(k_c:k_c+dimc-1) + c_tmps(:,h)
27    END DO
28    deallocate(c_tmps)
29    ADD_HASH_BLOCK( dbl_mb(k_c),...)
30    my_next_task = SharedCounter()
31   END IF
32  END DO
33 END DO
```

Figure 9: Pseudocode of buffered OPENMP task implementation.

OMP_PROC_BIND	Description
TRUE	The execution environment should not move OPENMP threads between processor cores. Threads are bound in a linear way, that is: thread 0 to CPU 0, thread 1 to CPU 1, etc. (unless GOMP_CPU_AFFINITY is set).
FALSE	The execution environment may move OPENMP threads between processor cores. This is also the case if OMP_PROC_BIND is not set (default bahavior).
CLOSE	Binds threads as close to the main thread as possible.
SPREAD	Binds threads as evenly distributed as possible among the processing cores.

Table 1: Thread binding options with OMP_PROC_BIND environment variable.

5 | PERFORMANCE EVALUATION FOR CC

This section analyzes the performance of the entire CC code using the various implementations of the "implicit" dataflow version of the CC subroutines and contrasts it with the performance of the original code and of the "explicit" dataflow version that was discussed in 3.

5.1 | Methodology

As input, we used the beta-carotene molecule ($C_{40}H_{56}$) in the 6-31G basis set, which was composed of 472 basis set functions. In our tests, we kept all core electrons frozen and correlated 296 electrons.

The subroutines, we targeted for our "implicit" and "explicit" dataflow conversion effort belong to a set of 29 CCSD T2 subroutines that comprise approximately 91% of the execution time of all 41 T2 subroutines. This is the same set of subroutines that was identified in the statistics in (5) as the computationally most expensive work when computing the CC correlation energy.

The performance tests were performed on the "Cascade" computer system at Pacific Northwest National Laboratory's (PNNL's) Environmental Molecular Sciences Laboratory (EMSL). 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 scalability tests, utilizing 1, 2, 4, 8, and 16 cores per node. NWCHEM version 6.6 was compiled with the Intel 15.0.0 compiler using the Math Kernel Library (MKL) 14.0, an optimized Basic Linear Algebra Subprograms (BLAS) library, provided on Cascade.

5.2 | Discussion

For the performance evaluation, all graphs discussed in this section show the execution time of the entire CC when the implementation found in the original NWCHEM code is used and when the dataflow implementations are used for the (earlier mentioned) 29 CCSD 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 has 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, and so the percent error is on the order of 10^{-12}%. The graphs depict the behavior of the original code using the black line and the behavior of the dataflow implementations using bars. Once again, the execution times of the dataflow runs do not exclude any steps performed by the modified code.

5.2.1 | OpenMP Tasks

All three CC implementations with OPENMP have been run and compared against each of the four OMP_PROC_BIND options. For each case, the CC execution for runs on 32 nodes performs best when the OPENMP threads are bound to processing cores and are as evenly distributed as possible among the cores (OMP_PROC_BIND=SPREAD). Consequently, for the analysis of the three different versions shown in Figure 10, we compare the performance of the three OPENMP versions with OMP_PROC_BIND=SPREAD.

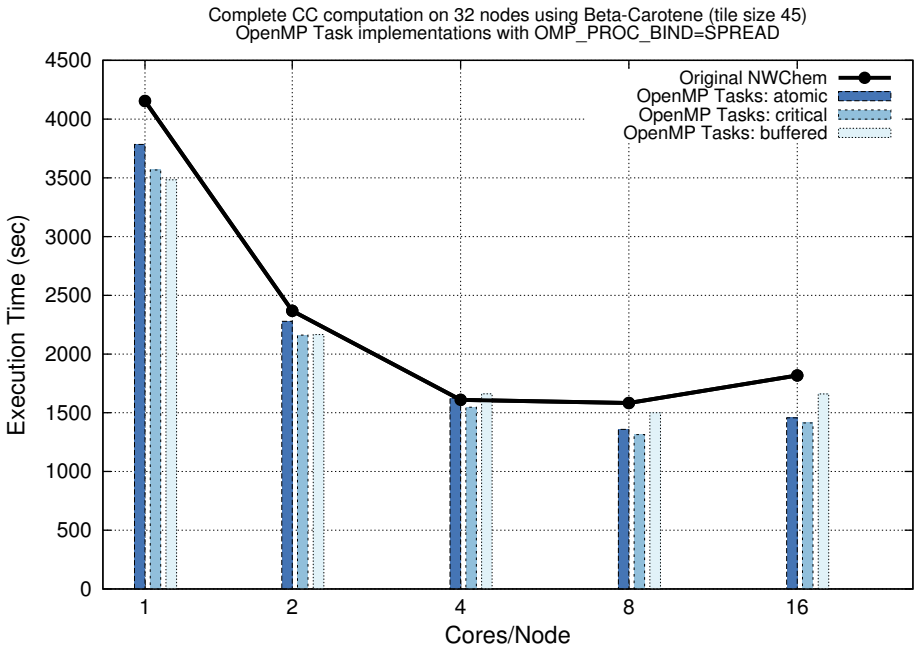

Figure 10: Comparison of CC execution time of all three OpenMP implementations.

The implementation that uses a `CRITICAL` region (to protect the addition of `c_tmp` to the `dbl_mb(k_c)`) performs best. Intuitively, one could expect a `CRITICAL` region, which serializes the update operation, to be outperformed by a loop with an `ATOMIC` update, which enables multiple threads to simultaneously access different parts of the array. On the other hand, the `ATOMIC` update incurs an overhead at every single memory access, whereas the `CRITICAL` region operates at the memory bandwidth. The interplay between the different overheads makes the choice between the different approaches a non-trivial one—a decision possibly dependent on the architecture and the OpenMP compiler. For that reason, we tested both approaches to find the one that performs best.

