

Dataflow Programming Paradigms for Computational Chemistry Methods

A Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Heike Jagode

May 2017

© by Heike Jagode, 2017
All Rights Reserved.

Acknowledgements

I would like to thank my advisor, Dr. Jack Dongarra, for his constant support and for being willing to let me explore this unknown path. I wish to sincerely thank Dr. George Bosilca, Dr. Michael Berry, and Dr. David Keffer for serving on my doctoral committee and providing feedback on my dissertation and research.

Thank you to Dr. Anthony Danalis for believing in this work and for the countless valuable discussions that have challenged me and shaped my research. I would like to thank Dr. Theresa Windus, Dr. Karol Kowalski, and Dr. Edoardo Apra for the many conversations about computational chemistry methods, and for never shying away from the questions I sent their way. Thank you to Dr. Thomas Herault and Dr. Aurelien Bouteiller for their support and help with debugging precarious corner cases in PARSEC, triggered by this work.

I would like to thank my parents and family for their love and support and for cheering me on from across the ocean. I have to acknowledge my furry babes, Domino and Lily, who faithfully kept their eyes on the screen for every typed word of this dissertation. Nothing would be the same without them.

Finally, I want to thank the U.S. Air Force Office of Scientific Research for providing significant monetary funding toward this research under AFOSR Award No. FA9550-12-1-0476.

Abstract

The transition to multicore and heterogeneous architectures has shaped the High Performance Computing (HPC) landscape over the past decades. With the increase in scale, complexity, and heterogeneity of modern HPC platforms, one of the grim challenges for *traditional programming models* is to sustain the expected performance at scale. By contrast, *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. *This work introduces dataflow programming models for computational chemistry methods, and compares different dataflow executions in terms of programmability, resource utilization, and scalability.*

This effort is driven by computational chemistry applications, considering that they comprise one of the driving forces of HPC. In particular, many-body methods, such as Coupled Cluster methods (CC), which are the “gold standard” to compute energies in quantum chemistry, are of particular interest for the applied chemistry community. On that account, the latest development for CC methods is used as the primary vehicle for this research, but our effort is not limited to CC and can be applied across other application domains.

Two programming paradigms for expressing CC methods into a dataflow form, in order to make them capable of utilizing task scheduling systems, are presented. *Explicit dataflow*, is the programming model where the dataflow is explicitly specified by the developer, is contrasted with *implicit dataflow*, where a task scheduling runtime derives the dataflow. An abstract model is derived to explore the limits of the different dataflow programming paradigms.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals and Original Contributions	4
1.3	Outline of the Dissertation	5
2	Background and Significance	7
2.1	Dataflow versus Coarse-Grained Parallelism	8
2.2	The Dataflow Runtime PaRSEC	10
2.3	Computational Chemistry	11
2.3.1	Electronic Structure Theory	11
2.3.2	NWChem Coupled Cluster Methods	12
2.3.3	Coupled Cluster Methods over Dataflow	16
3	Dataflow Programming Models	20
3.1	Dataflow Extensions to NWChem	22
3.2	The Task Insertion Interface	23
3.3	Support for Heterogeneity	26
3.4	DAG Tuning	27
4	Explicit Dataflow Execution of Coupled Cluster Methods	28
4.1	Design Decisions	30
4.1.1	Parallelization and Optimization	36

4.1.2	Additional Levels of Parallelism	39
4.2	Algorithmic Variations	40
4.2.1	Exporting PARSEC data to Global Arrays	45
4.2.2	Using task priorities in PARSEC	48
4.2.3	Load Balancing	49
4.2.4	Performance Evaluation for Algorithmic Variations	50
4.3	Dataflow as a programming paradigm for CC	57
4.3.1	Communication computation overlapping	57
4.3.2	Freedom from control-flow	61
4.3.3	Multi-threading and accelerators	61
4.4	Performance Evaluation for entire CC	62
4.4.1	Methodology	63
4.4.2	Discussion	66
4.5	Summary	68
5	Implicit Dataflow Execution of Coupled Cluster Methods	70
5.1	Design Decisions	71
5.1.1	Insert_Task() Version	72
5.1.2	Design Variations for OpenMP Version	74
5.2	Performance Evaluation for entire CC	80
5.2.1	Methodology	80
5.2.2	Discussion	81
6	An Abstract Model for Dataflow Comparison	90
6.1	Affine Programs	90
6.2	Non-affine Programs	94
7	Related Work	100

8	Conclusions and Future Directions	104
8.1	Conclusions	105
8.2	Future Directions	108
	Bibliography	110
	Vita	121

List of Tables

4.1	Result of task_meta_data(loop-depth, L2-size, L1-size) for H_2O input.	33
4.2	GEMM size and shape, and total number of GEMMs for the top 10 heaviest CC subroutines when running beta-carotene with tileSize=45.	64
5.1	Thread binding options with OMP_PROC_BIND environment variable.	80
5.2	Speedup for dataflow versions over original CC.	87

List of Figures

2.1	Pseudocode of one CC subroutine as generated by the TCE.	14
2.2	DAG of the 41 T2 subroutines and its data dependencies.	18
3.1	Dataflow extensions for NWChem: Insert Task (left) and PTG (right).	23
3.2	Bridge between PARSEC and other Task Scheduling Runtimes	24
3.3	Task insertion interface example.	25
4.1	High level view of PARSEC code in NWChem.	29
4.2	(a) NWChem CC; (b) Populate Meta Data for PaRSEC version.	30
4.3	Minimal Vampir performance trace of CC t1.2.2.2 kernel for H_2O	34
4.4	PTG for GEMM tasks organized in a chain.	35
4.5	PTG snippet for parallel GEMM tasks.	36
4.6	Parallel GEMM operations followed by reduction.	38
4.7	Chain of GEMM operations computed sequentially.	38
4.8	Serialized sort and single write.	43
4.9	Parallelized sort and single write.	44
4.10	Parallelized sort and write	46
4.11	Exporting data to GA.	47
4.12	Comparison of algorithm variations and original code.	51
4.13	Trace of v4 (priority decreasing with chain number).	53
4.14	Trace of v2 (no task priorities).	53
4.15	Trace of original NWChem code.	55

4.16	Zoomed in on execution trace of original NWChem code.	56
4.17	Zoomed in on execution trace of dataflow-based NWChem code.	59
4.18	High level view of PARSEC code in NWChem.	63
4.19	Relative load of top 10 heaviest CC subroutines.	64
4.20	Distribution of Chain length for top five subroutines.	65
4.21	CC Execution time comparison using 32 nodes on EMSL/PNNL Cascade (for beta-carotene with tileSize=45).	67
4.22	CC Execution time comparison using 64 nodes on EMSL/PNNL Cascade (for beta-carotene with tileSize=45).	67
5.1	Original CC subroutine code (left), compared to the Insert_Task() version (right).	72
5.2	Insert_task_core() subroutine that is submitted to the runtime.	73
5.3	Pseudocode of OPENMP atomic (left); and critical (right) versions.	76
5.4	Pseudocode of buffered OPENMP Task implementation.	78
5.5	Comparison of CC execution time with OpenMP Tasks with “ATOMIC summation of C” using different OMP_PROC_BIND.	83
5.6	Comparison of CC execution time with OpenMP Tasks with “CRITI- CAL summation of C” using different OMP_PROC_BIND.	83
5.7	Comparison of CC execution time with OpenMP Tasks with “Buffered summation of C” using different OMP_PROC_BIND.	84
5.8	Comparison of CC execution time of all three OpenMP implementa- tions using OMP_PROC_BIND=SPREAD.	84
5.9	Comparison of CC run-time with StarPU using different schedulers.	86
5.10	CC Execution time comparison using 32 nodes (using up to 512 cores) on EMSL/PNNL Cascade (for beta-carotene with tileSize=45).	88
5.11	CC Execution time comparison using 64 nodes (using up to 1,024 cores) on EMSL/PNNL Cascade (for beta-carotene with tileSize=45).	88

5.12 CC Execution time comparison using 128 nodes (using up to 2,048 cores) on EMSL/PNNL Cascade (for beta-carotene with tilesize=45).	89
---	----

Chapter 1

Introduction

To this day, Coarse Grained Parallelism (CGP) remains the most popular programming model for large scale scientific applications, where the software is commonly structured as serial code, and parallelism is achieved by injecting carefully implemented send and receive calls to the communication layer, and executing multiple parallel instances of the code. The execution of such applications on today's largest High Performance Computing (HPC) systems shows that the CGP models already struggle to efficiently harness the parallelism available on systems with multiple cores, accelerators, and multiple NUMA hierarchies. As we move toward Exascale computing, with increasing parallelism, one of the challenges for the mainstream programming models is to sustain the expected performance scalability. Further challenges emerge from the increasing complexity and heterogeneity of hardware architectures, where coarse grained parallelism and fork-join expressions of parallelism limit performance portability of applications, and thereby sacrifice the productivity of scientific application developers.

1.1 Motivation

The main objective is to move away from the concepts of (a) *programming at the communication layer* and (b) *developing scientific applications for specific*

architectures or platforms; and instead, use dataflow programming paradigms to express and manage extreme-scale parallelism, while maintaining the performance portability of the code and increase scientific productivity.

The “scientific application vehicle” in this dissertation is quantum chemistry methods, but the insights gained from this research are much more broadly applicable across other application domains. The field of computational chemistry, aiming to simulate non-trivial physical systems, is chosen as the vehicle in this work because it imposes such high demands on the performance of the software and hardware, that it comprises one of the driving forces of High Performance Computing. In particular, computational methods—as they are present in quantum chemistry codes such as GAMESS [58, 29] and NWChem [62]—are extremely compute intensive and consume significant computer resources at national supercomputer centers [7].

These codes are used to address topics important to sustainability like renewable energy sources (e.g., solar and biorenewables), efficient batteries, and chemical catalysis. These applications require *highly accurate electronic methods* such as the Coupled Cluster methods (CC) [4], described in this work.

The price for this accuracy, however, is steep scaling of the cost with system size, which, even with access to powerful supercomputers, renders these methods inapplicable to many problems of interest [60]. Also the complexity of these software packages—with diverse code hierarchies, and millions of lines of code in a variety of programming languages—represents a central obstacle for long-term sustainability in the rapidly changing landscape of HPC. As a result, most of the computational methods are unable to take full advantage of today’s supercomputers, and will fall even further behind on tomorrow’s Exascale systems.

“NWChem is an extensive application (with approx. 4 million lines of code (3 million generated by the Tensor Contraction Engine (TCE)) that will shortly celebrate its 20th anniversary. Some of the challenges are (1) Reliance on hand rewriting of code for new hardware, (2) Lack

of adaptability and portability to the hardware environment. (3) Limited approaches to dynamically balance the workload.”

— Theresa L. Windus

Exascale Computing Project, 1st Annual Meeting, 2017

In spite of the importance of the CC methods, many of these algorithms have traditionally been thought of in a fairly sequential fashion. Particularly, it is not uncommon to fully form a matrix before allowing any of its elements to be used in a subsequent calculation. As a result, these applications have been predominantly parallelized and programmed using Coarse Grained Parallelism with explicit message passing. Meaning, in regard to the ratio of “computation” to the “amount of communication”, data is communicated infrequently after a large amount of computation takes place on the processors. The main motives of this trend have been the relative simplicity of the CGP programming effort, the high performance that such implementations can achieve, as well as the ubiquity and longevity of the tools that are necessary for developing and executing such applications.

In the race for post-petascale computing, several research groups—including our own—have been increasingly concerned with the feasibility of developing large scale applications that can utilize a satisfactory fraction of the computing power of future machines, and do so while preserving their portability and maintainability characteristics.

Dataflow-based execution on top of task scheduling runtimes has started emerging as an alternative programming paradigm to CGP models. Several success stories, which, as of today, are limited to the area of dense linear algebra [2, 10, 24, 26, 40], have encouraged the pursuit of this dissertation to *research the applicability and efficiency of dataflow programming models for production strength, large scale, scientific applications*, such as NWChem Coupled Cluster methods.

1.2 Goals and Original Contributions

This project “**Dataflow Programming Paradigms for Computational Chemistry Methods**” is a multi-disciplinary effort to convert computational chemistry algorithms into a dataflow-based form to make them compatible with next generation task scheduling systems, such as PaRSEC [11] and STARPU [1]. This work is not limited to the field of computational chemistry but the results from this research can be directly applied across other application domains.

This research investigates the following *key objectives* to attack the challenges associated with programmer productivity and portable performance of advanced computational chemistry applications on massively-parallel, hybrid, many-core systems:

1. Breaking down the computation into fine-grained tasks with explicitly defined data dependencies, so that the serialization imposed by the traditional, linear algorithms can be transformed into parallelism, allowing the overall computation to scale to much larger computational resources.
2. Using the flexibility afforded by the additional parallelism to enable greater utilization of compute resources by maximizing computation-communication overlap, and therefore using the hardware in more efficient ways.
3. Providing a robust and scalable directed acyclic graph (DAG) execution model for CC methods that is capable of utilizing different dataflow paradigms.
4. Comparing different dataflow paradigms for CC methods in terms of scalability, resource utilization, programmability. Specifically, this work contrasts “explicit dataflow”, where the developer has to explicitly specify the dataflow, with “implicit dataflow”, where a task scheduling runtime derives the dataflow.
5. Deriving an abstract model to explore the limits of the different dataflow paradigms.

In this work, all experiments and statistical studies use beta-carotene ($C_{40}H_{56}$) as the input molecule. Although, a large set of experiments has been performed with a variety of molecules, the beta-carotene molecule serves as a strongly representative test case of *large molecular complexes with a relatively small basis set*. This is a fairly common use case for researchers. Thus, in that context, the computational resource requirements for beta-carotene, both from the memory and computational power perspective, represent a classic use-case scenario with large implications.

1.3 Outline of the Dissertation

This work is organized as follows.

- [Chapter 2](#) provides an overview of the current state of the art techniques for programming models and theoretical quantum chemistry software. This chapter also presents the challenges this work seeks to overcome.
- [Chapter 3](#) introduces task-based dataflow models and the associated programming paradigms. It discusses how different dataflow models have been introduced to the computational chemistry package NWChem, and summarizes the impact on CC methods.
- [Chapter 4](#) presents the explicit dataflow version of NWChem CC methods. Various design decisions and their associated levels of parallelism, optimizations, and impacts on the performance are discussed.
- [Chapter 5](#) presents various implicit dataflow versions of NWChem CC and compares the performance for different runtimes. This chapter also analyzes implicit and explicit dataflow in terms of programmability and performance scalability.
- [Chapter 6](#) derives a theoretical model that investigates the limits of the different dataflow paradigms.

- [Chapter 7](#) provides a brief overview of the related work.
- Finally, [Chapter 8](#) summarizes the findings and research results, and concludes the dissertation with an outline of future research opportunities identified during this effort.

Chapter 2

Background and Significance

To provide context for the research effort presented in this work, the discussion starts with relevant prior work, emphasizing the enabling technical accomplishments and software; challenges this work seeks to overcome; and a brief description of the state of the art in the targeted scientific domains.

Portions of this chapter are drawn from the following publications:

- Jagode, H., Danalis, A., Bosilca G., Dongarra, J. **“Accelerating NWChem Coupled Cluster through dataflow-based Execution,”** *Parallel Processing and Applied Mathematics: 11th International Conference*, Krakow, Poland, September 6-9, 2015, Springer International Publishing, R. Wyrzykowski et al. (Eds.): PPAM 2015, Part I, LNCS 9573, pp. 366-376, 2016.
- Danalis, A., Jagode, H., Bosilca G., Dongarra, J. **“PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution,”** *2015 IEEE International Conference on Cluster Computing*, Chicago, Illinois, USA, IEEE, pp. 304-313, September 8-11, 2015.
- Jagode-McCraw, H., Danalis, A., Herault, T., Bosilca G., Dongarra, J., Kowalski, K., Windus, T.L. **“Utilizing Dataflow-based Execution for**

Coupled Cluster Methods,” *2014 IEEE International Conference on Cluster Computing*, Madrid, Spain, IEEE, pp. 296-297, September 22-26, 2014.