In the third version, referred to as "buffered", each OpenMP task accumulates results into a per-thread copy of (part of) `dbl_mb`; once all tasks are completed, the main thread accumulates these per-thread results. This implementation performs even worse and is outperformed by the "critical" and the "atomic" version of the CC code. This is most likely due to the size of the arrays. From the statistics in (5), one can derive that a typical value of `dimc` ranges from `1,225,449` to `807,003`, resulting in fairly big `c_tmps` arrays. In that case, it is probably worth parallelizing both the initialization and the reduction of the per-thread copies. This may not vectorize well owing to the non-unit stride on `c_tmps`, depending on how smart the compiler is, but since `dimc` is so big, one could try blocking this loop nest. These further optimizations for the "buffered" version will be left for future work.

5.2.2 | Insert_Task()

For the `Insert_Task()` version of the NWChem CC's implicit dataflow implementation, two different runtimes—StarPU and PaRSEC, which both support the serial task-insertion interface—have been used. Specific details about how these runtimes differ is beyond the scope of this work. However, because both runtimes use a different "default" scheduler, the performance of three schedulers was evaluated, and the scheduler delivering the best performance was chosen for subsequent performance tests. Figure 11 shows the differences in the execution time for runs on 32 nodes when StarPU is configured with different schedulers:

lws: The locality work stealing (lws) scheduler uses one queue per worker and schedules a task on the worker that released it by default. When a worker becomes idle, it steals a task from neighbor workers.

ws: The work stealing (ws) scheduler is similar to the **lws** but steals a task from the most loaded worker.

eager: The eager scheduler uses a central task queue from which all workers draw tasks to work on concurrently.

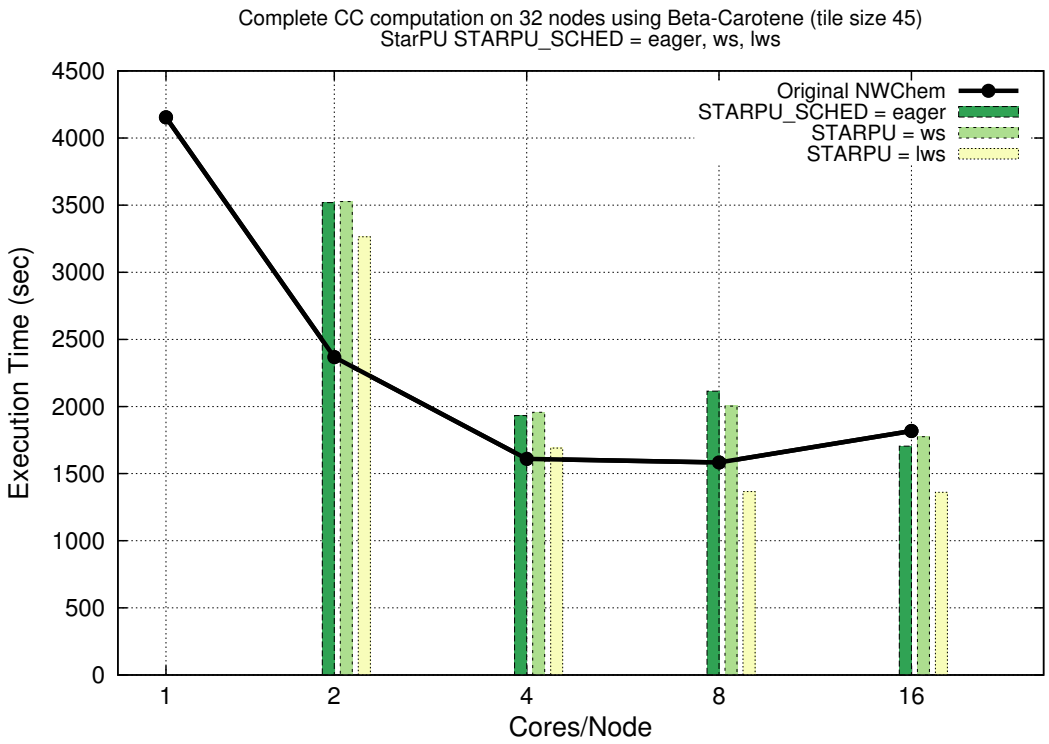

Figure 11: Comparison of CC execution time with StarPU using different schedulers.

Let us recall that the implicit dataflow implementations take advantage of the original control flow of CC, where all `GEMMs` that belong to a chain are computed serially. In the implicit dataflow versions, these `GEMM` tasks are processed in a parallel fashion on a node. The number of parallel tasks per node, at any given time, is equal to the size of the chain, which is executing at that time. For beta-carotene, chain sizes are less than or equal to 16. That means, on a machine with a maximum of 16 cores per node, there will be on average less tasks than cores. Therefore, one may assume that, with the eager scheduler, threads are constantly hitting the global task queue, which requires a lock to be acquired and released each time—thereby causing an overhead that is avoided with the other schedulers that do not use a global queue.

Figure 11 confirms that the **eager** scheduler results in the most expensive execution of CC, while the **lws** scheduler performs best. Thus, both runtimes, StarPU and PaRSEC, have been configured to use **lws** for all subsequent experiments.

5.3 | Results

Performance studies were performed on 32 nodes (see Figure 12), 64 nodes (see Figure 13), and 128 nodes (see Figure 14). Each of these graphs depicts the results of the three best-performing implicit dataflow versions, the explicit dataflow version, and the original CC code.

Note that in the case of two cores per node with the explicit dataflow version (PTG) and the implicit dataflow version with StarPU, both runtimes (PaRSEC [PTG] and StarPU) dedicate one thread for communication, which is the cause of the poor performance. In other words, even though two cores per node are allotted, only one worker thread is used for the computation and execution of all tasks, which results in a performance that is comparable to the versions using one core per node.

Overall, both dataflow versions outperform the original code on all three node partitions. While the speedup of the explicit method over the original method decreases, the speedup of the implicit method over the original method increases as the number of nodes grows from 32 to 128. Table 2 lists the exact speedups for each node partition.

Additionally, we compare the two dataflow versions to one another. While the performance of the three implicit dataflow versions is on par, which is expected since the runtimes operate similarly, there is a significant difference between the performance of explicit versus implicit. The implicit version always performs best on 8 cores per node, while explicit manages to use an increasing number of cores—all the way up to 16 cores per node—to improve the performance. This behavior is not surprising for a number of reasons, detailed below.

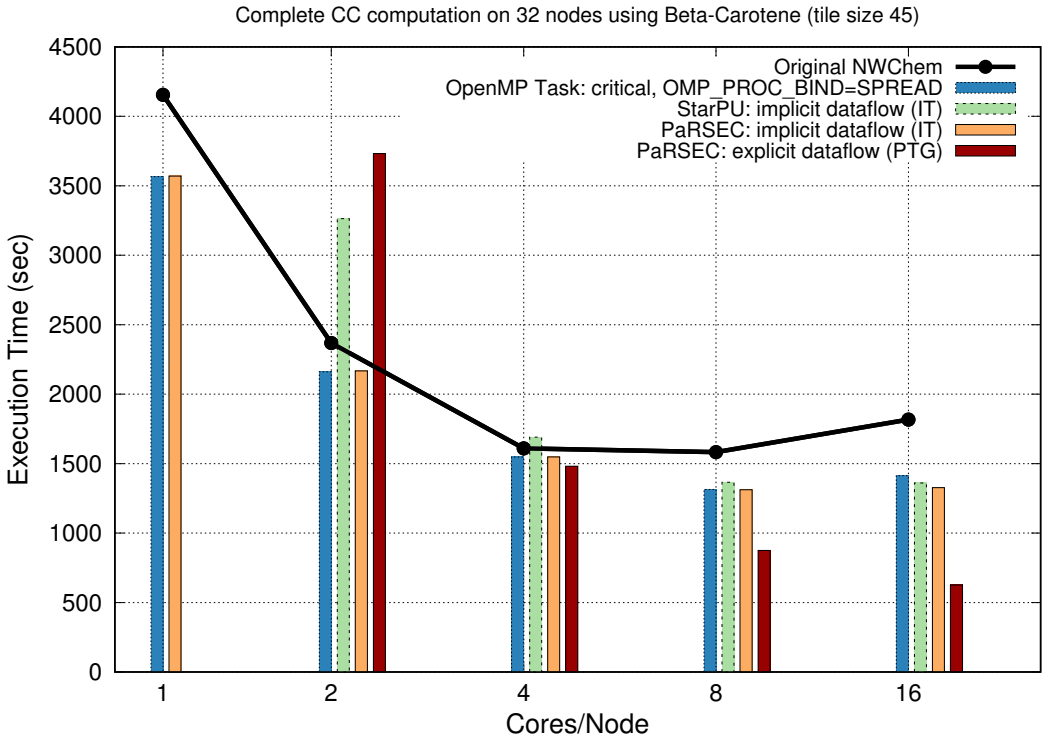

Figure 12: CC Execution time comparison using 32 nodes (using up to 512 cores) on EMSL/PNNL Cascade.

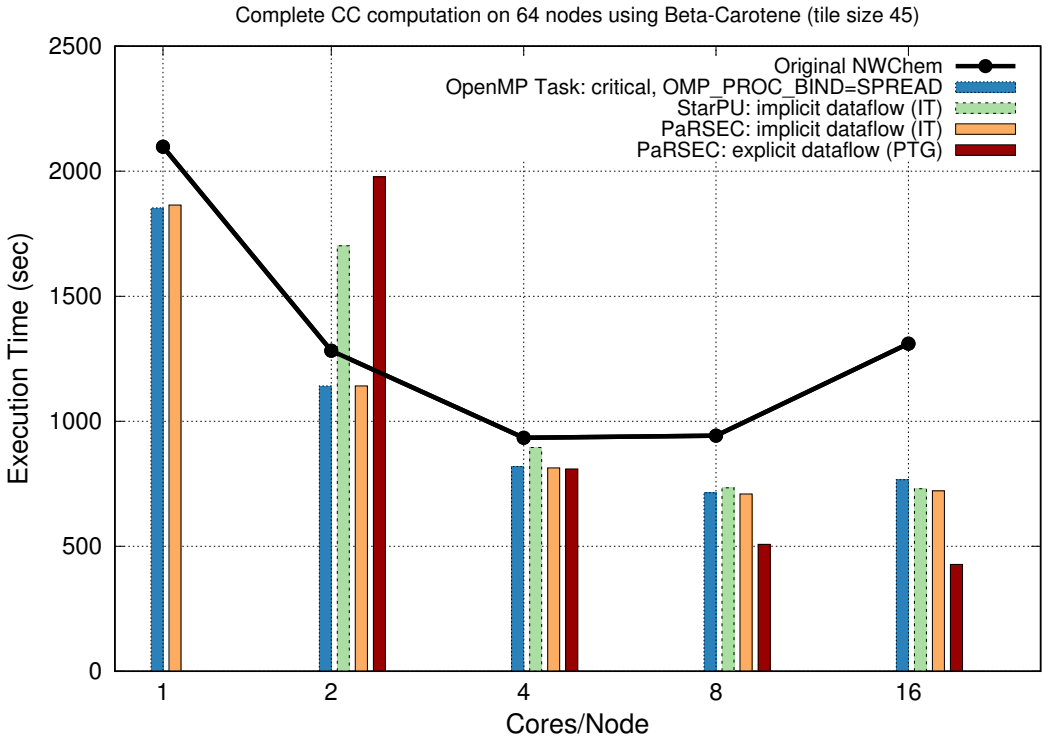

Figure 13: CC Execution time comparison using 64 nodes (using up to 1,024 cores) on EMSL/PNNL Cascade.