I was responsible for the design and implementation of the CC dataflow versions corresponding to each of these publications. In addition, I served as the primary author.

2.1 Dataflow versus Coarse-Grained Parallelism

Despite the radical changes of the hardware landscape over the past decades—featuring systems that combine multi-core CPUs and accelerators, shared and distributed memory, PCI-express and other interconnects—the most popular programming paradigm of choice for large scientific applications has remained coarse-grained parallelism (CGP). The most common communication layer used to write such programs is MPI [28], or some abstraction layer built on top of MPI such as Global Arrays (GA) [50]. Before diving into a more detailed discussion about “explicit” and “implicit” dataflow paradigms, the following section discusses the main benefits of dataflow programming models over CGP and how they relate to NWChem CC methods.

The control flow limits the flexibility of CGP applications. Parallelism in the current NWChem CC MPI code comes from the fact that different MPI ranks run on different CPUs in parallel. However, each rank is effectively a serial program since different CC subroutines execute in the order dictated by the control flow, even if they do not depend on one another. In this work, the latest state-of-the-art CC implementations [39] is used to develop different dataflow representations of the CC tensor contractions. The task-based runtime systems will deal with the specific hardware characteristics and orchestrates the execution based on resources and work availability.

Noise in the execution environment of large scale CGP applications—such as those written with MPI—has been shown to have significant effects on performance [48, 27, 34]. Dataflow-based task runtime systems, such as PARSEC, are less affected by such noise since it does not follow static coarse grain execution patterns, but rather decides dynamically the scheduling of tasks.

Idle time, due to the coarse grain nature of NWChem, is almost unavoidable. In the current programming model CC subroutines are organized in *levels*, and explicit barriers are used between levels to guarantee correctness. As a consequence, all compute units have to remain idle, waiting for the completion of the last subroutine at each level. However, the actual data dependencies between the subroutines do not require such a fork-join behavior. Expressing the application in a dataflow model will expose the minimum necessary dependencies and enable a finer grain composition of subroutines, reducing idle time.

Communication-computation overlap is a desirable feature that most modern networks and communication libraries support. However, for an application to take advantage of overlap it must issue communication initiation requests before it needs to use the data and allow other (independent) work to take place while the data is being transferred. In the coarse grain programming model this is not a trivial task, as the developer needs to explicitly manage the communication. A testament to this complexity is the fact that the CC code (as currently generated by NWChem) issues the data transfers right before the data is needed, thus missing the opportunity to achieve overlap.

In dataflow runtime systems like PARSEC, on the other hand, communication is implicit. The runtime derives the need for a data transfer based on the dependencies between tasks and the runtime’s knowledge of the affinity of tasks to nodes. When all predecessors of a task have finished, the runtime initiates the (asynchronous) transfer of the data while executing other tasks. Consequently, data transfers are overlapped with computation.

Heterogeneity is a challenge that parallel application developers have to address in order to modernize their applications given the pervasiveness of accelerators and co-processors in HPC. In the dataflow models that have been adopted for NWChem in this research effort, work is naturally decomposed in tasks and therefore it can be divided between heterogeneous resources more readily than in the coarse grain parallelism model.

2.2 The Dataflow Runtime PaRSEC

The Parallel Runtime Scheduling and Execution Control (PaRSEC) framework [11] is a task-based dataflow-driven runtime designed to achieve 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.

In this work, PaRSEC has been chosen as one of the runtime systems because of the execution model it offers—which is based on a symbolic Parameterized Task Graph (PTG) [16, 18, 19]. The PTG abstraction is the main difference between PaRSEC and other task engines (e.g. STARPU [1]), differentiating the way tasks, and their data dependencies, are represented. The PTG enables PaRSEC to employ a unique, *symbolic way of discovering and processing the graph of tasks*, which can deliver unparalleled performance in the era of many-core, highly heterogeneous, extreme scale platforms. Previous research [26, 41, 24, 2, 10] in the area of dense linear algebra has convincingly shown that this is the case, and based on the fact that tensor contractions consist mainly of linear algebra operations, we assert that those previous results are of direct relevance to computational chemistry and to tensor algebra in particular.

PaRSEC’s PTG programming paradigm proposes a complete departure from the way we have been designing and developing applications. However, as has be

demonstrates in [Chapter 4](#), the conversion from CGP to dataflow-based execution can happen gradually. Performance critical parts of an application can be selectively ported to execute over PARSEC and then be re-integrated seamlessly into the larger application which is oblivious to this transformation.

2.3 Computational Chemistry

2.3.1 Electronic Structure Theory

Electronic structure theory, commonly referred to as theoretical quantum chemistry, is at the core of the more general field of computational chemistry. It investigates e.g. the *ground state* of individual atoms and molecules, the *excited states*, and the *transition states* that occur during chemical reactions. Practically, all experimental properties can be computed from first principles quantum chemistry methods.

The formal scaling of different first principles quantum chemistry methods—where N is a measure of the molecular system size—ranges from N^4 (Hartree-Fock, Density-Functional Theory) to N^5 (second order perturbation theory MP2) to N^7 (Coupled Cluster) to exponential Multi-Reference methods. In order to apply *accurate quantum chemistry methods to real problems*, it is a topmost interest of the computational chemistry community to improve the scaling of these methods, by the development of novel theoretical approaches, the development of novel algorithms, and by taking optimal advantage of new computer architectures. Therefore, this work is the tip of an iceberg of bring electronic structure codes into the realm of exascale computations.

While the focus in this effort is on Coupled Cluster methods—as they are referenced as the “*gold standard*” to compute *highly accurate energies in quantum chemistry* [60, 56, 46]—the insights gained from this research will be much more broadly applicable.

2.3.2 NWChem Coupled Cluster Methods

NWCHEM [62] 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 (CC) theory [4, 5, 15, 63] is considered by many to be the best and most reliable method, particularly, for accurate quantum-mechanical description of ground and excited states of chemical systems. Its accuracy, however, comes at a significant computational cost. Given the steep computational cost of the CC methods, the scalability of NWCHEM in this context is extremely important for real science. Many chemical problems related to combustion, energy conversion and storage, catalysis, and molecular spectroscopy are untenable without CC methods on supercomputers [37]. *One of the goals for this project is to strengthen the NWCHEM Coupled Cluster methods by enabling more powerful computations through dataflow programming and execution, much better resource management, and a robust path to exploit hybrid computer architectures.*

Especially important in the hierarchy of the CC formalism is the iterative CC model with Single and Double excitations (CCSD) [54], which is the base for many accurate perturbative CC formalisms, including the ubiquitous CCSD(T) (Triple) approach [55]. Given the size N of the molecular system, CCSD has a computational cost of $O(N^6)$ and storage cost of $O(N^4)$. The starting point for this research is the CCSD version that takes advantage of the alternative task scheduling, and the details of these implementations have been described in [39].

TCE

An important role in designing the optimum “memory versus computational cost” strategies in CC implementations is played by the program synthesis system, Tensor Contraction Engine (TCE) [33]. The TCE abstracts and automates the

time-consuming and error-prone processes of deriving the working equations of second-quantized many-electron theories and synthesizing parallel computer programs on the basis of these equations. Current development is mostly focused on CC implementations 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 are parallelized with the use of *Global Arrays* (GA), which provides a shared memory style programming environment in the context of distributed array data structures [50]. Global Arrays is an extension to the message-passing interface and requires MPI to work. In general, NWChem contains approx. 4 million lines of code, of which 3 million are generated by the TCE.

Structure of the CC Approach

The iterative CC code is generated through the TCE* into many 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 tensor contractions which are implemented as series of matrix-matrix multiplications (GEMMs). There are typically many thousands of such terms in any one problem, but their regularity makes it relatively straightforward to translate them into FORTRAN code—parallelized with the use of *Global Arrays*—through the TCE. Figure 2.1 shows the pseudo FORTRAN code for one of the generated T1 and T2 subroutines, highlighting that most work is in deep loop nests. These loop nests consist of three types of code:

- Local memory management (i.e., `MA_PUSH_GET()`, `MA_POP_STACK()`),

*TCE-generated code in NWChem 6.6

```

1 my_next_task = SharedCounter()
2 DO h7b = 1,noab
3     DO p3b = noab+1,noab+nvab
4         IF (int_mb(k_spin+h7b).eq.int_mb(...)) THEN
5             ...
6             call MA_PUSH_GET(f(p3b,h7b),...,k_c)
7             call DFILL(..., dbl_mb(k_c),...)
8             ...
9             DO p5b = noab+1,noab+nvab
10                DO h6b = 1,noab
11                    call GET_HASH_BLOCK(dbl_mb(k_b)...f(p3b,p5b,h7b,h6b)
12                    call TCE_SORT_4(dbl_mb(k_b),...,f(p3b,p5b,h7b,h6b))
13                    ...
14                    call DGEMM(...,f(p3b,p5b,h7b,h6b))
15                END DO
16            END DO
17            ...
18            call ADD_HASH_BLOCK(dbl_mb(k_c),...)
19            call MA_POP_STACK(k_c)
20
21            my_next_task = SharedCounter()
22        END IF
23    END DO
24 END DO

```

Figure 2.1: Pseudocode of one CC subroutine as generated by the TCE.

- Calls to functions (i.e., `GET_HASH_BLOCK()`, `ADD_HASH_BLOCK()`) that transfer data over the network via the GA layer,
- Calls to the subroutines that perform the actual computation on the data `DGEMM()` and `TCE_SORT_*` (which performs an $O(n)$ remapping of the data, rather than an $O(n * \log(n))$ sorting).

The control flow of the loops is parameterized, but static. That is, the induction variable of a loop with a header such as “`DO p3b = noab+1,noab+nvab`” (i.e., `p3b`) may take different values between different executions of the code, but during a single execution of CC the values of the parameters `noab` and `nvab` will not vary; therefore every time this loop executes it will perform the same number of steps,

and the induction variable `p3b` will take the same set of values. This enables us to restructure the body of the inner loop into tasks that can be executed by PARSEC. Specifically, tasks with an execution space that is parameterized (by `noab`, `nvab`, etc.), but constant during execution.

Parallelization of CC

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

Load balancing within each of the seven levels of subroutines is achieved through shared variables (exemplified in [Figure 2.1](#) through `SharedCounter()`) that are atomically updated (read-modify-write) using GA operations. This is an excellent case 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.

Also, the notion of *task* in the current CC implementation of NWChem and the notion of *task* in PARSEC are not identical. As discussed before, in NWChem, a

task is a whole chain of **GEMMs**, executed serially, one after the other. In our PARSEC implementation of CC, each individual **GEMM** kernel is a task on its own, and the choice between executing them as a chain, or as a reduction tree, is almost as simple as flipping a switch.

There is clearly a need to improve the NWChem CC workflow to withstand the expected vast increase in intra-node parallelism and deeper memory hierarchies at Exascale. Unfortunately, there are difficult challenges to both performance and productivity in the development of computational modeling programs. However, the manual development of accurate quantum chemistry models can typically take an expert months to years of tedious effort to develop and debug a high-performance implementation [32].

The main tasks, with the most significant impact, are (1) the elimination of synchronization points by describing the data dependencies between the matrix blocks—creating a true dataflow programming model and execution, and (2) dividing the current tasks into finer grained tasks to allow for the exploitation of more parallelism. Once a dataflow execution of CC exists, the choice between executing individual tasks as a chain, or as a reduction tree, is almost as simple as flipping a switch.

This effort to research dataflow programming and execution of CC, in order to take advantage of DAG execution models capable of utilizing task scheduling systems, is important as it can successfully address heterogeneous computing resources with different capabilities, such as accelerators and co-processors, and hence alleviates the burden on application developers.

2.3.3 Coupled Cluster Methods over Dataflow

PARSEC provides a front-end compiler for converting canonical serial codes into the PTG representation. However, due to computability limits, this tool is limited to polyhedral codes, i.e., if a code is not affine then a static analysis is not possible

at compile time. Affine codes can only contain loop headers (bounds and steps) and array indices with expressions that are limited to addition and subtraction of the induction variables, constants, and numeric literals (as well as multiplication by numeric literals) and branches (if-then-else) that contain only similar arithmetic expressions and comparison operators. These affine codes (e.g., dense linear algebra is largely affine codes) can be fully analyzed using polyhedral analysis and a compact representation of the DAG—the PTG—can be generated. A critical characteristic about the PTG is it can be used by the runtime to evaluate any part of the DAG without having to store the entire DAG in memory. This is one of the important features that differentiate PARSEC from any other task-scheduling system.

The problem with affine codes, though, is they are a very small subset of the real world applications. The CC code generated by TCE is neither organized in pure tasks (i.e., functions with no side-effects to any memory other than arguments passed to the function itself) nor is the control flow affine (e.g. loop execution space has holes in it; branches are statically undecidable since their outcome depends on program data, and thus it cannot be resolved at compile time) [37].

While the CC code seems polyhedral, it is not quite so. The code generated by TCE includes branches that perform array lookups into program data. For example, branches such as “`IF(int_mb(k_spin+h7b-1)...)`” (see [Figure 2.1](#)) are very common. Such branches make the code not only non-affine, but statically undecidable since their outcome depends on program data, and thus it cannot be resolved at compile time.

However, while the behavior of the CC code depends on program data, this data is constant during a given execution of the code. Therefore, the code can be expressed as a parameterized DAG, by using lookups into the program data, either directly or indirectly. In the explicit dataflow implementation, described in [Chapter 4](#), the program data is accessed indirectly by building meta-data structures in a preliminary step. The details of this “first step” are explained in [Section 4.1](#).

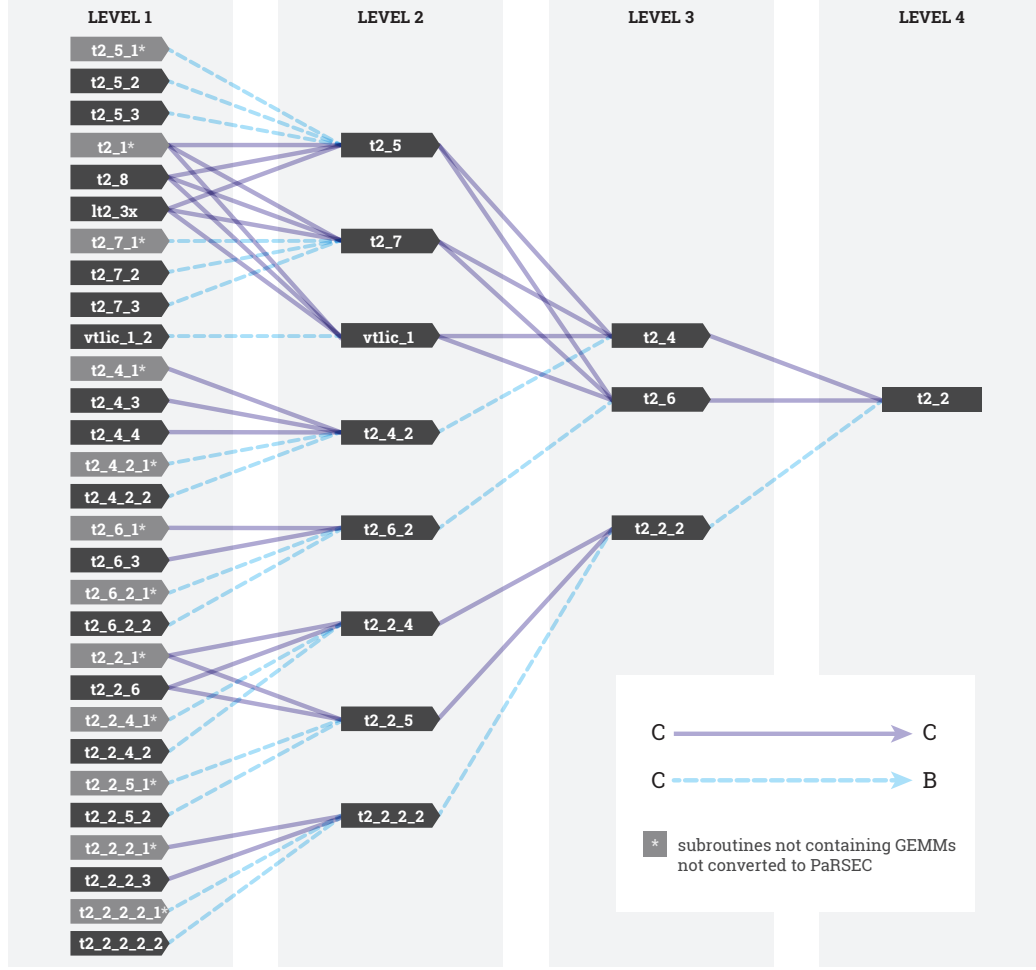


Figure 2.2: DAG of the 41 T2 subroutines and its data dependencies.

In this work, different dataflow versions have been implemented for all functions of the CC computation that are associated with calculating parts of the T2 amplitudes, particularly the ones that perform a GEMM operation (the most time consuming parts). More precisely, a total of 29[†] of the 41 T2 subroutines have been converted, which will be referred to under the unified moniker of “Orig:T2” for the original version, and “Dataflow:T2” for the dataflow versions of the subroutines.

[†]All subroutines with prefix “icsd_t2_” and suffices: 2_2_2_2(), 2_2_3(), 2_4_2(), 2_5_2(), 2_6(), lt2_3x(), 4_2_2(), 4_3(), 4_4(), 5_2(), 5_3(), 6_2_2(), 6_3(), 7_2(), 7_3(), vtlic_1_2(), 8(), 2_2_2(), 2_4(), 2_5(), 4_2(), 5(), 6_2(), vtlic_1, 7(), 2_2(), 4(), 6(), 2()

Figure 2.2 illustrates the dataflow of the original 41 T2 subroutines. It shows how the work is divided into four distinct levels. The solid edges of this DAG represent the dataflow where output data (matrix **C**) is added to output data in another level (**C**->**C**); and the dotted edges show where output data (matrix **C**) is used as input data (matrix **B**) for different subroutines (**C**->**B**).

From this DAG it becomes very apparent that not every subroutine in the upper levels has to wait for the completion of all subroutines in the lower levels. For example, `t2_2_2` in Level 3 only depends on the data coming out of `t2_2_4`, `t2_2_5`, and `t2_2_2_2` in Level 2; while these three only depend on 10 (of the 29) subroutines in Level 1.

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

The chosen 29 subroutines comprise approx. 91% of the execution time of all 41 T2 subroutines when computing the CC correlation energy of the *beta-carotene* molecule ($C_{40}H_{56}$). (not including the 19 T1 subroutines and additional execution steps that set up the iterative CC computation). More details are discussed in Section 4.4.1 and illustrated in Figure 4.19.

Chapter 3

Dataflow Programming Models

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 Coarse Grain Parallelism with explicit message-passing by using 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 [53, 1, 64, 14] 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 [10] library, which is built on top of PARSEC, offers a concrete experimental demonstration of the performance scalability that the PTG approach offers.

In this work, different dataflow-based representations of the CC methods have been developed:

1. An “explicit” dataflow-based representation via the Parameterized Task Graph (PTG).
2. A Task-based representation where the dataflow will be “implicitly” derived by various runtimes.

One of the goals is to allow for the execution of the “implicit” dataflow representations over *any runtime with “Insert Task” interface support* (starting with—but without being limited to—PARSEC, STARPU), instead of relying solely on the outdated FORTRAN 77 with Global Arrays form that the TCE currently generates.

A completely modular design is presented with the main goal to add DAG execution models as a new module that will be tightly coupled with the basic components of the quantum chemistry software package. This approach enables NWChem to continue having a transformative impact in the field of computational chemistry in the post-petascale era.

This work is a multi-disciplinary approach with research and development challenges for both the computational science and computer science, driven by shared objectives in chemistry/physics and a desire to advance computational science as a whole. The key challenges for the domain sciences consist of abstracting the relevant algorithms to a higher level, and providing as much of the implementation as possible through the support of the dataflow runtime layer. The goal is to replace existing code that interlaces algorithm, data model, and performance optimization in legacy implementations that, in most cases, were begun as serial algorithmic expressions. For the Coupled Cluster theory, the computation to be done consists of the calculation of arrays of integrals, and the contraction of these with amplitude tensors to form

intermediate quantities, which themselves are inputs for further contractions. The new implementations separate each of these tasks, and present them to the dataflow runtime layer that allows a compact specification of the algorithm that is close to the underlying theory.

On the computer science side, the dataflow runtimes 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 computational kernel, and the dataflow between those tasks. This information is provided through the algebraic specification of the DAG, which is only instantiated as needed. The runtime combines the information contained in the DAG 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. A major research goal of this work is to incorporate the dataflow runtimes into NWChem through the process of the automatic generation of the symbolic representation of the tasks and their data dependencies, without diminishing the efficiency and expressivity of the dataflow runtimes, e.g., through the hierarchical specification of the DAG, abstract specification of place, incorporation of prioritized tasks, and enhanced serialization and migration of tasks.

3.1 Dataflow Extensions to NWChem

One of the key goals of this work is to extend NWChem with a modular and production-quality software framework. [Figure 3.1](#) depicts a schematic representation of the framework, where it shows the input is a list of intermediates and tensor contractions that the TCE provides (and currently uses to synthesize FORTRAN code) to evaluate the sequence of tensor contractions. Both of the new extensions—depicted in [Figure 3.1](#) as “Insert Task” (left) and the PTG generator (right)—are tightly coupled with the basic NWChem CC components. The role of the extensions is to identify the tasks and level of tasks to be performed, and generate an *explicit*

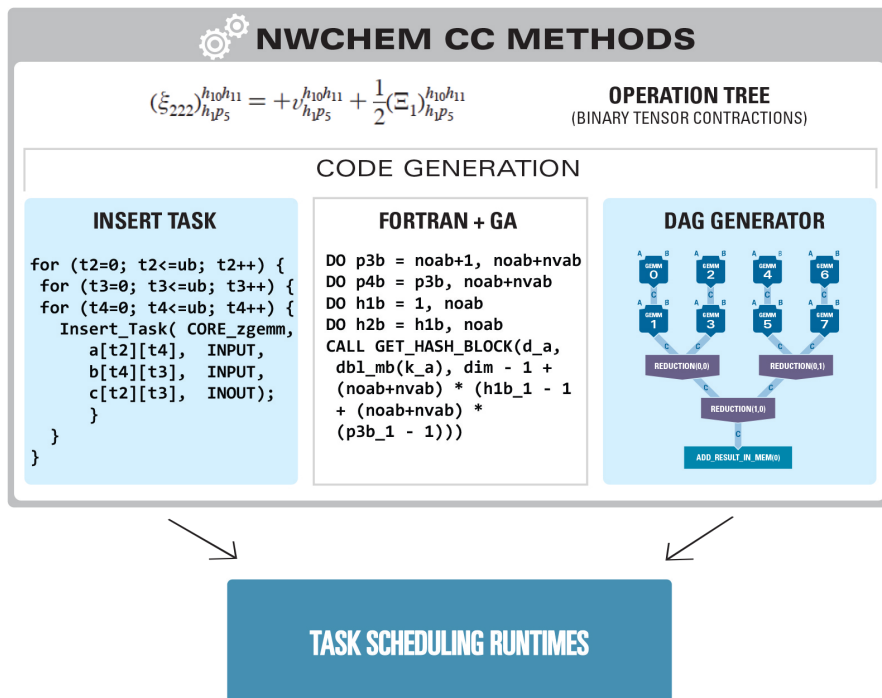


Figure 3.1: Dataflow extensions for NWChem: Insert Task (left) and PTG (right).

and *implicit* dataflow representations. The extension shown on the right generates the PTG that defines the task-based execution for PARSEC, and interacts with the meta-data structures. The PTG extension is described in [Chapter 4](#). The extension shown on the left generates serial C code that any runtime that supports the “Insert Task” interface may use to submit tasks and generate the DAG of execution. The Insert-Task extension is described in [Chapter 5](#). An abstract model that compares both execution and quantifies the differences between “explicit” and “implicit” dataflow models is presented in [Chapter 6](#).

3.2 The Task Insertion Interface

[Figure 3.2](#) shows how the two new dataflow modules (task-insertion approach and PTG approach) for CC can be plugged into PARSEC or any other runtime. The development of distributed CC algorithms *based on the serial task-insertion API*

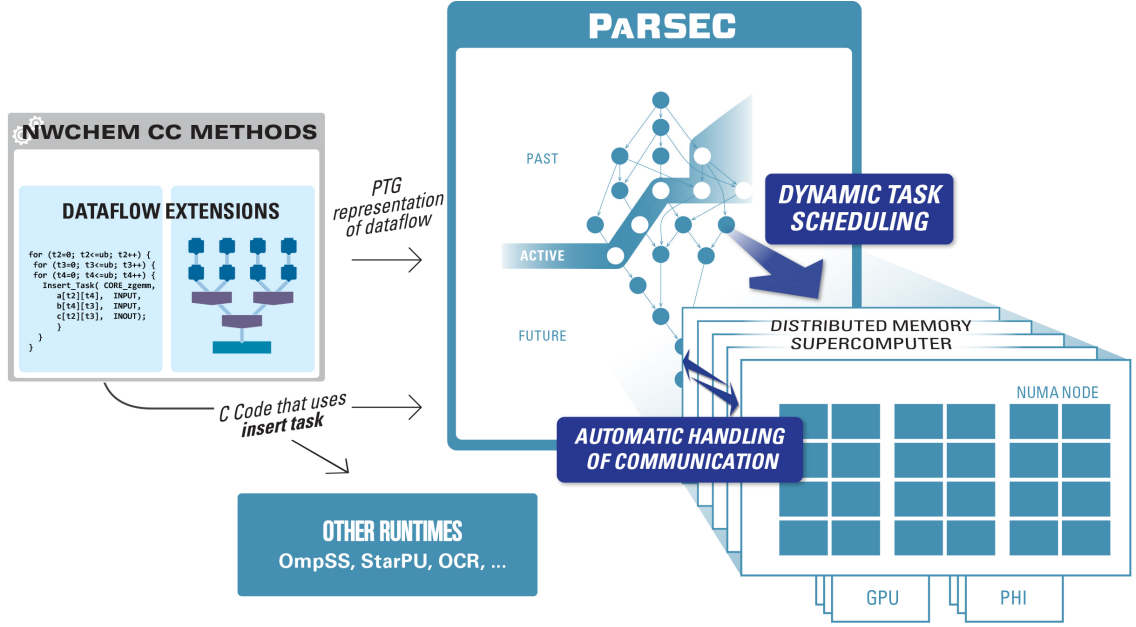


Figure 3.2: Bridge between PARSEC and other Task Scheduling Runtimes

(Figure 3.1 - left) is particularly crucial as it allows for distributed task scheduling and data dependency management by *any* runtime that supports a task-insertion API (e.g., STARPU, OmpSS, Open Community Runtime (OCR)). It is crucial to such a degree that: (a) it offers flexibility and choice of runtime to a broad group of scientific application users that depend on CC methods; (b) it makes the dataflow version of CC broadly accessible to both the development and applications communities; (c) it provides interoperability and sustainability for the foreseeable future.

Methodology

A simplified task-insertion example is presented in Figure 3.3. The call to the function `Insert_Task()` adds a new task `CORE_zgemm` to the scheduler of the runtime, as well as providing the data pointers, sizes, and dependency information. Note, `a`, `b`, `c` and `ub` are input parameters to the kernel.

```

1  int zgemm(TYPE a, TYPE b, TYPE c, int N_sz, int ub )
2  {
3      ...
4      for (t2=0; t2<=ub; t2++) {
5          for (t3=0; t3<=ub; t3++) {
6              for (t4=0; t4<=ub; t4++) {
7                  Insert_Task( CORE_zgemm,
8                              a[t2][t4], INPUT,
9                              b[t4][t3], INPUT,
10                             c[t2][t3], INOUT,
11                             t2,      VALUE,
12                             t3,      VALUE,
13                             t4,      VALUE,
14                             N_sz,    VALUE);
15              }
16          }
17      }
18      ...
19  }

```

Figure 3.3: Task insertion interface example.

Runtimes with Insert Task Support The user submits tasks into the system using a task-insertion API (similar to what is depicted in [Figure 3.3](#)). After each task is inserted into the runtime, the arguments for the task are used to make a dependency structure, where reads and writes on data are queued. These queues of data requests are checked for data dependencies and the ordering of these dependencies forms an explicit DAG for the tasks. The runtime system schedules the tasks that are not waiting for dependencies, and execution threads pick up and execute the tasks. After a task completes execution, for each data parameter, the status of the DAG is updated to reflect the completed task. Then the runtime schedules any released tasks that do not have to wait for any more dependencies.

Data Dependencies In order for the runtime to be able to determine dependencies between the tasks, it needs to know how each task is using its arguments. Constant arguments are marked as **VALUE**, which means that they are not dependencies, but

values like constants or loop indices; these values are just stored with the task. Actual dependencies can be `INPUT`, `OUTPUT`, or `INOUT` (standing for input/output). They allow one to specify, for each task in the program, what data a task is waiting for and indicates its readiness. Each time a new task is created, its `INPUT` and `OUTPUT` dependencies are matched against the dependencies of existing tasks.

The implicit dataflow version of CC offers any runtime—that accepts tasks via a task insertion process—the ability to determine the task and data dependencies dynamically based on the data usage patterns.

3.3 Support for Heterogeneity

The different dataflow versions of NWChem’s CC kernels—which are generated as a result of this work—enable much finer grained parallelism than the original CC implementation. Also, since the unnecessary synchronization points that currently exist in CC—due to the coarse grain design—can be eliminated, the resulting tasks can be composed in more elaborate ways than the current fork join pattern. In particular, groups of interdependent tasks can be treated as individual entities, or aggregated into a single super-task when coarser grain execution is required. The explicit and implicit dataflow strategies have many advantages since these finer grained parallel tasks, or coarser super-tasks, could run on different parts of heterogeneous platforms, utilizing the components (e.g., CPU or coprocessor) that perform best for the type of task under consideration.

An advantage of this effort, in comparison to independent efforts to bring heterogeneous computing capabilities to software packages such as NWChem, comes from the ability of task-based runtimes to hide the complexity of heterogeneous resources from the application layer. For instance, PARSEC allows the same algorithm (as defined by the data dependencies between tasks) to seamlessly execute on CPUs, accelerators, and co-processors, as long as the application provides appropriate kernels

for the corresponding devices. A few of these kernels are available in NWChem, but additional ones can be identified for development and used in the runtime environment. The scheduling and management of data transfers to and from the devices can then be handled internally by the runtime.

3.4 DAG Tuning

When developing the DAG for the CC subroutines, multiple design choices were investigated in several stages of the development. In some cases the best choice seemed intuitively obvious, but in others it was not clear which choice is the best and whether the same choice remains optimal when the problem size or execution environment changes. However, implementing multiple versions of the code by hand for several subroutines in order to compare them and choose the optimal solution was not practical, so for several choices educated guesses were made.

Part of this work is the careful design of the dataflow generations in a way that multiple options can be selected by the developer leading to the creation of different DAGs. This way, the generated code is flexible and tunable for different inputs and different execution environments. As an example, a set of tensor contractions can be executed as a serial chain, or fully in parallel followed by a reduction tree, or as a small set of smaller parallel chains followed by a reduction. Choosing between these options is not trivial since a serial chain optimizes the memory locality, and the fully parallel execution maximizes available parallelism and load balancing, whereas the set of smaller chains aims to balance both. Furthermore, it is not expected that any such choice will be optimal across all problem sizes and execution environments. Therefore, the DAG generation stage has been designed so that different choices can be made in a per case basis.

Chapter 4

Explicit Dataflow Execution of Coupled Cluster Methods

This chapter 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.