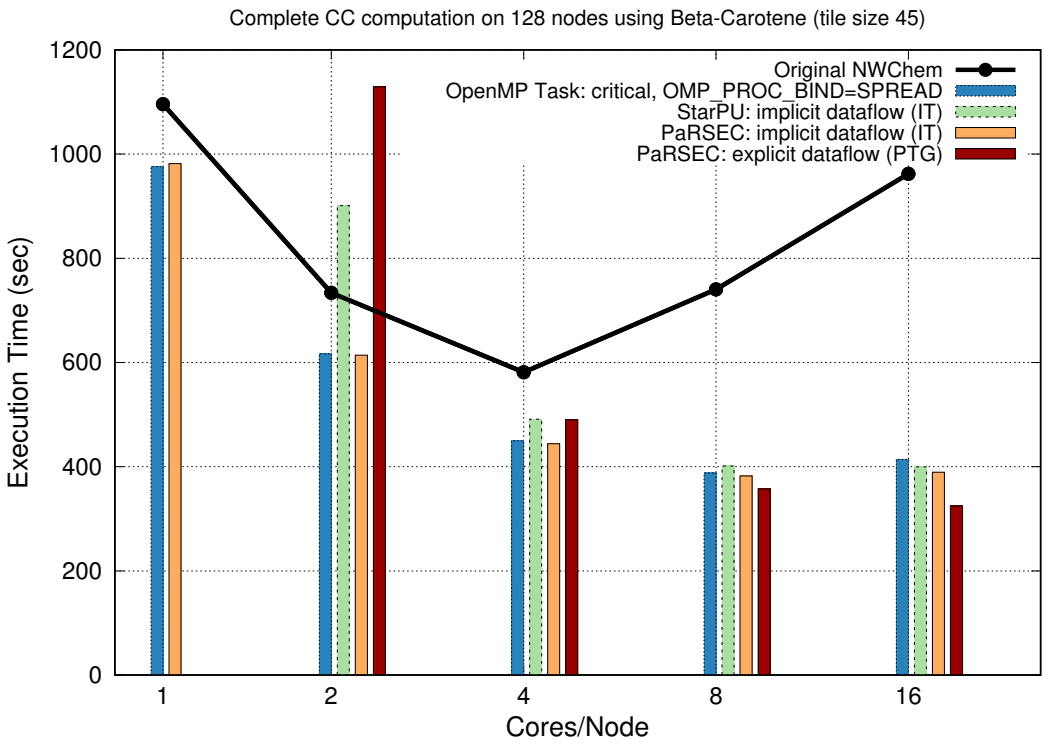

Figure 14: CC Execution time comparison using 128 nodes (using up to 2,048 cores) on EMSL/PNNL Cascade.

Number of Nodes	$Speedup = \frac{T_{orig}}{T_{expl}}$	$Speedup = \frac{T_{orig}}{T_{impl}}$
32	2.6	1.2
64	2.2	1.3
128	1.8	1.5

Table 2: Speedup for dataflow versions over original CC.

1. As mentioned earlier, for the implicit dataflow versions, the number of parallel tasks per node, at any given time, is equal to the size of the chain that executes at that time. For beta-carotene, chain sizes are either 8 or 16 for the most expensive subroutines, and smaller (e.g., 4, 6, 8) for less expensive subroutines. Therefore, one of the limiting factors of why the implicit versions stagnate after 8 cores per node is due to the average chain size being ≈ 6 and the weighted average—using the size of `GEMM` as the weight—being ≈ 11. In other words, when the average chain size $\frac{N}{N_{chain}}$ is smaller than the number of cores per node, C, the available parallelism is bound by the chain size. As a result, when modeling this behavior in Section 6, we can express the maximum available parallelism (per node) of implicit dataflow models as $min(\frac{N}{N_{chain}}, C)$, as opposed to the simple scalar C.

2. The implicit code inherits the limitations from the original code, which uses a global atomic variable for load balancing. The explicit version, however, distributes the work in a round-robin fashion and avoids any kind of global agreement in the critical path.

3. Furthermore, the implicit code inherits the control flow of the original code, which calls functions that transfer data over the network via the GA layer. This communication is blocking and leaves no room for communication and computation to overlap within the same thread of the execution. On the other hand, the explicit dataflow model uses non-blocking communication and allows for communication-computation overlap.

When moving from 32 to 64 to 128 nodes, both dataflow versions continue to show speedups, but the implicit dataflow version converges closer to the performance of the explicit dataflow version. On the 32 node partition, explicit outperforms implicit by a factor of two, while the speedup of explicit over implicit drops to 1.7 on 64 nodes and 1.2 on 128 nodes. This is in agreement with our theoretical model, which we discuss in the next section, and where we show that for a large number of nodes P, the speedup of the *explicit* CC code, in comparison to the *implicit* code, will converge to the ratio of their constant overheads $\frac{T_O^{(3)}}{T_O^{(2)}}$.

6 | AN ABSTRACT MODEL FOR DATAFLOW COMPARISON

This section compares *explicit* dataflow—and its PTG abstraction as it is used in PARSEC—with *implicit* dataflow models of runtimes that support the serial task-insertion interface. We derive an abstract model from our empirical studies (discussed in Section 5), to investigate the limits of the different dataflow paradigms. In Section 6.1, we start building our mathematical model for highly regular codes, and we continue to extend it in Section 6.2 for dynamic codes, which comprise the majority of real-world applications.

6.1 | Highly Regular Programs

Suppose there is a class of programs whose behavior can be expressed with functions that do not depend on program data. Affine codes that can be fully analyzed using polyhedral analysis are examples of such programs, and this class commonly occurs in fields such as dense linear algebra.

Explicit Dataflow:
In highly regular codes, the PTG is a compact representation of the DAG of execution, and everything can be described algebraically and can be generated without any overhead. Meaning, no additional work needs to be performed that grows with the number of nodes, P. Furthermore, the space requirement to store the PTG does not grow as the number of tasks grows but remains constant. Thus, the memory requirement is $O(1)$.

Given the PTG representation of a program, let T_p be the execution time it takes to compute all tasks assigned to a node, p. Then:

$$T_p = \sum_{t_i=0}^{N_p} T_{compute}^{t_i} + T_{comm}^{t_i} - T_{overlap}^{t_i}$$

where:

- N_p is the total number of tasks assigned to node p,
- $T_{compute}^{t_i}$ is the execution time it takes to compute task t_i,
- $T_{comm}^{t_i}$ is the communication time it takes to transfer data (from another node) that is required for task t_i (note, this time is negligible if the data is on the same node), and
- $T_{overlap}^{t_i}$ is the time of the data transfer that is overlapped with computation.