Portions of this chapter are drawn from the following publications:

- Jagode, H., Danalis, A., Dongarra, J. “**Accelerating NWChem Coupled Cluster through dataflow-based Execution,**” *International Journal of High Performance Computing Applications (IJHPCA)*, SAGE Publications, pp. 1-13, <http://dx.doi.org/10.1177/1094342016672543>, January 2017.
- Danalis, A., Jagode, H., Bosilca G., Dongarra, J. “**PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution,**” *2015 IEEE International Conference on Cluster Computing*, Chicago, Illinois, USA, IEEE, pp. 304-313, September 8-11, 2015.

I was responsible for the design and implementation of the CC dataflow versions corresponding to each of these publications. In addition, I served as the primary author.

<pre> 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 END IF 23 END DO 24 END DO 25</pre>	<pre> L1_idx = 1 DO i DO j ! Validation of loop index values IF (int_mb(k_spin+i) .eq. int_mb(k_spin+j)) IF (int_mb(k_sym+j) ...) DFILL(C) DO k DO m ! Validation of loop index values IF (int_mb(k_spin+k) .eq. ...) task_meta_data(1,L2_idx,L1_idx)=i task_meta_data(2,L2_idx,L1_idx)=j task_meta_data(3,L2_idx,L1_idx)=k task_meta_data(4,L2_idx,L1_idx)=m L2_idx = L2_idx + 1 END IF END DO END DO L2_cnt = L2_idx - 1 inner_task_count(L1_idx) = L2_cnt L1_idx = L1_idx + 1 END IF END DO END DO END DO </pre>
---	---

Figure 4.2: (a) NWChem CC; (b) Populate Meta Data for PaRSEC version.

4.1 Design Decisions

The original code of our chosen subroutines consists of deep loop nests that contain the memory access routines as well as the main computation, namely `SORT` and `GEMM`. In addition to the loops, the code contains several `IF` statements, such as the one mentioned above. When CC executes, the 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`. As shown in Figure 4.1, to create the PaRSEC-enabled version, we decomposed the code into two steps:

Step 1: The Discovery Phase

The first step is the discovery phase that bridges the original legacy code with PARSEC. It 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)`) that is involved in the `IF` branches (lines 4-6 and 10-11 in [Figure 4.2](#)), and stores the results in custom meta-data vectors that we defined. In addition, this step probes the GA library to discover the physical location of the program data on which the `GEMMs` will operate on, and stores these addresses into the meta-data structures as well. Since the data of NWChem that affects the control flow is immutable at run 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 4.2](#) (a) shows the TCE-generated pseudo-code for one CC sub-kernel, while the pseudo-code shown in (b) is manually derived from the original TCE code, in order to populate all the meta-data that PARSEC will use to dynamically derive the data dependencies between the real tasks. Specifically, in the version depicted in [Figure 4.2](#) (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()`, etc. Instead, in the place of the original subroutine calls, we insert operations that store the status of the execution into custom meta-data arrays that we have introduced.

The custom meta-data vectors merely hold information regarding the actual loop iterations that will execute the computational kernels at run time, i.e., iterations where all the `IF` statements evaluate to `true`. This step significantly reduces the execution space of the loop nests by eliminating all entries that would not have executed. After the first step is finished, the meta-data 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 to these data.

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

1. Determine what the tasks are: For the CC code, the tasks are

DFILL()	Initialization of an array
SORT_2() , SORT_4()	Sorting of an array
DGEMM()	double precision matrix-matrix multiplication

2. Dimensions of meta data: The number of dimensions of meta data 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 meta data:

C: **task_meta_data**[**L1_size**][**L2_size**][**loop_depth**]

FORTTRAN: **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. I.e., not all possible tasks will actually execute, as a matter of fact, only a small subset of tasks are truly executed.

Since not the entire execution space results in “real tasks”, the population of meta data gives us the real tasks only, excluding all the remaining tasks. For the *H₂O* input data set, **task_meta_data()** results in the following output:

Size of L1: **L1 = 2**

Size of L2: **L2 = 4**

Loop depth: **loop_depth = 4**

Inner task count for L1: **inner_task_count(1) = 4; inner_task_count(2) = 4**

This code generates a total of ten tasks for the H_2O test case. Table 4.1 shows the complete list of tasks that has been generated by `task_meta_data()`. This list matches the “real tasks” that are shown in the VAMPIR trace file from the original code, see Figure 4.3.

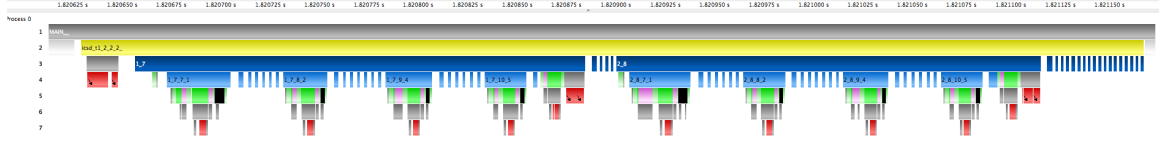
Table 4.1: Result of `task_meta_data( loop-depth, L2-size, L1-size )` for H_2O input.

DFILL 1	(1,1,1) = 1	(2,1,1) = 7		
GEMM 1-1	(1,1,1) = 1	(2,1,1) = 7	(3,1,1) = 7	(4,1,1) = 1
GEMM 1-2	(1,2,1) = 1	(2,2,1) = 7	(3,2,1) = 8	(4,2,1) = 2
GEMM 1-3	(1,3,1) = 1	(2,3,1) = 7	(3,3,1) = 9	(4,3,1) = 4
GEMM 1-4	(1,4,1) = 1	(2,4,1) = 7	(3,4,1) = 10	(4,4,1) = 5
DFILL 2	(1,1,2) = 2	(2,1,2) = 8		
GEMM 2-1	(1,1,2) = 2	(2,1,2) = 8	(3,1,2) = 7	(4,1,2) = 1
GEMM 2-2	(1,2,2) = 2	(2,2,2) = 8	(3,2,2) = 8	(4,2,2) = 2
GEMM 2-3	(1,3,2) = 2	(2,3,2) = 8	(3,3,2) = 9	(4,3,2) = 4
GEMM 2-4	(1,4,2) = 2	(2,4,2) = 8	(3,4,2) = 10	(4,4,2) = 5

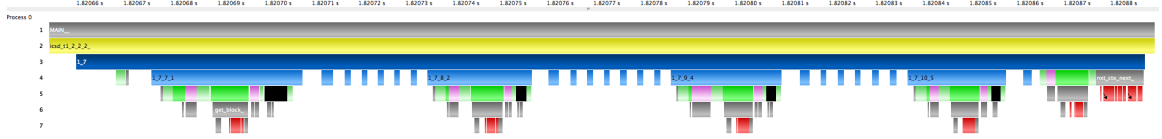
Step 2: The Parameterized Task Graph

The second step is the execution of the Parameterized Task Graph 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 meta-data vectors populated by the first step; this allows the execution space of the modified subroutines over PARSEC to match the original execution space of these subroutines. Also, using the meta-data structures, PARSEC accesses the program data directly from memory, without using GA.

The example in Figure 4.4 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 in multiple parallel chains, with each chain containing multiple **GEMMs** that execute sequentially. In this



One iteration of t1_2_2_2: showing entire execution space.



One current task: four GEMMs computed sequentially for H_2O .

Function	Min Inclusive Time
MAIN_	233.271 μ s
icsd_t1_2_2_2_	233.271 μ s
1_7	227.322 μ s
tce_sort_2_	1.750 μ s
nxt_ctx_next_	10.138 μ s
ma_push_get_	3.167 μ s
ma_pop_stack_	1.192 μ s
dfill_	568.776 ns
add_hash_block_	7.026 μ s
1_7_9_6	1.041 μ s
1_7_9_5	1.946 μ s
1_7_9_4	23.485 μ s
1_7_9_3	1.161 μ s
1_7_9_2	1.174 μ s
1_7_9_1	1.257 μ s
1_7_8_6	1.051 μ s
1_7_8_5	1.121 μ s
1_7_8_4	1.208 μ s
1_7_8_3	2.120 μ s
1_7_8_2	22.067 μ s
1_7_8_1	1.170 μ s
1_7_7_6	1.066 μ s
1_7_7_5	1.120 μ s
1_7_7_4	1.063 μ s
1_7_7_3	1.123 μ s
1_7_7_2	2.330 μ s
1_7_7_1	32.014 μ s
tce_sort_4_	1.831 μ s
tce_sort_2_	2.383 μ s
tce_restricted_4_	145.069 ns
tce_restricted_2_	167.625 ns
ma_push_get_	3.451 μ s
ma_pop_stack_	2.306 μ s
get_hash_block_ma_	2.417 μ s
tce_hash_	217.161 ns
get_hash_block_	8.011 μ s
tce_hash_	229.987 ns
get_block_	7.074 μ s
dgemm_	4.753 μ s
1_7_10_6	1.766 μ s
1_7_10_5	20.775 μ s
1_7_10_4	1.318 μ s
1_7_10_3	1.216 μ s
1_7_10_2	1.077 μ s
1_7_10_1	1.166 μ s

Color Legend.

Figure 4.3: Minimal Vampir performance trace of CC t1_2_2_2 kernel for H_2O .

```

1 GEMM(L1, L2)
2   L1 = 0..(mtdata->size_L1-1)
3   L2 = 0..(mtdata->size_L2-1)
4
5   A_reader = find_last_segment_owner(mtdata, 0, L2, L1)
6   B_reader = find_last_segment_owner(mtdata, 1, L2, L1)
7
8   : descRR(L1)
9
10  READ  A <- A input_A(A_reader, L2, L1)
11  READ  B <- B input_B(B_reader, L2, L1)
12
13  RW C <- (L2 == 0) ? C DFILL(L1)
14         <- (L2 != 0) ? C GEMM(L1, L2-1)
15         -> (L2 < (mtdata->size_L2-1)) ? C GEMM(L1, L2+1)
16         -> (L2 == (mtdata->size_L2-1)) ? C SORT(L1)
17
18 ; mtdata->size_L1-L1 + P
19 BODY {
20   dgemm('T', 'N', ...
21 }

```

Figure 4.4: PTG for GEMM tasks organized in a chain.

PTG, L1 corresponds to the chain number and L2 corresponds to the position of a GEMM inside the chain to which it belongs. As can be seen in [Figure 4.4](#), the number of chains and the length of each chain do not have to be known a priori. PARSEC will dynamically look them up from the meta-data 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), 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 coarse grain parallel programs, but the learning curve that must be climbed comes with rewards for those who climb it. In order to change the the organization of the `GEMMs` from a serial chain to a parallel execution followed by a reduction, the four lines (13-16), which define the dataflow of matrix `C`, would have to be replaced by the following single line depicted in [Figure 4.5](#).

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

Figure 4.5: PTG snippet for parallel `GEMM` tasks.

Due to the PTG representation, all communication becomes implicit and is handled automatically by the runtime without user intervention. 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 manage large numbers of outstanding messages and duplicate buffers which quickly becomes a 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 such as `GET_HASH_BLOCK()` (see [Figure 4.2\(a\)](#)).

4.1.1 Parallelization and Optimization

One of the main reasons we are porting `CC` over `PARSEC` is the ability of the latter to express tasks and their dependencies at a finer granularity, as well as the decoupling of work tasks and communication operations that enables us to experiment with more advanced communication patterns than serial chains. 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 $\beta = 1$). Since matrix addition is an associative and commutative operation, the order in which the **GEMMs** are performed does not bear great significance *, as long as the results are atomically added. This enables us to perform all **GEMM** operations in parallel and sum the results using a binary reduction tree. Figure 4.6 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 [45]. For the sake of completeness, Figure 4.7 depicts such a chain where eight **GEMM** operations are computed sequentially.

In addition, the sequential steps are matrix additions, not **GEMM** operations, so they are significantly faster, especially for larger matrices. Reductions only apply to **GEMM** operations that execute on the same node, thus avoiding additional communication. 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 grows fast.

It is important to note, the original version of the code performs an atomic accumulate-write operation (via calls to `ADD_HASH_BLOCK()`) at the end of each chain. Since our dataflow version of the code computes the **GEMMs** for each chain in parallel, we eliminate the **global** atomic GA functionality and perform direct memory access instead, using **local** atomic locks within each node to prevent race conditions. The choice of our implementation, discussed in this chapter, is based on earlier investigations presented in [21, 36], where we compare the performance of different variants of the modified code and explain the different behaviors that lead to the differences in performance.

* 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.

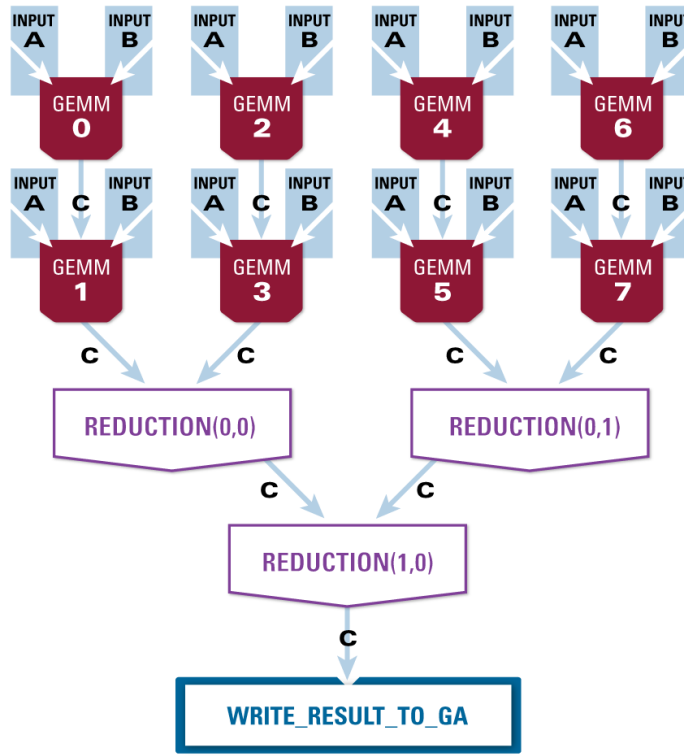


Figure 4.6: Parallel GEMM operations followed by reduction.

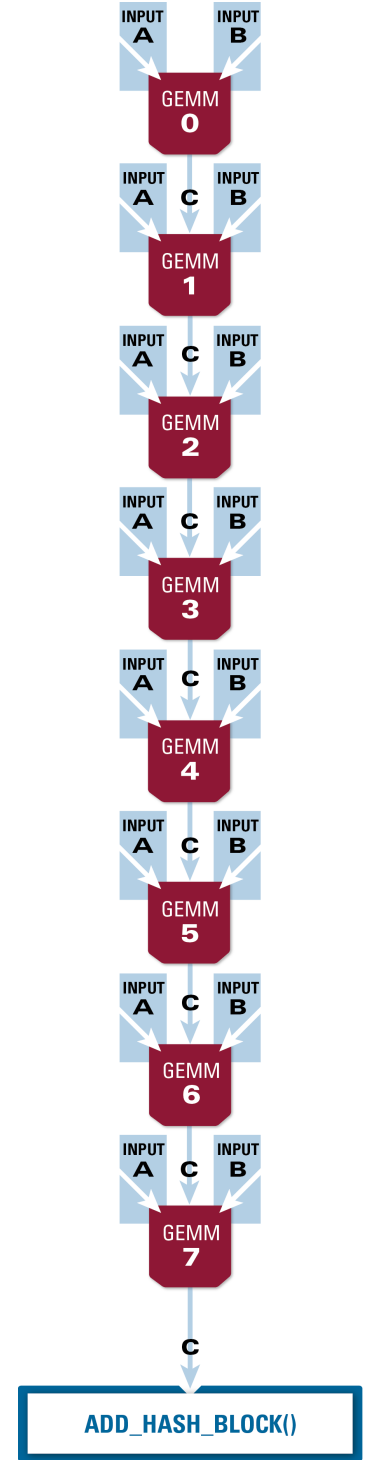


Figure 4.7: Chain of GEMM operations computed sequentially.

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 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 [37].

4.1.2 Additional Levels of Parallelism

It is important to note that our work of converting the entire NWChem CC “FORTRAN plus Global Arrays” implementation into a dataflow form has been of incremental nature in order to preserve the original behavior of the code. This allowed us to initially focus only on the most time-consuming and most significant subroutines (the 29 heavy **GEMM** routines), and more importantly, execute them over PARSEC without having to convert the entire CC module. The successful conversion of these 29 kernels has proven to be very beneficial, resulting in a performance improvement of more than a factor of two in the execution of the entire CC component of NWChem. This result justifies our conclusion that the utilization of dataflow-based execution of CC methods enabled more efficient and scalable computation.

After completion of the dataflow implementation within each of the four levels, the next increment of work that we are currently pursuing focuses on implementing dataflow between the levels. From the DAG in Figure 2.2 it becomes very apparent that not every subroutine in the upper levels has to wait for the completion of all subroutines in the lower levels. For instance, `t2_2_2()` in Level 3 only depends on the data coming out of `t2_2_4()`, `t2_2_5()`, and `t2_2_2_2()` in Level 2; while these three only depend on 10 [†] (of the 29) subroutines in Level 1. Instead of putting the execution of the tasks that comprise `t2_2_2()` on hold until all subroutines in Level 1 and Level 2 are completed, the output of the tasks that flow into `t2_2_2()` can

[†]`t2.2.1()`, `t2.2.6()`, `t2.2.4.1()`, `t2.2.4.2()`, `t2.2.5.1()`, `t2.2.5.2()`, `t2.2.2.1()`, `t2.2.2.3()`, `t2.2.2.2.1()`, `t2.2.2.2.2()`

be passed as soon as it becomes available. The return of enabling dataflow between levels is twofold. First, it increases the level of parallelism even more; and second, it enabled the freedom to choose a certain placement of tasks. For example, tasks of subroutine `t2_2_2()` in Level 3 can be computed where the Level 2 subroutines, whose output flows into `t2_2_2()`, store their data.

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

4.2 Algorithmic Variations

When we designed the PARSEC implementation of CC it became apparent that several operations can be ordered in multiple ways and still preserve the semantics of the original algorithm. This is the case because these operations – i.e., matrix addition, data remapping and updating of data in memory – are associative and commutative. This section presents several variations of our algorithm, each using a different ordering of tasks, and we discuss the impact of these variations on performance. For this investigation, we focus our discussion on one of the CC subroutines, `icsd_t2_7()`. We note that the final result (correlation energy) computed by the different variations matched up to the 14th digit.

The choice with the highest impact on performance relates to the parallelism of the GEMM operations. As mentioned earlier, the original CC code has the GEMM

operations organized in chains. Different chains execute in parallel, independently of one another, while the GEMMs within a chain execute in serial order and they all use the same output, C . However, none of the input matrices (A, B) of any GEMM operation ever overlaps with the output of any other GEMM of any chain in the subroutine. In other words, all input matrices are read-only within the scope of a subroutine. Matrix multiplication itself is noncommutative (i.e., $A \times B \neq B \times A$), however the GEMM kernel does not perform only a multiplication, but also an addition, $C \leftarrow A \times B + C$. The matrix addition, is both commutative and associative. Therefore, if we segment the single chain of GEMMs into a number of shorter chains that each works on private memory and then we accumulate these partial results through a reduction tree, the semantics of the original code will be preserved. This variation of the algorithm, shown graphically in [Figure 4.6](#), increases available parallelism, but decreases locality, since different chains work on different private C matrices.

The height of the shorter chains can vary from one (for maximum parallelism) to the height of the original chain (for maximum locality). In this section we consider the two extreme cases. Another side-effect of segmenting the chains is an increased readiness of work. In the single chain approach of the original code the input matrices A and B of the **first** GEMM in the chain must be fetched for **any** work to start. In the modified version useful work can start as soon as the input matrices of **any** GEMM become available.

The significance of this difference can be appreciated if we consider that all matrices are stored and managed by the Global Arrays library that distributes data among nodes in an application insensitive manner. For example, when the input molecule is large (as is the case with the beta-carotene we use here) it is often the case that the data that constitutes matrix A , or B is segmented and distributed on more than one nodes. Therefore, waiting for the data of a particular operation $GEMM^k$ – the one dictated by the ordering imposed by the chain – is guaranteed to be slower than waiting for the data of any $GEMM^i$ (for $0 < i < \text{chain_length}$) when

the input matrices, A^k and B^k , of $GEMM^k$ are distributed between nodes and there is at least one i for which the input matrices A^i and B^i are stored on one node only (and ideally the node that performs that $GEMM^i$ operation).

As a result of the increased readiness of work, the variations of the algorithm where the GEMM operations are performed in parallel have a lower starting overhead—an expectation that is confirmed by our experiments, discussed in [Section 4.2.4](#).

In [Figure 4.6](#) we depict the work needed to write the output of a set of GEMMs back into the Global Array as a single box named `WRITE_RESULT_TO_GA`. We employed this abstraction to focus on the parallelism of the GEMMs. Nonetheless, the writing of the results back to the Global Array is a non trivial operation that can also lead to several variations of the algorithm. In the original code of subroutine `icsd_t2_7()`, after the last GEMM in a chain, there are four `IF` branches each containing a call to a subroutine called `SORT_4()`. This subroutine – which we will refer to as “SORT” hereafter – takes as input the output of the chain of GEMMs, C , and outputs a modified version of it that we will call C^{sorted} . In the interest of accuracy, we should mention that despite its name, the SORT operation does not perform actual sorting of the data, but rather a remapping of the C matrix where the elements of the input matrix are shuffled to different locations in the output matrix regardless of the value they hold. Each of the four SORT operations is followed by a call to `ADD_HASH_BLOCK()` that adds the data into the Global Array (i.e, performs the operation $C^{orig} += C^{sorted}$ where C^{orig} is the data that is stored in the Global Array before the call to `ADD_HASH_BLOCK()`). Hereafter, we will refer to this operation as `WRITE`. Interestingly, the predicates of the four `IF` branches are not mutually exclusive. One can easily see in the code below – which contains snippets of the `IF` statements of subroutine `icsd_t2_7()` – that when the variables that are being compared are equal, then multiple of these `IF` statements will evaluate to true.

```
IF ((p3b .le. p4b) .and. (h1b .le. h2b)) ...
IF ((p3b .le. p4b) .and. (h2b .le. h1b)) ...
IF ((p4b .le. p3b) .and. (h1b .le. h2b)) ...
```

```
IF ((p4b .le. p3b) .and. (h2b .le. h1b)) ...
```

As a result, depending on the value of these variables, the original code will perform one, two, or four SORT operations – each followed by a WRITE – and will do so in a serial way. In our code, we implemented different variations of the WRITE_RESULT_TO_GA work, with each one having a different level of parallelism in performing the sorting and writing of the data.

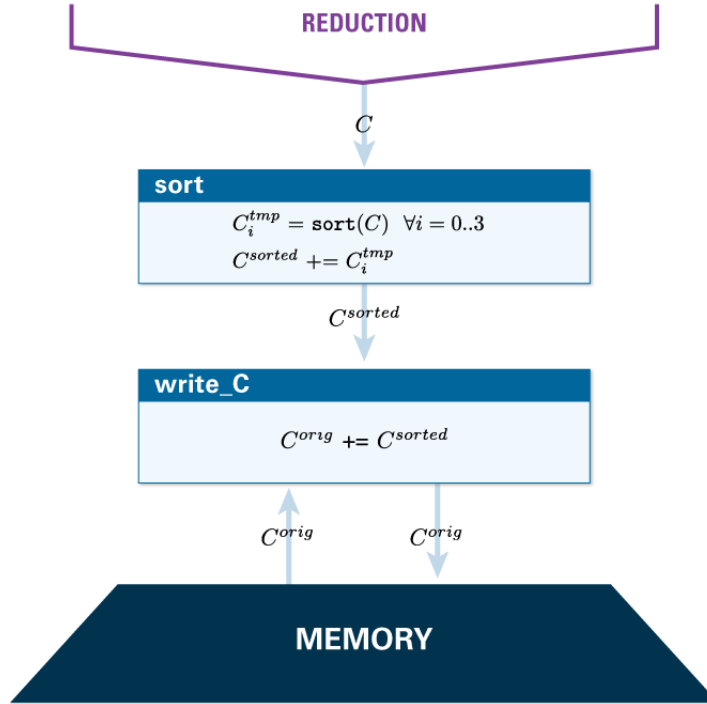


Figure 4.8: Serialized sort and single write.

The simplest variation, which is depicted in [Figure 4.8](#), executes all operations in serial, but in a different order than the original CC code. Namely, the output of the GEMMs, C , is passed to a task – called **SORT** – that contains four consecutive calls to the **SORT_4()** kernel, each one guarded by the corresponding **IF** statement. Each call, i , uses a different temporary matrix, C_i^{tmp} as output. After each call, C_i^{tmp} is accumulated into a master matrix C^{sorted} – which was initialized to zero before the first call to **SORT_4()**. When the **SORT** task is done, the resulting matrix C^{sorted}

is passed to the task `WRITE_C`. This task reads the data that is currently stored in the target memory location C^{orig} and updates it by performing the addition: $C^{orig} += C^{sorted}$. To prevent race conditions between different threads the `WRITE_C` task performs the reading, adding, and storing into memory atomically by protecting all this code with pthread mutexes. Clearly, changing the order of operations and adding all the temporary matrices to one another before adding them to the memory does not alter the semantics of the original code since addition is commutative and associative.

In this variation of `WRITE_RESULTS_TO_GA`, the work exhibits the lowest amount of parallelism, and the highest level of data locality, since matrices C and C^{sorted} are read and written multiple times in quick succession by the same task (`SORT`) and thus the same operating system thread, since in PARSEC tasks do not migrate between threads after they have started executing.

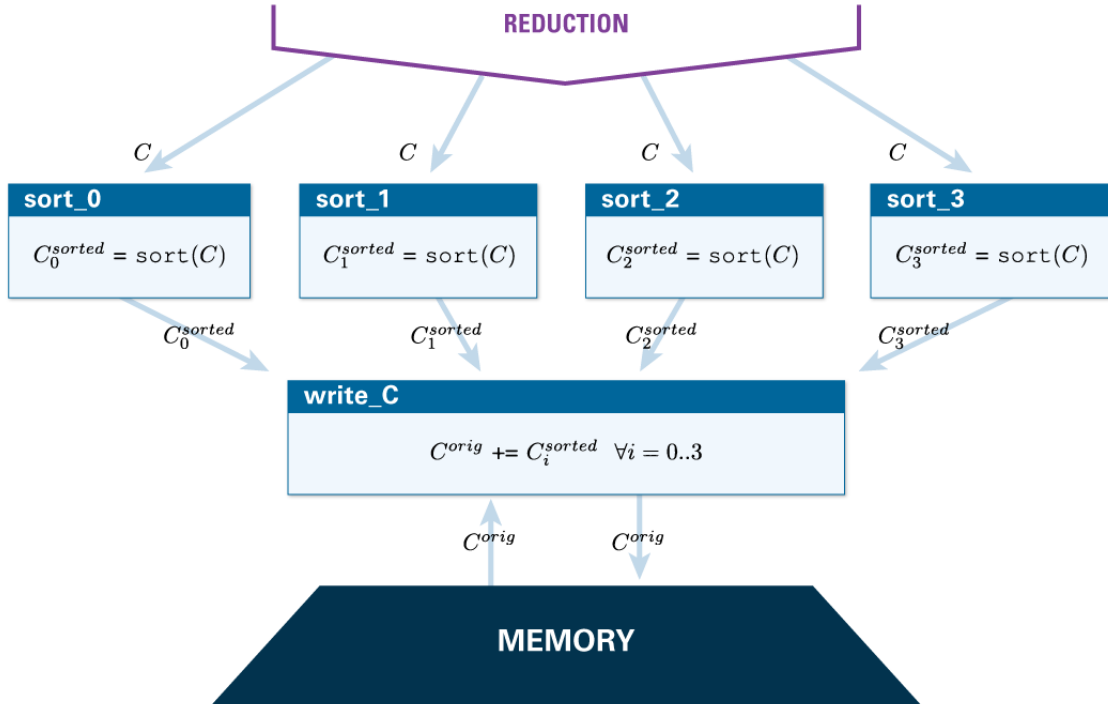


Figure 4.9: Parallelized sort and single write.

The next variation involves parallelization of the `SORT` task. Namely, as shown in Figure 4.9, the code now implements four different tasks, `SORT_0`, `SORT_1`, `SORT_2`, and `SORT_3`. The execution of each task is guarded by the same predicate as the corresponding `IF` branch in the original code. Each `SORTi` task receives the same C matrix as input and stores the output of the `SORT_4()` subroutine into a private C_i^{sorted} matrix. Then, it forwards this private matrix to the `WRITE_C` task. The latter receives one, two, or four input sorted matrices (depending on the values of the predicates), and uses them to update the data in memory: $C^{orig} += C_i^{sorted} \forall i = 0..3$. This variation has increased parallelism in comparison to the previous scenario, since the `SORT` operations now happen in parallel, but it has reduced data locality and additional memory requirements, since each `SORTi` task needs to allocate a C_i^{sorted} matrix. Furthermore, this variation can lead to more idle time since it has a longer atomic operation. This is the case because the work performed by the `WRITE_C` task is treated as a critical region that is protected by mutexes in order to run atomically, and this variation of the code assigns more work to the `WRITE_C` task than the previous case.

In order to address the last concern regarding the length of the critical region, and increase parallelism even further, we created another variation that we show in Figure 4.10. In this variation, there are multiple `WRITE_Ci` tasks, in addition to the multiple `SORTi` tasks. Each `SORTj` task (that executes) sends its output matrix to the corresponding `WRITE_Cj` task, which updates the data in memory using only this matrix: $C^{orig} += C_j^{sorted}$. This variation has the least amount of data locality, but exhibits maximum parallelism without increasing the length of the critical regions.

4.2.1 Exporting PARSEC data to Global Arrays

In the discussion above we analyzed the high level organization of the tasks that sort and write the data into the memory, but we abstracted away the details of how PARSEC actually accesses the memory addresses that correspond to the Global

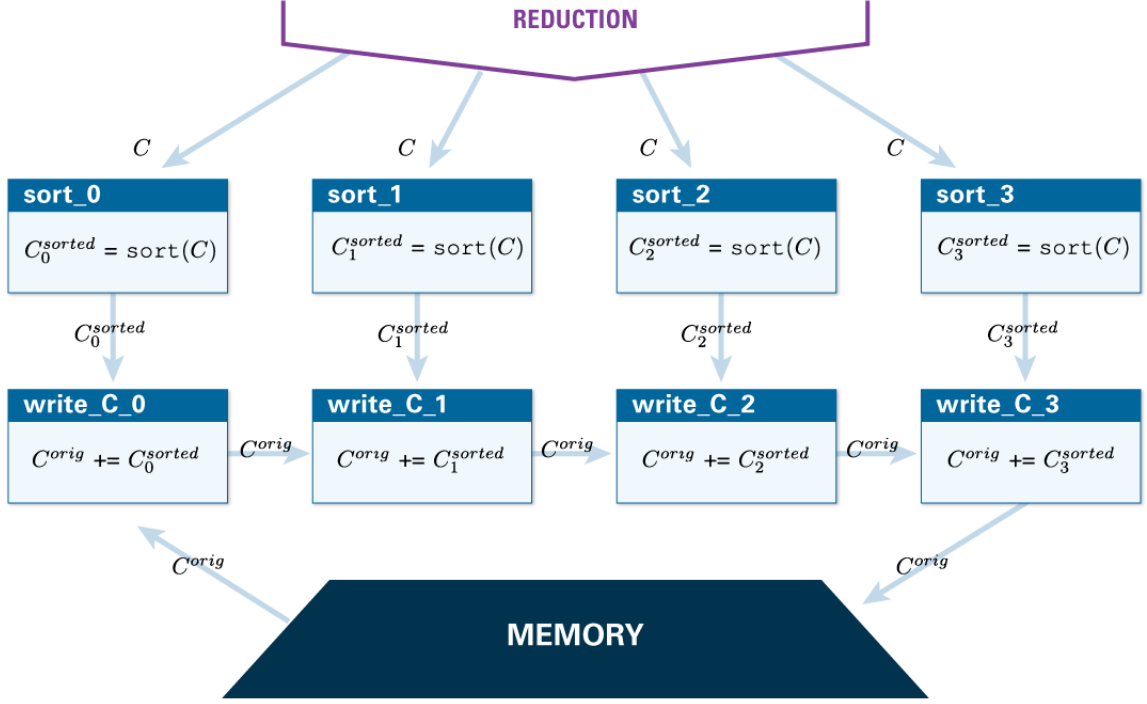


Figure 4.10: Parallelized sort and write

Array data, which in the general case can be distributed between multiple nodes. For simplicity, we will explain the process using the algorithmic variation where there is only one `SORT` and one `WRITE_C` task type, shown in Figure 4.8, but the same logic applies to all variations.