Suppose the workload is fairly equally balanced over the P nodes, then the execution time per node can be expressed as:

$$T_p = T_{compute}^{p} + T_{comm}^{p} - T_{overlap}^{p} \tag{1}$$

where:

- $T_{compute}^{p}$ is the compute load of node p (i.e., the average compute load per node),
- T_{comm}^{p} is the cumulative time it takes to transfer all data communicated by node p, and
- $T_{overlap}^{p}$ is the cumulative communication time that is overlapped with computation on node p.

In PTG-based executions, all communication is performed implicitly by the runtime using asynchronous communication primitives, thus it is fully overlapped with computation. As a result, Equation 1 can be rewritten as:

$$T_p^{expl} = max\left(T_{compute}^{p},\, T_{comm}^{p}\right). \tag{2}$$

This can be elaborated further into the following equation:

$$T_p^{expl} = max\left(\frac{T_w \cdot N}{P \cdot C},\, T_{comm}^{p}\right), \tag{3}$$

where T_w is the average task execution time, N is the total number of tasks in a DAG, P is the number of nodes, and C is the number of cores per node.

Implicit Dataflow:

For the *implicit* dataflow representation, where tasks are sequentially inserted in a queue by the runtime, the DAG connecting individual tasks for a given execution is not known *a priori*. For the runtime to make scheduling decisions for the tasks in the queue, and to detect data dependencies between tasks, it needs to discover the DAG of the execution, which needs to be stored in memory and traversed at execution time. This adds an overhead, $T_O^{(1)}$, of $O(N)$ for *implicit* dataflow models, since the size of the DAG grows as the number of tasks N grow. Again, there is no such overhead for the *explicit* dataflow paradigm due to its parameterized representation of the DAG. The PTG allows for the evaluation of any part of the DAG without having to store the DAG.

Once the DAG is discovered for *implicit* dataflow, the sole time requirements for the completion of all tasks in a DAG are identical to Equation 3. Note that, for both models, the communication is performed asynchronously by the runtime, so we can consider that it is performed at the maximum throughput of the network. In terms of total time per node for the *implicit* dataflow, this results in:

$$T_p^{impl} = max\left(\frac{T_w \cdot N}{P \cdot C}, T_{comm}^{p}\right) + T_O^{(1)}\,. \tag{4}$$

Implicit versus Explicit Dataflow:

Suppose problem sizes are sufficiently large and both dataflow models can completely hide the cost of non-blocking communication. For strong scaling, where the total problem size remains fixed and the number of compute nodes grows, *implicit* dataflow becomes even more expensive because the *relative* cost of the overhead, $T_O^{(1)}$, grows as the work per node shrinks with an increasing number of P.

Thus, on a machine with P nodes and C cores per node, the PTG version of a regular code will execute with a speedup of $O(P \cdot C)$, as below:

$$Speedup \;=\; \frac{T_p^{impl}}{T_p^{expl}} \;=\; \frac{\frac{T_w \cdot N}{P \cdot C} + T_O^{(1)}}{\frac{T_w \cdot N}{P \cdot C}} \;=\; 1 + \frac{T_O^{(1)}}{T_w \cdot N} \cdot P \cdot C\,. \tag{5}$$

This demonstrates the effect of the constant overhead caused by the sequential task insertion at the beginning of the execution in the *implicit* dataflow model. In real applications, where $T_w \cdot N$ is large, the constant $\frac{T_O^{(1)}}{T_w \cdot N}$ will be small, but large HPC systems have values of $P \cdot C$ into the tens of thousands, and this number is only expected to grow, thereby making the difference between *implicit* and *explicit* dataflow significant.

This model is congruent with the experimental observations of Agullo et al. (8), which compares implicit versus explicit dataflow implementations of the linear algebra operation *Cholesky*. The authors find that in order for the performance of the two approaches to be comparable, as the number of nodes grows, the programmer must "prune" the DAG. In other words, the programmer must be able to find a way to eliminate the overhead $T_O^{(1)}$ of building the whole DAG on each node. However, performing this prunning optimization is not feasible in the general case, and since our model is not limited to Cholesky, or any other specific linear algebra operation, we maintain that in the general case, when highly regular programs are taken into consideration, the performance ratio of an explicit versus implicit dataflow implementation will be described by Equation 5.

6.2 | Dynamic Programs

At the other end of the spectrum waits the reality that regular codes are merely a small subset of the real-world scientific applications. It would be desirable to be able to automatically generate PTGs by compilers for arbitrary programs, not just regular

ones. However, compiler analyses stumble on several obstacles, from technical implementation details to theoretical decidability limitations, and thus far have failed to deliver a general-purpose solution.

Explicit Dataflow:

Nonetheless, it is still possible to generate PTGs for dynamic applications, like NWCHEM, which are not algebraic, and have the metadata on which the execution depends. In such cases, the program data needs to be inspected and stored in metadata structures in a preliminary step before the PTG can be executed. Section 3.1 describes how this program data is generated and how it can then be accessed in the PTG during execution.

Unlike a fully algebraic PTG of a regular code that has no overhead, a more dynamic PTG will spend some time traversing metadata structures in memory, which adds an overhead, $T_O^{(2)}$, to the *explicit* dataflow model for dynamic programs. After paying this additional overhead of $T_O^{(2)}$, the runtime now has a PTG, so the communication of the explicit version of NWCHEM CC can now perform at the maximum throughput of the network.