A GA matrix C^{orig} can potentially be split between the memories of different nodes participating in the execution of the program. In Figure 4.11 we depict a matrix as being split between the memory of nodes $node_i$, $node_{i+1}$, and $node_{i+2}$.

As a result, although there is only one class of `WRITE_C` tasks, in this example there need to be three instances of this task class, each running on one of the three nodes that contains data in order to access and modify this data. In the figure we depict this with `WRITE_C(i)`, `WRITE_C(i+1)` and `WRITE_C(i+2)`. Consequently, the transmission of data between the `SORT` task and the different `WRITE_C` task instances

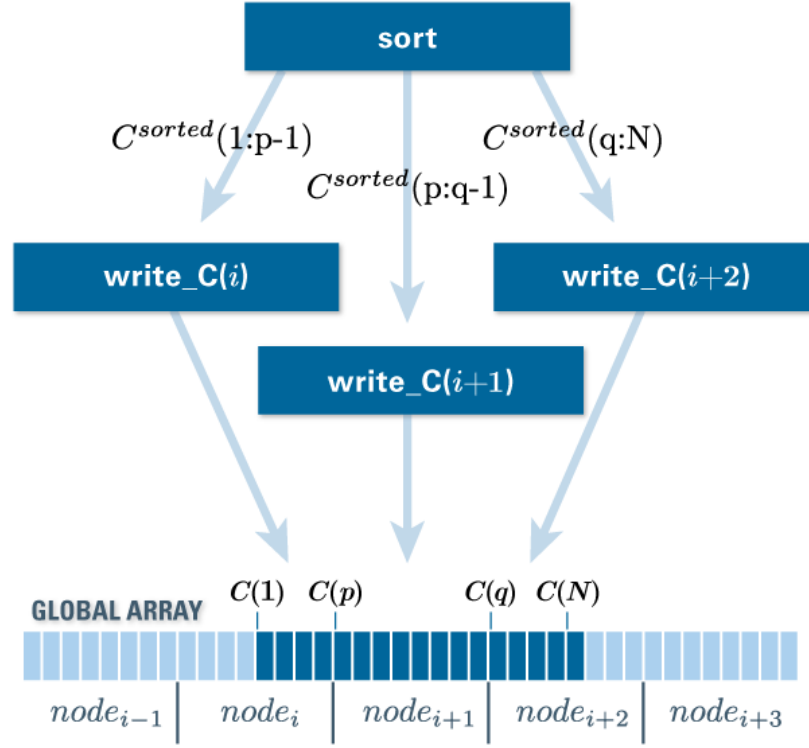


Figure 4.11: Exporting data to GA.

is implemented such that each task instance only receives the data that is relevant to the node on which the task instance executes.

Achieving all this data management is possible because PARSEC abstracts away a lot of the details. During the inspection phase (i.e., before any tasks start running) we query the Global Arrays library regarding the actual location of the program data using calls such as `ga_distributed()` and `ga_access()`. The information that we acquire from these calls is passed to PARSEC along with a unique ID for each matrix. As a result, the PTG is Global Array agnostic and only uses these unique IDs to refer to data relying on PARSEC for the management of the ID-to-node and ID-to-pointer mapping.

4.2.2 Using task priorities in PaRSEC

PARSEC includes multiple task scheduling algorithms, each designed to maximize a different objective function, i.e., cache reuse, load balancing, etc. The default scheduling algorithm, which is what we used for the performance experiments in this paper, tries to achieve a balance between several objective functions and also takes task priorities into consideration, if such have been defined by the developer. Task priorities are taken into account by the scheduler when a set of available tasks are considered for execution, and they only have a relative meaning. That is, between two available tasks, the one with a higher priority will execute first.

In this paper we explicitly set task priorities for all the variants of our algorithm, except for one, in order to study the effect of priorities on the behavior of the program. To set task priorities in PARSEC the developer has to explicitly add a line in the PTG representation of each task class. This line starts with a semicolon and is followed by an expression that can be as simple as a numeric literal, or as complex as an arbitrary function of any variable that is available to the task class (i.e., global and local variables, as well as the parameters of the task class). In the implementation presented in this paper, we assigned to all task classes priority expressions that are decreasing functions of the chain number[‡]. To differentiate the priority of different task classes within a single chain, we use constant offsets. We assign a higher priority to the tasks that read the input data (matrices A and B) from the Global Array, by giving them the highest offset (+5), then follow the tasks that perform the GEMM operation with offset +1, and all other tasks classes do not have an offset. The general expression used in the task priorities is:

$$\text{max_L1} - \text{L1} + \text{offset} * \text{P}$$

[‡]As we mentioned previously, during the inspection phase we record which chain (of the original code) each GEMM corresponds to. In our code the chain number is one of the parameters of each task and it has no effect on the parallelism between those tasks. It serves only as a means to differentiate between tasks that are working toward the computation of the same final C matrix, or different ones.

Where `max_L1` is the total number of chains, `L1` is the chain number of each chain, and `P` is the number of nodes participating in the execution. This scheme has the following consequences:

- All tasks of a given task class (i.e., GEMM) that participate in the same chain will have the same priority.
- All tasks of a given task class that belong to chain i will have higher priority than any task of the same task class that belongs to chain j where $j > i$.
- Since we assign an offset of +5 to the tasks that read the input data, and an offset of +1 to the GEMM tasks, there will always be at least $5 * P$ reader tasks that executed before each GEMM task. Therefore, there is a data prefetching pipeline of depth $5 * P$.

4.2.3 Load Balancing

The original NWChem code aims to achieve load balancing through a work stealing scheme that is often referred to as NXTVAL [52]. This approach relies on the updating of a global atomic variable provided by the Global Arrays library for ensuring that each MPI rank will atomically acquire a single unit of work each time. The unit of work in the original NWChem code (a whole chain) is much coarser than in our PARSEC enabled version. Nevertheless, requiring that every MPI rank has to update a single global atomic variable for every unit of work is not a scalable approach.

In the PARSEC enabled code presented in this paper we took the opposite approach. We performed a static, round-robin work distribution between nodes and allowed PARSEC to perform dynamic work stealing within each node. This approach may lead to higher load imbalance than the work stealing approach between nodes, but incurs zero overhead in the critical path. Our performance results validate our choice.

4.2.4 Performance Evaluation for Algorithmic Variations

In this section we will discuss performance results that we obtained by executing the original NWChem code as well as five variants of our PARSEC implementation, as discussed in [Section 4.2](#). In particular we timed the following five variants of the algorithm:

- v1.** GEMMs are organized in a serial chain, but SORTs and WRITES are parallel. Priorities are a decreasing function of the chain number.
- v2.** GEMMs and SORTs are parallel, but there is one WRITE. No priority is set for any task.
- v3.** GEMMs, SORTs, and WRITES are all parallel. Priorities are a decreasing function of the chain number.
- v4.** GEMMs and SORTs are parallel, but there is one WRITE. Priorities are a decreasing function of the chain number.
- v5.** GEMMs are parallel, but there is one SORT and one WRITE. Priorities are a decreasing function of the chain number.

[Figure 4.12](#) shows the execution time of the original code and the different algorithmic variants discussed above. All experiments used beta-carotene as the input molecule, in 6-31G basis set composed of 472 basis set functions, and run on a 32 node partition of the Cascade cluster at the Pacific Northwest National Laboratory. To improve readability we show the execution time of the PARSEC variants only for 1, 3, 7, and 15 cores per node in the form of boxes with different shades (and different border style), stacked next to one another. In contrast, the execution time of the original version is shown for every number of cores/node and is shown as a (green) line with circular points. In the following text we discuss observations that can be made by studying this graph.

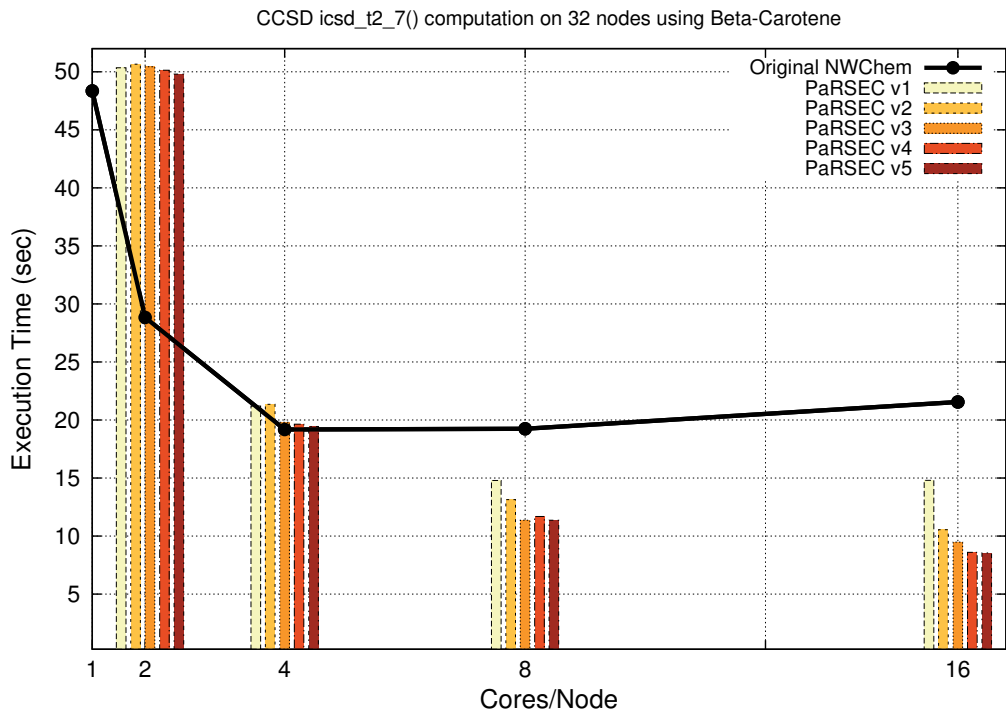


Figure 4.12: Comparison of algorithm variations and original code.

The original code scales fairly well up to three cores/node (achieving a speedup of 2.35x over the one core/node run) but tapers off after that, achieving little additional improvement until the best performance at seven cores/node – which achieves a speedup of 2.69x over the one core/node case. After this point the performance deteriorates, although not significantly. In contrast, the PARSEC code scales much better with all variants, except v1, improving their performance all the way up to 15 cores/node.

PARSEC outperforms the original code as soon as three cores per node are used, and shows a significant performance improvement. Namely, at 15 cores/node the best PARSEC variant (v5) achieves a speedup of 2.1x over the fastest run of the original code at seven cores/node.

The different variants of the PARSEC code show little difference when few cores per node are used, but diverge significantly when the machine reaches saturation.

Namely, at 15 cores/node, the fastest PARSEC variant is 1.73x faster than the slowest variant.

The slowest PARSEC variant, v1, is the one where the GEMMs are organized in a serial chain (mimicking the behavior of the original code). This indicates that parallelism between GEMMs is more significant than locality for the performance of this program, despite the parallelism that already exists between different chains.

The fastest PARSEC variant, v5, is the one where the GEMMs are parallel but the SORT operation and the critical section that writes back to the memory are serialized. This is not an obvious outcome, mainly because the critical region of the WRITE is protected by a mutex that is shared by all threads in the node and used by all WRITE operations on the node. As a consequence, one could expect that smaller critical regions would lead to better interleaving of different threads and thus better performance. However, variant v3, which implements the WRITE in parallel, delivers worse performance than v5 for all core counts and especially when 15 cores/node are used. We attribute this behavior to two factors. First, the better data locality of v5 – which is especially important for an operation such as WRITE that does nearly no work and is memory bound. Second, while the serialized version has a longer critical region, each `write_C` task locks and unlocks the mutex only once per chain, as opposed to the parallel writing scheme of v3 where there are up to four `write_C_i` tasks per chain increasing the number of the system wide operations required to lock and unlock the mutex that protects the critical region.

Comparing variants v2 and v4, which only differ in the task priorities, we can see the importance of task priorities with respect to performance. Furthermore, by comparing v2 against all other variants, we can deduce that priorities are the single most important design decision after the parallelism of the GEMM operations, since the v2 variant performs worse than all other variants except for v1. To substantiate this point further, we present the execution trace of variants v4 and v2 in [Figure 4.13](#) and [Figure 4.14](#), respectively. These traces were generated using PARSEC’s native performance instrumentation module.

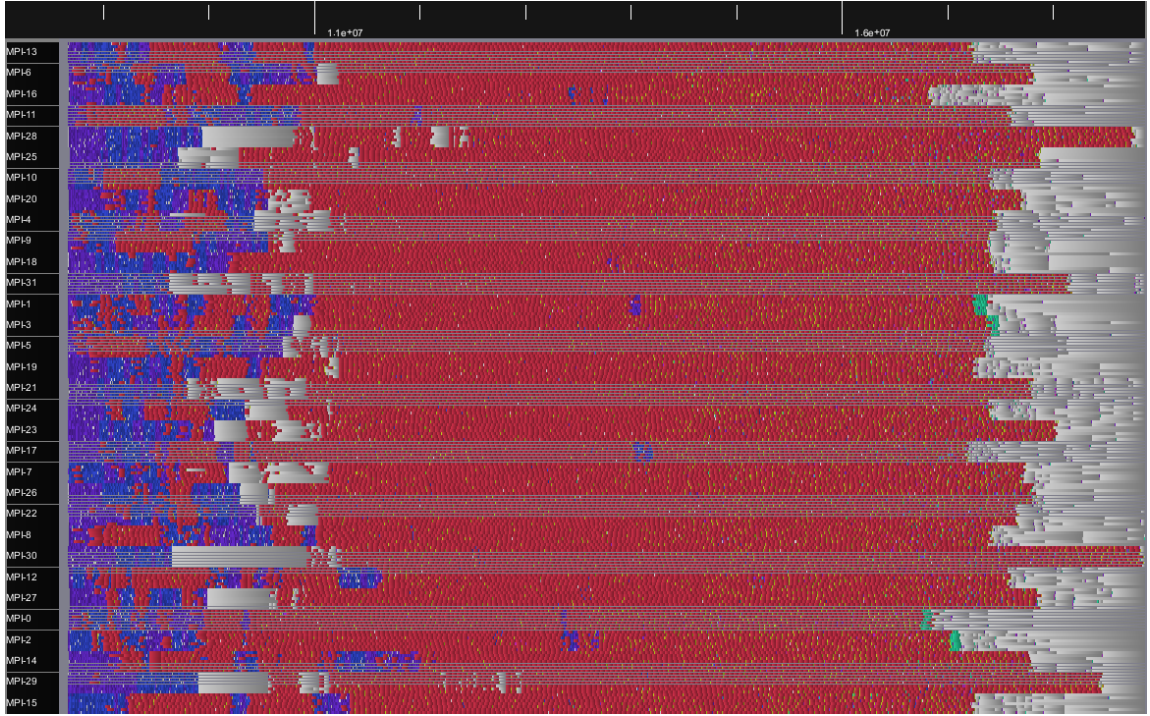


Figure 4.13: Trace of v4 (priority decreasing with chain number).

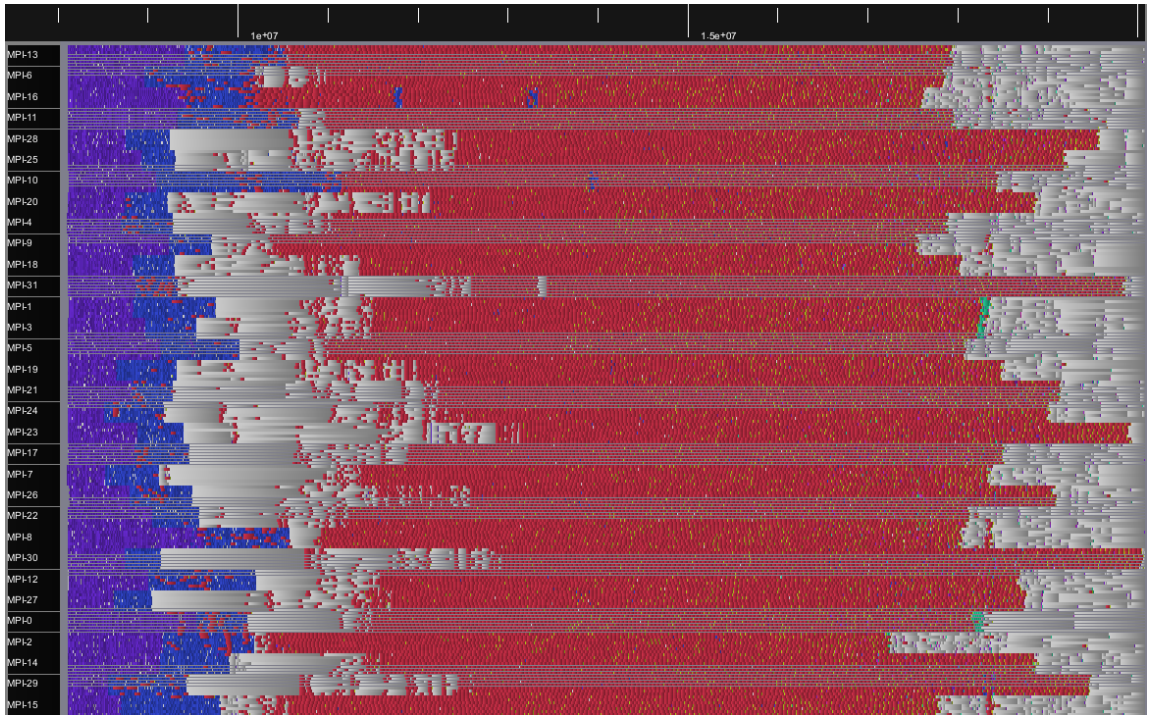


Figure 4.14: Trace of v2 (no task priorities).

In these traces, each row represents a thread (out of 7 threads per node) and each group of seven adjacent rows represents a node. The horizontal axis represents time and the two figures are not at the same scale. Instead, each trace shows all the events that occurred from the beginning to the end of the program’s execution, regardless of the length of the execution. Different colors represent different task classes and the colors are consistent between the two figures. Red represents GEMM operations, blue represents the reading of input matrix A , purple represents the reading of B , yellow represents the reductions and light green represents the writing back to the global array. Finally, grey represents idle time.

The traces make it abundantly clear that variant v2 – which lacks task priorities – has too much idle time in the beginning. The reason relates to the way PARSEC handles communication. In PARSEC, tasks do not explicitly perform communication, rather they express their communication needs to the runtime system by specifying their dependencies to other tasks. The actual data transfer calls are issued by the runtime system (and in the case of the code discussed in this paper, data transfer calls are issued by a specialized communication thread that runs on a dedicated core). When variant v2 starts running, PARSEC discovers that all the tasks that read the input matrices A and B are ready for execution, since they do not depend on any previous tasks. The lack of priorities allows PARSEC to execute all these tasks which enqueue their communication requests for the communication thread to fulfill, and return immediately. As a result, the network is flooded with communication requests between all nodes that participate in the execution, and there is no computation with which to overlap this communication. In contrast, variant v4 defines priorities that decrease with the chain number. This means that the GEMMs of the early chains have a higher priority than the read tasks of all but the first few chains. However, no GEMMs can execute before their input data has been transferred. Therefore, while the early communication is ongoing, PARSEC will keep executing read tasks, but as soon as data starts arriving at a node, the priorities of the different tasks will guide PARSEC to schedule work (GEMMs) interleaved with communication (reading of A

and B). This way, the idle time at the beginning is smaller (as is evident in the trace) and a significant part of the communication is overlapped with computation, leading to a faster overall execution.

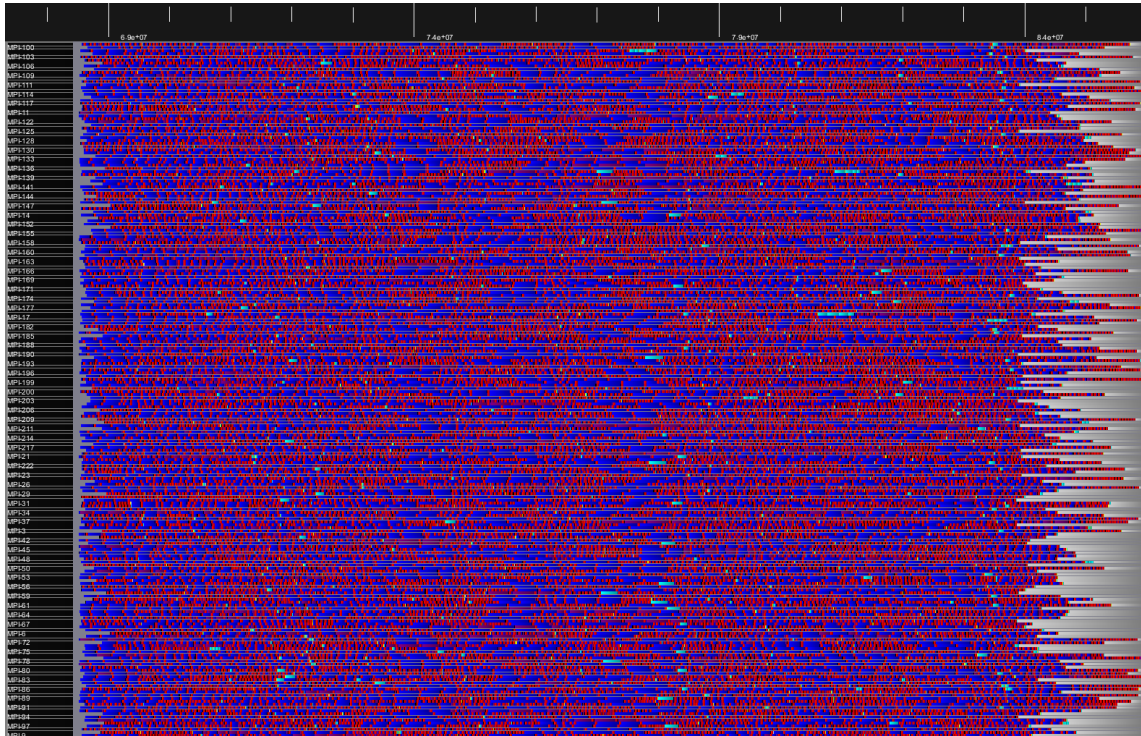


Figure 4.15: Trace of original NWChem code.

Figure 4.15 shows the trace of the original NWChem code. Once again, the horizontal axis is not at the same scale as the other traces, red represents GEMMs and blue represents communication, although in this case it is the cost of the subroutine `GET_HASH_BLOCK()`. Although this trace depicts the performance of the original code, it was also generated using PARSEC’s performance instrumentation functionality. This is possible because PARSEC exports this functionality as an API that can be used to instrument arbitrary code.

This trace tells a very different story from the previous ones. Here, communication is interleaved with computation, however it is not overlapped. This is an artifact of the way the original NWChem code is structured. Namely, the calls to the communication

subroutine `GET_HASH_BLOCK()` that fetches the input data (A and B) from the Global Array into a local buffer are issued immediately preceding the call to the GEMM kernel. Therefore, regardless of the underlying communication library, or network conduit, the communication is not overlapped with the computation, because it is not given a chance to do so. There is no computation in the code between the point where the data transfer starts and the point where the data is needed.

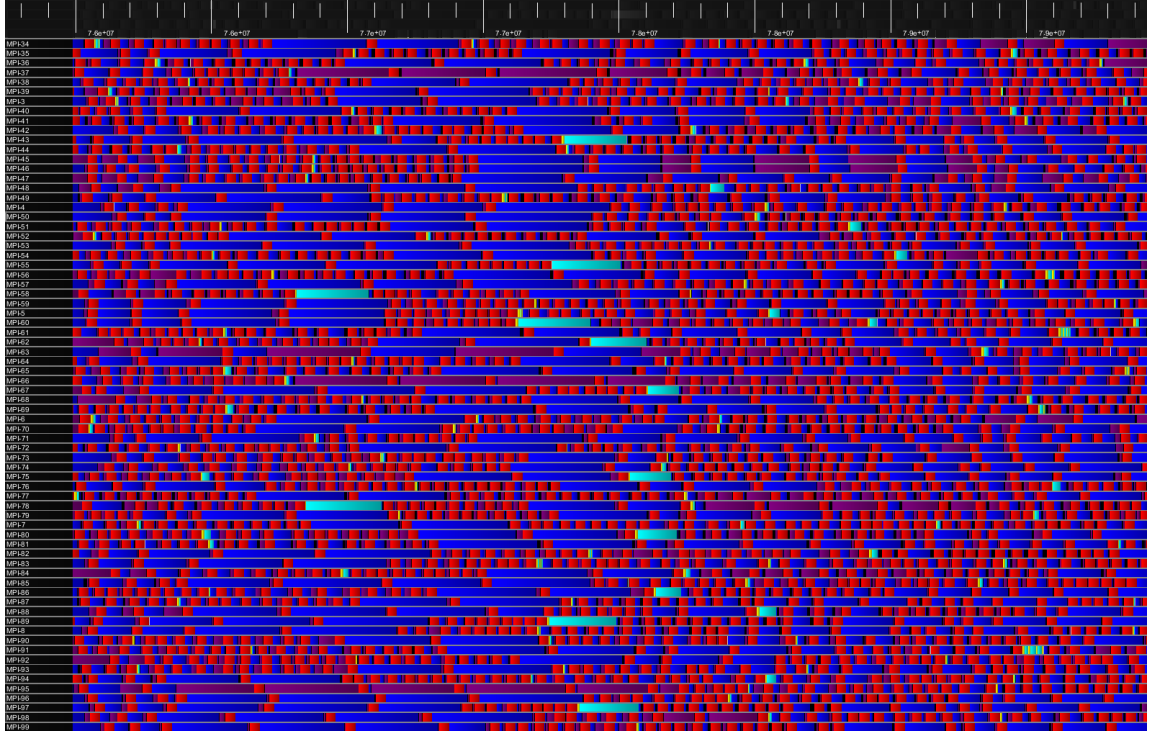


Figure 4.16: Zoomed in on execution trace of original NWCHEM code.

In [Figure 4.16](#) we show a part from the bottom middle of the previous trace zoomed in so that individual tasks can be discerned. In this figure the lack of communication computation overlapping is evident by the length of the blue, purple and light green rectangles in comparison to the length of the red triangles.

In summary, porting the `icsd_t2_7()` subroutine of NWCHEM over PARSEC enabled us to modify the behavior of the code and explore the trade-offs between locality and parallelism in the execution of the GEMM, SORT and WRITE

operations. It also enabled us to trace the execution of the code, so that we can understand the different sources of overhead. As a result, we managed to produce multiple variants of the code that outperform the original code with our best variant achieving a speedup of 2.69x on 32 nodes.

4.3 Dataflow as a programming paradigm for CC

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

4.3.1 Communication computation overlapping

The original code is written in FORTRAN[§] and makes calls to the Global Arrays (GA) toolkit for communication and synchronization purposes. Global Arrays enables applications to adopt the Partitioned Global Address Space (PGAS) programming model and provides primitives for one-sided communication and atomic operations. However, the structure of the CC code that makes the communication calls does not take advantage of these more advanced concepts and rather follows a more traditional Coarse Grain Parallelism (CGP) programming paradigm. Let us consider again the pseudocode shown in [Figure 2.1](#). This figure abstracts away many of the details of the actual code but captures the structure very accurately. Namely, the work is organized in subroutines (that are further organized in logical steps) and inside each subroutine there are multiple nested loops with the call to the most computationally intensive functions (i.e., `DGEMM` and `TCE_SORT_*`) contained in the innermost loop. Interestingly, the calls to the functions that fetch the data to be processed by these calls (i.e., `GET_HASH_BLOCK`) are also in the innermost loop, immediately preceding the computation. In other words, the program contains no additional work that is

[§]NWCHEM uses a mixture of FORTRAN dialects, including 77 and more modern ones.

available between the call to the data transfer function and the call to the computation function that uses the data. This means no matter how sophisticated and efficient the communication library and the network hardware is, this data exchange will not be overlapped with computation. This behavior leads to a major waste of efficiency, since at almost any given point in the program there is additional work (in other subroutines) that is not semantically dependent on the work currently being performed. However, the coarse grain structure of the program—with work and communication contained in deep loop nests inside subroutines—does not allow for different units of work, that are independent to one another, to be utilized for communication-computation overlapping.

This missed opportunity for overlapping can be witnessed in the execution trace shown in [Figure 4.16](#). The figure shows a zoomed-in segment of the execution trace to improve readability. The shown segment is representative of the whole execution and does not exhibit a unique behavior. In this trace, the different colors represent different operations (i.e., **GEMM**, **SORT**, data transfer), the X-axis represents time (i.e., longer boxes signify operations that took longer to finish), and each row corresponds to one MPI rank (i.e., one instance of the parallel application) out of the total 224 used for this run. The red color represents the execution of a matrix multiply (**GEMM**), which is the most computationally expensive operation in this code. The blue and purple colors represent data transfers (matrices A and B needed to perform the $C+ = A * B$ operation that a **GEMM** performs). As can be easily seen by the prominence of the blue color in the trace, the communication imposes an unacceptably high overhead in the execution of the original code.

This behavior is in stark contrast with the dataflow-based execution model that the PARSEC-enabled version of the code follows. In the latter, computation load is organized in tasks (not loops, or subroutines), and tasks declare to the runtime their dependencies to other tasks. When a task completes, the runtime can initiate the asynchronous transfer of its output data while scheduling for execution the next task whose dependencies have been satisfied. This opportunistic execution,

typical of dataflow-based systems, allows for maximum communication computation overlapping. This is true because at any given time during the life of the program, if there is available computation, it will be performed without unnecessary waiting for some predetermined order of loops or subroutines to be satisfied.