The time it takes, T_p^{expl}, per node p to complete the computation of all tasks now becomes:

$$T_p^{expl} = max\left(\frac{T_w \cdot N}{P \cdot C}, T_{comm}^{p}\right) + T_O^{(2)} . \tag{6}$$

Implicit Dataflow:

Compared to the overhead of $T_O^{(2)}$ for the PTG version of the NWCHEM CC, the *implicit* dataflow version of the NWCHEM CC, as discussed in Section 4, has much higher overhead, which we will refer to in the text as $T_O^{(3)}$. `Insert_task()` calls are placed directly in the control flow of the original CC code and replace the calls to the `GEMM` functions. This implies that the overheads of the original code are inherited by the *implicit* dataflow version. $T_O^{(3)}$ can be classified as follows:

- Calls to functions (e.g., `GET_HASH_BLOCK`, `ADD_HASH_BLOCK`) that transfer data over the network using the GA layer (13): This communication is in the user code, and it is blocking, so there is no opportunity for communication and computation to overlap within the same thread of execution.
- The overhead of the original shared counter: Load balancing within each of the original NWCHEM CC subroutines is achieved through variables that are shared across the entire distributed-memory program and are atomically updated (read-modify-write) using GA operations. The use of such shared atomic variables—which is currently at the heart of the task-stealing and load-balancing solution—is bound to create significant overhead at large scales.

Because of the heavy inherited overhead of the original code, it can be asserted that $T_O^{(3)} > T_O^{(2)}$.

In the *implicit* dataflow implementation of CC, all original sequential chains of `GEMMs` have been parallelized by the insertion of parallel tasks. That implies that the fixed-size window of tasks, which is stored in memory at any given time, is $\frac{N}{N_{chain}}$, where N is the total number of tasks in a DAG, and N_{chain} is the total number of chains. Therefore, the runtime can never exploit more parallelism than the fixed-size window of tasks, which is the maximum number of tasks in a chain. This is supported in Section 5.3, where we show that one of the limiting factors of why the implicit versions stagnate after 8 cores per node is due to the average chain size being ≈ 6 (the weighted average, using the size of `GEMM` as weight, is $\approx$ 11).

Since the chain length, which defines the maximum parallelism, can be less than the number of cores on a node, the expression "$(P \cdot C)$" in our equations, which expresses the amount of available parallelism, must be replaced by:

$$min\left(\frac{N}{N_{chain}}, C\right) \cdot P .$$

And so, the time it takes, T_p^{impl}, per node p to complete the computation of all tasks in the *implicit* version of CC is then:

$$T_p^{impl} = max\left(\frac{T_w \cdot N}{min\left(\frac{N}{N_{chain}}, C\right) \cdot P}, T_{comm}^{p}\right) + T_O^{(3)} . \tag{7}$$

One can see that for communication-bound algorithms, the second term of Equation 7 will define the execution time, otherwise the time to completion will equal the total amount of work divided by the maximum available parallelism. For an NWCHEM

problem with very large chain sizes, the maximum available parallelism will be the number of cores C per node; otherwise, for problems with moderate chain sizes, the parallelism is limited by the chain size, which is $\frac{N}{N_{chain}}$.

In the beta-carotene case study, the chain sizes are always less than or equal to the number of cores per node. More specifically, chain sizes are either 8 or 16 for the most expensive subroutines, and smaller chain sizes (4, 6, 8) for less expensive subroutines.

Thus, for the experiments performed in this paper, the equation can be simplified to:

$$T_p^{impl} = max\left(\frac{T_w \cdot N_{chain}}{P}, T_{comm}^{p}\right) + T_O^{(3)} . \tag{8}$$

This means that the implicit dataflow CC allows for maximal 16 and on average 11 parallel workers per node. This is in agreement with the performance graphs in Figure 12 and Figure 13, which show that the scalability of all three *implicit* dataflow implementations (PaRSEC, StarPU, and OpenMP) stagnates after 8 cores per node.

Implicit versus Explicit Dataflow:

Combining Equation 6 and Equation 8, we see that the *explicit* dataflow version of NWChem CC will execute with a speedup of:

$$Speedup = \frac{T_p^{impl}}{T_p^{expl}} = \frac{\frac{T_w \cdot N_{chain}}{P} + T_O^{(3)}}{\frac{T_w \cdot N}{P \cdot C} + T_O^{(2)}} . \tag{9}$$

This demonstrates the following:

1. For a large number of nodes P, the speedup of the *explicit* CC code, in comparison to the *implicit* code, will converge to a constant $\propto \frac{T_O^{(3)}}{T_O^{(2)}}$. This can also be observed from the experiments in Section 5.3, where we compare the performance changes when moving from 32 to 64 to 128 nodes. With an increasing size of P, the performance of the *implicit* dataflow version converges closer to the performance of the *explicit* dataflow version:

$$Speedup_{P=32} = \frac{T_p^{impl}}{T_p^{expl}} = 2$$

versus

$$Speedup_{P=64} = \frac{T_p^{impl}}{T_p^{expl}} = 1.7$$

versus

$$Speedup_{P=128} = \frac{T_p^{impl}}{T_p^{expl}} = 1.2 .$$

2. For moderately large scales, where we can assert that the execution of the work is still much larger than the overheads, i.e., $\frac{T_w \cdot N}{P \cdot C} \gg T_O^{(2)}$ and $\frac{T_w \cdot N_{chain}}{P} \gg T_O^{(3)}$, Equation 9 can be simplified to:

$$\frac{T_p^{impl}}{T_p^{expl}} \approx \frac{N_{chain}}{N} \cdot C .$$

This simplified formula is in agreement with our experimental results. It can be observed in the performance graphs for 32 (Figure 12), 64 (Figure 13), and 128 nodes (Figure 14) that, as the number of cores per node increases for a **fixed** P, the speedup of the *explicit* dataflow version continues to grow with the increasing number of cores used per node.

This serves as a theoretical demonstration that—for the *implicit* dataflow version of NWChem CC, which allows for the utilization of the original control flow of CC and adds the sequential task insertions to it—some parallelism will remain hidden when only tasks are inserted for the `GEMMs` in a chain. Because CC already puts a significant burden on the memory requirements—with a storage cost of $O(M^4)$, where M is the size of the molecular system—there will be significant limitations for the *implicit* dataflow version when it stores the DAG of the execution in memory.

In contrast, the *explicit* dataflow version of NWChem CC discovers the same amount of parallelism as the *implicit* dataflow version that would store the entire DAG in memory but without the heavy memory requirements and without the high overhead of the sequential task insertion, $T_O^{(3)}$.

7 | RELATED WORK

In terms of dataflow environments, several groups have studied parallel execution models since the early 1990s that (1) allowed the same code to run on shared memory and distributed memory systems and (2) provided load-balancing features for irregular applications (16, 17). Unfortunately, most of these systems are impossible to use and evaluate today. Newer approaches, such as PM2 (18), SMARTS (19), Uintah (20), and Mentat (16), exist but do not satisfy the requirement for decentralized execution of medium-grain tasks ($\approx 10\mu s - 1ms$) in distributed-memory environments.

In terms of task scheduling systems, there are several approaches that employ DTD (i.e., build the entire [or part of the] DAG of execution in memory using skeleton programs). As a result, several solutions are currently in active development, including StarSs (21), OmpSs (22), Legion (23), and the Open Community Runtime (24). While their level of maturity, feature set, and similarity with PARSEC vary, none of them supports the PTG programming model. A notable exception is Intel's Concurrent Collections (CnC) (25) which provides an API for describing sets of tasks and their dependencies in a way that is closer to the way it is done in the PTG model, rather than DTD where the control flow of the original program is followed to submit the tasks to the runtime. Several other projects are embracing the DTD principle on shared memory (Cilk (26), Thread Building Blocks [TBB] (27)) or accelerator-based systems (28). Some of these systems support medium-sized distributed memory machines, but they introduce synchronization at both endpoints.

In an effort to offer alternative programming paradigms to the coarse grain parallel model (commonly associated with the Message Passing Interface [MPI]), a significant body of work has been dedicated to languages or language extensions—such as the partitioned global address space (PGAS) languages (29, 30, 31, 32, 33)—where the compiler is expected to perform the parallelization of the input program. Habanero (34) 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 (35) is another compiler tool that utilizes the Tarragon (36) prototype runtime system 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, which is based on light-weight processes and and actor model, a design which differentiates Charm++ from the other runtimes. Furthermore, there are large community efforts, such as TERAFLUX (37), which encompass the full software stack, including task execution runtime systems, in an effort to provide a large set of solutions.

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 here, we do not require the developers of NWCHEM to change the programming language they use, but rather adapted their FORTRAN 77 code to use task scheduling runtimes (specifically, PARSEC, STARPU, and OPENMP tasking).

8 | CONCLUSIONS

This work adopts dataflow programming models for a state-of-the-art electronic structure theory application and evaluates different dataflow executions in terms of scalability, resource utilization, and programmability. In terms of programmability, the implicit dataflow model expresses parallelism by submitting tasks in the sequential flow of the original program. This gives domain scientists the advantage to take an extensive, multi-million line application, like NWCHEM, and express additional parallelism by purely submitting tasks in the original control flow of the code. It also allows for the utilization of long-established features from the original implementation without diverging too much into an entirely different representation as one would for the explicit dataflow model.

The performance of implicit dataflow CC is on par for all three runtimes and exhibits a speedup over the original CC of a factor of 1.5 when running on 128 nodes. This effort substantiates that even *implicit dataflow-based execution at the node level* reveals notable performance benefits and enables more efficient and scalable computation of CC.

On the other hand, the explicit dataflow model demands a much bigger engineering effort compared to the implicit dataflow models. The PTG programming paradigm—which enables explicit dataflow representation of programs—proposes a completely different path from the way parallel applications have been designed and developed up to the present. The PTG decouples the expression of parallelism in the algorithm from the control flow ordering, data distribution, and load balance (4). Despite the lower startup overhead of implicit dataflow paradigms in terms of development effort (i.e., simply submitting tasks in the

sequential flow of the original code), the significance of the increased implementation effort from the PTG becomes apparent when comparing the superior performance of the explicit dataflow version of the CC to the implicit dataflow version and to the traditional CC computation. The PTG version of the CC outperforms the original CC version by a significant margin—to be precise, by a factor of 2.6 on 32 nodes.

Additionally, the explicit dataflow version manages to use an increasing number of cores to improve the performance—all the way up to 2,048 cores when running on 128 nodes (with 16 cores per node)—demonstrating not only a significant performance boost but also better scaling and greater utilization of compute resources thanks to the ability to fully overlap computation with communication. On the contrary, the original and the implicit dataflow CC code perform best on 8 cores per node and are not able to take full advantage of the 16 available cores per node. This is for the reason that both versions are tied to the limitations of the original program, i.e., blocking communication; shared variables that are atomically updated, which is at the heart of the original load balancing solution; and a significant amount of synchronizations that limit the overall scaling on much larger computational resources. In contrast, the PTG CC version with PaRSEC distributes the work in a round-robin fashion and avoids any kind of global agreement in the critical path of the DAG execution.

9 | ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their improvement suggestions. This material is based upon work supported by the US Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476 “A Co-Design Approach for Advances in Software and Hardware (CoDAASH)”. A portion of this research was performed using the Environmental Molecular Sciences Laboratory, a DOE Office of Science User Facility sponsored by the Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory.

References

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

[2] Kowalski Karol, Krishnamoorthy Sriram, Olson Ryan M., Tipparaju Vinod, Aprà E.. Scalable Implementations of Accurate Excited-state Coupled Cluster Theories: Application of High-level Methods to Porphyrin-based Systems. In: SC ’11:72:1–72:10ACM; 2011; New York, NY, USA.

[3] Cosnard M., Loi M.. Automatic Task Graph Generation Techniques. In: IEEE Computer Society; 1995; Washington, DC.

[4] Danalis Anthony, Bosilca George, Bouteiller Aurelien, Herault Thomas, Dongarra Jack. PTG: An Abstraction for Unhindered Parallelism. In: WOLFHPC ’14:21–30IEEE Press; 2014; Piscataway, NJ, USA.

[5] Jagode, Heike . Dataflow Programming Paradigms for Computational Chemistry Methods. Ph. D. Dissertation, Department of Computer Science and Engineering, University of Tennessee, Knoxville, http://trace.tennessee.edu/utk_graddiss/4469/, May 2017.

[6] Bosilca G., Bouteiller A., Danalis A., Herault T., Lemarinier P., Dongarra J.. DAGuE: A generic distributed DAG Engine for High Performance Computing.. *Parallel Computing.* 2012;38(1-2):27-51.

[7] Augonnet Cédric, Thibault Samuel, Namyst Raymond, Wacrenier Pierre-André. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience.* 2011;23(2):187–198.

[8] Agullo Emmanuel, Aumage Olivier, Faverge Mathieu, et al. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems.* 2017;PP(99).

[9] Jagode Heike, Danalis Anthony, Herault Thomas, et al. Utilizing dataflow-based execution for coupled cluster methods. In: :296–297; 2014.

[10] Danalis Anthony, Jagode Heike, Bosilca George, Dongarra Jack. PaRSEC in Practice: Optimizing a Legacy Chemistry Application through Distributed Task-Based Execution. In: :304-313IEEE; 2015.

[11] Jagode Heike, Danalis Anthony, Dongarra Jack. Accelerating NWChem Coupled Cluster through dataflow-based execution. *The International Journal of High Performance Computing Applications (IJHPCA).* 2017;:1–13.

[12] Bosilca George, Bouteiller Aurelien, Danalis Anthony, Herault Thomas, Dongarra Jack. From Serial Loops to Parallel Execution on Distributed Systems. In: Euro-Par’12:246–257Springer-Verlag; 2012; Berlin, Heidelberg.

[13] Nieplocha Jarek, Palmer Bruce, Tipparaju Vinod, Krishnan Manojkumar, Trease Harold, Apra Edoardo. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications.* 2006;20(2):203-231.

[14] Dagum L., Menon R.. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science Engineering.* 1998;5(1):46-55.

[15] Ayguade E., Copty N., Duran A., et al. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems.* 2009;20(3):404-418.

[16] Grimshaw A., Strayer W., Narayan P.. Dynamic object-oriented parallel processing. In: :33–47; 1993. http://www.cs.virginia.edu/~mentat/.

[17] Rinard M.. The Design, Implementation, and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language. PhD thesisStanford, CA1994.

[18] Namyst R., Méhaut J.-F.. PM^2: Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures. In: D'Hollander E.H., Joubert G. R., Peters F. J., Trystram D., eds. *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, Advances in Parallel Computing, vol. 11: :279–285Elsevier, North-Holland; 1996; Amsterdam.

[19] Vajracharya S., Karmesin S., Beckman P., et al. SMARTS: Exploting Temporal Locality and Parallelism through Vertical Execution. In: :302–310; 1999.

[20] St. Germain J. Davison, McCorquodale J., Parker S. G., Johnson C. R.. Uintah: A Massively Parallel Problem Solving Environment. In: ; 2000.

[21] Planas J., Badia R. M., Ayguadé E., Labarta J.. Hierarchical Task-Based Programming with StarSs. *Int. J. High Perf. Comput. Applic.*. 2009;23(3):284-299.

[22] Bueno J., Planas J., Duran A., et al. Productive Programming of GPU Clusters with OmpSs. In: ; 2012.

[23] Bauer Michael, Treichler Sean, Slaughter Elliott, Aiken Alex. Legion: Expressing Locality and Independence with Logical Regions. In: SC '12IEEE; 2012.

[24] Mattson T. G., Cledat R., Cavé V., et al. The Open Community Runtime: A runtime system for extreme scale computing. In: :1-7; 2016.

[25] Intel Concurrent Collections for C/C++ http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

[26] Blumofe R.D., Joerg C.F., Kuszmaul B.C., Leiserson C.E., Randall K.H., Zhou Y.. Cilk: An efficient multithreaded runtime system. In: ; 1995.

[27] Intel . Intel Threading Building Blocks http://threadingbuildingblocks.org/ .

[28] Ayguade Eduard, Badia Rosa M., Igual Francisco D., Labarta Jesus, Mayo Rafael, Quintana-Orti Enrique S.. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips Henk J., Epema Dick H. J., Lin Hai-Xiang, eds. *Euro-Par 2009 Parallel Processing (15th Euro-Par'09)*, Lecture Notes in Computer Science (LNCS), vol. 5704: Delft, The Netherlands: Springer-Verlag (New York) 2009 (pp. 851–862).

[29] Chamberlain Bradford L., Callahan David, Zima Hans P.. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications.* 2007;21:291–312.

[30] Charles Philippe, Grothoff Christian, Saraswat Vijay, et al. X10: An Object-oriented approach to non-uniform Clustered Computing. *SIGPLAN Not.*. 2005;40:519–538.

[31] El-Ghazawi T. A., Carlson W. W., Draper J. M.. UPC Specification v. 1.3 http://upc.gwu.edu/documentation.html2003.

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

[33] Yelick K., Semenzato L., Pike G., et al. Titanium: A high-performance Java dialect. In: ; 1998.

[34] Chatterjee S., Tasırlar S., Budimlic Z., et al. Integrating Asynchronous Task Parallelism with MPI. In: :712-725; 2013.

[35] Nguyen Tan, Cicotti Pietro, Bylaska Eric, Quinlan Dan, Baden Scott B.. Bamboo: translating MPI applications to a latency-tolerant, data-driven form. In: SC '12:39:1–39:11IEEE Computer Society Press; 2012; Los Alamitos, CA, USA.

[36] Cicotti Pietro. Tarragon: a Programming Model for Latency-Hiding Scientific Computations. Ph. D. Dissertation, Department of Computer Science and Engineering, University of California, San Diego, 2011.

[37] Giorgi Roberto, Badia Rosa M., Bodin François, et al. TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems.* 2014;38(8, Part B):976 - 990.