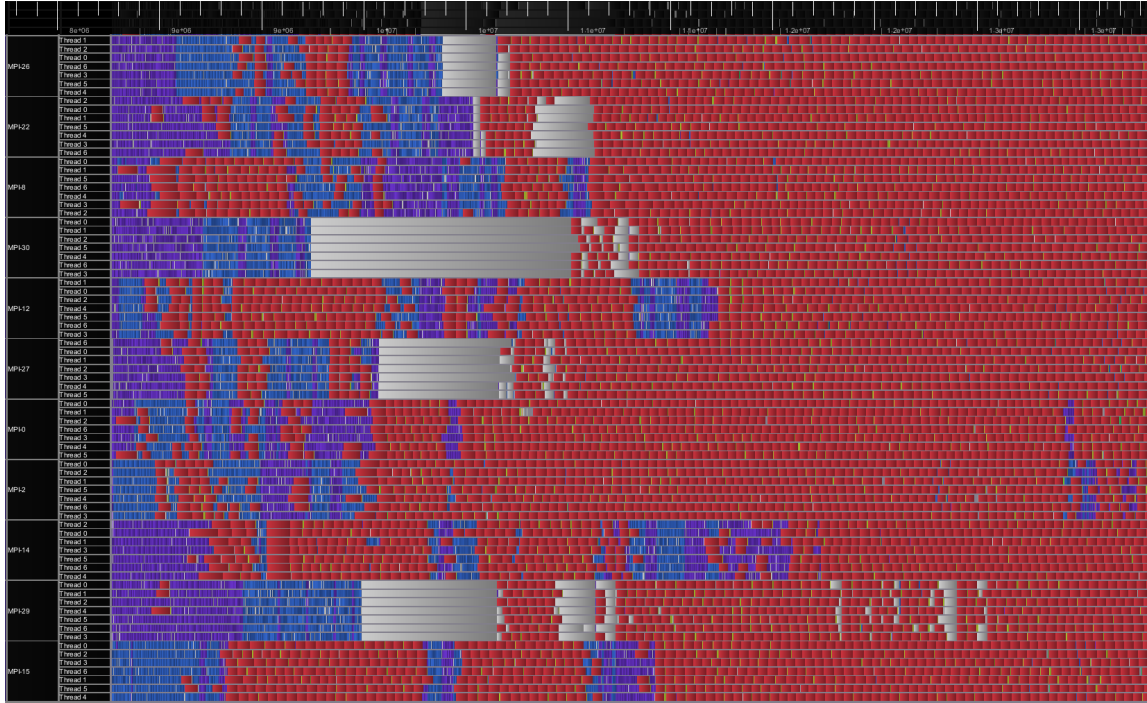


Figure 4.17: Zoomed in on execution trace of dataflow-based NWChem code.

The opportunistic execution of the dataflow-based code is demonstrated in the trace shown in Figure 4.17. As can be seen in this figure—which is a zoomed-in view of the beginning of the execution—many of the tasks, responsible for reading input data (blue and purple boxes) and sending it to the tasks that will perform the computation (red boxes), are scheduled for execution early on. However, as soon as some data transfers complete, computation tasks become ready and, from that point on, communication and computation are overlapped. It is worth noting that, in contrast with the trace of the original code (Figure 4.16), the length of the blue and purple boxes in Figure 4.17 do not represent the communication cost of the data transfers, since tasks in PARSEC do not perform explicit communication.

Rather, these boxes represent the tasks that merely express their communication needs to the runtime by specifying their dependencies to other tasks. The actual data transfers are scheduled and managed by the runtime and are fully overlapped with the computation. As a result, they do not appear in the trace, but they are responsible for the light gray gaps ([Figure 4.17](#)), which are periods of time where no work can be executed because the corresponding data has not been transferred yet (a phenomenon that only happens at the beginning of the execution). To summarize, by comparing the two traces we can clearly see that the original NWChem code faces significant communication delays which can be largely addressed through the use of communication-computation overlapping.

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

Work that in the original code resides in completely different subroutines and can not be utilized for overlapping, unless a major restructuring of the code is performed.

4.3.2 Freedom from control-flow

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

4.3.3 Multi-threading and accelerators

Another artifact of the way the original code is structured is that taking advantage of multi-threading to utilize multiple cores on each node is not easy to implement efficiently. As a result, the NWChem application does not use threads and rather relies on multiple MPI ranks per node in order to utilize multiple cores. In PARSEC, multi-threading comes at no additional cost for the developer. Once the developer has defined the tasks and the dataflow between them, PARSEC will automatically use threads to accomplish the work, utilizing multiple hardware resources. Furthermore, PARSEC uses a combination of thread local queues and global queues to store available tasks during the execution. When a task T_i completes, the tasks T_j that

were waiting for the output of T_i will be placed in the thread local queue of the thread that executed T_i . As a result, work that should naturally execute as a chain (because the output of one task is the input of another) has a high probability of executing as a chain in PARSEC taking advantage of cache locality and increasing execution efficiency. At the same time, the existence of global queues and work stealing guarantees that PARSEC will also exhibit good load balance.

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

4.4 Performance Evaluation for entire CC

This section presents the performance of the entire CC code using the PTG version “Dataflow:T2” of the 29 CC subroutines and contrasts it with the performance of the original code “Orig:T2”. Note, the unique monikers “Orig:T2” and “Dataflow:T2” are introduced in [Section 2.3.3](#), and refer to the set of CC subroutines targeted in this research.

[Figure 4.18](#) depicts a high level view of the integration of the PARSEC-enabled code in NWChem’s CC component. The code region timed (see **start** and **end** timers in [Figure 4.18](#)) includes all 19 T1 and 41 T2 subroutines as well as additional execution steps that set up the iterative CC computation. The only difference between the original NWChem runs and the modified version is the replacement of the 29 original T2 subroutines “Orig:T2” with their dataflow version “Dataflow:T2” and the prerequisites discussed earlier. In a nutshell, these prerequisites include: meta-data

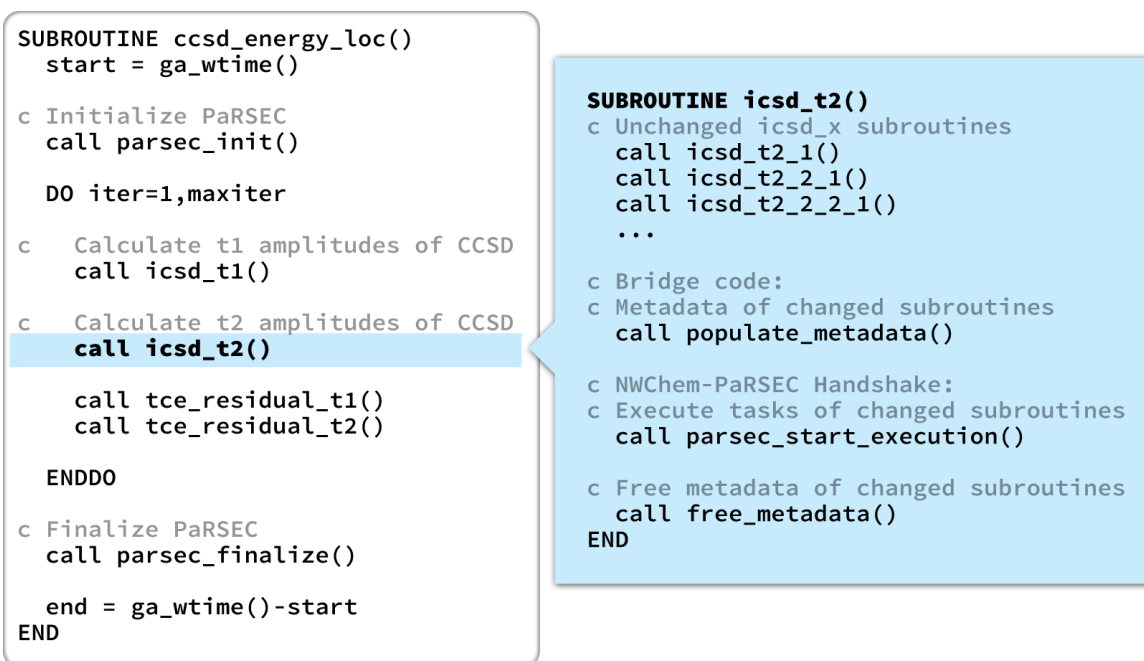


Figure 4.18: High level view of PARSEC code in NWChem.

vector population, initialization, and finalization of PARSEC. Also, the experiments allow for all iterations of the iterative CC code to reach completion.

4.4.1 Methodology

The input data is the beta-carotene molecule ($C_{40}H_{56}$) in the 6-31G basis set, composed of 472 basis set functions. In these tests, all core electrons are kept frozen, and 296 electrons are correlated. Figure 4.19 shows the relative workload of different subroutines (omitting those that fell under 0.1%). To calculate this load, the sum is computed of the number of floating point operations of each GEMM that a subroutine performs (given the sizes of the input matrices).

Table 4.2 lists additional statistics for the 10 computationally most expensive CC subroutines, such as the size and shape of individual GEMM operations, the amount of floating-point operations (FLOPs) per GEMM, as well as the total number of such GEMM operations. This data lays out that, for the beta-carotene molecule in particular, the

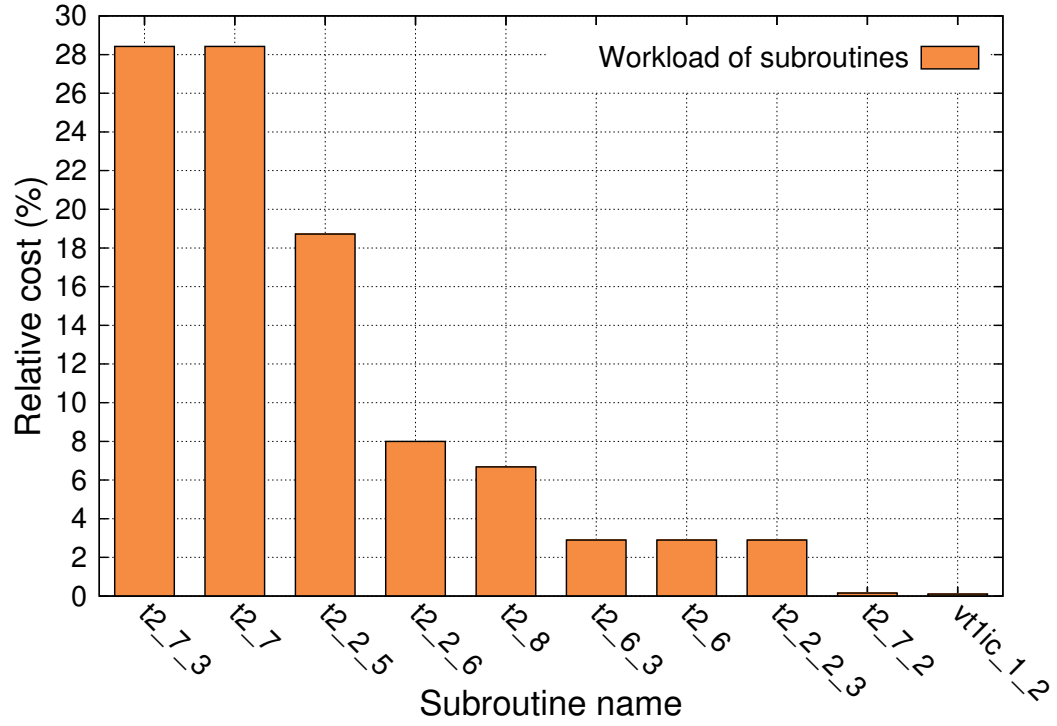


Figure 4.19: Relative load of top 10 heaviest CC subroutines.

Table 4.2: GEMM size and shape, and total number of GEMMs for the top 10 heaviest CC subroutines when running beta-carotene with tilesize=45.

CC subroutines	M	N	K	GEMM FLOP count	Total # of GEMMs
t2_7_3	1107	1107	1107	2,713,144,086	327,680
t2_7	1107	1107	1107	2,713,144,086	327,680
t2_2_5	1107	729	1107	1,786,704,642	327,680
t2_2_6	729	1107	1681	2,713,144,086	92,160
t2_8	729	1681	1681	4,119,959,538	50,688
t2_6_3	729	729	1681	1,786,704,642	50,688
t2_6	1681	729	729	1,786,704,642	50,688
t2_2_2_3	729	729	1681	1,786,704,642	50,688
t2_7_2	27	45,387	41	100,486,818	49,152
vt1ic_1_2	27	29,889	41	66,174,246	49,152

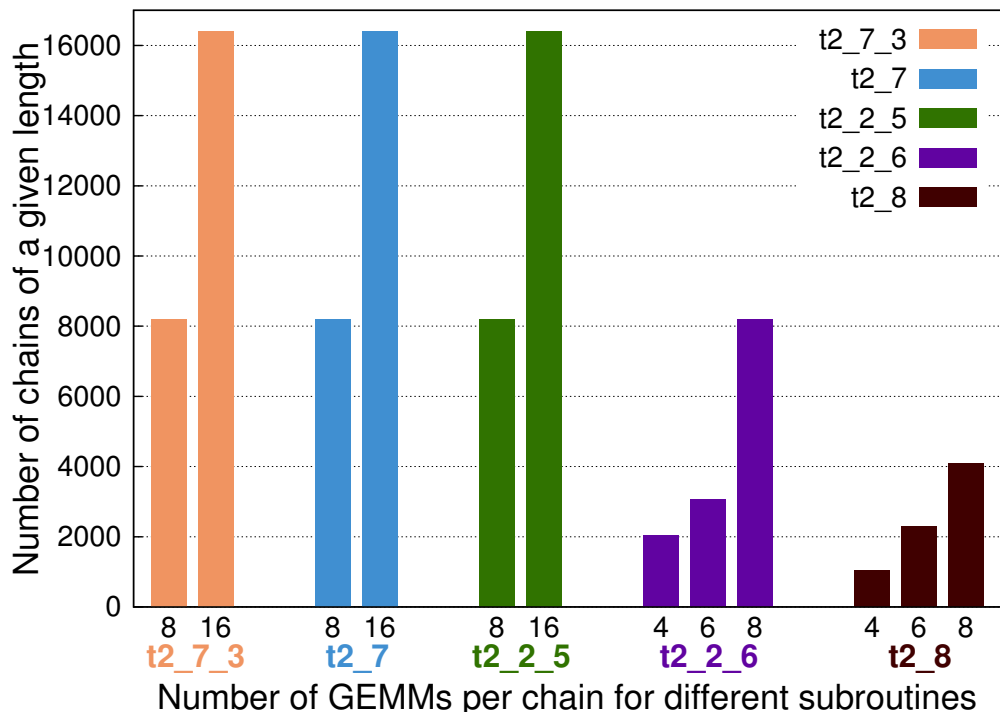


Figure 4.20: Distribution of Chain length for top five subroutines.

resulting matrix size and shape ranges from squared ($M = N$) to roughly squared ($M \approx N$) all the way to short-fat ($M \ll N$). While the relative workload among the various subroutines—as shown in Figure 4.19—will be similar for different input molecules and basis sets, the exact GEMM statistics as shown in Table 4.2, however, will differ.

Additionally, Figure 4.20 shows the distribution of chain lengths for the five subroutines with the highest workload in the case of beta-carotene. The different colors in this figure are for readability only. As can be seen from these statistics, the subroutines targeted for the dataflow conversion effort comprise approx. 91% of the execution time of all 41 T2 subroutines in the original NWChem TCE CC execution.

The performance tests for the original TCE-generated code and the dataflow version of Dataflow:T2 were performed on the Cascade computer system at EMSL/PNNL. Each node has 128 GB of main memory and is a dual-socket Intel Xeon E5-2670 (Sandy Bridge EP) system with a total of 16 cores running at 2.6 GHz. A

number of different scalability tests are performed, utilizing 1, 2, 4, 8, and 16 cores per node. NWChem v6.6 was compiled with the Intel 14.0.0 compiler, using the optimized BLAS library MKL 11.0, provided on Cascade.

4.4.2 Discussion

Figure 4.21, and Figure 4.22 show the execution time of the entire CC kernel when the implementation found in the original NWChem code is used, and when the PARSEC-based dataflow implementation is used for the (earlier mentioned) 29 `Dataflow:T2` subroutines. Each of the experiments were run three times; the variance between the runs, however, is so small that it is not visible in the figures. Also, the correctness of the final computed energies have been verified for each run, and differences occur only in the last digit or two (meaning, the energies match for up to the 14th decimal place). The graph depicts the behavior of the original code using the dark (green) dashed line and the behavior of the PARSEC implementation using the light (orange) solid lines. Once again, the execution time of the PARSEC runs does not exclude any steps performed by the modified code.

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

The results are similar on a 64 node partition: the PARSEC version of CC is fastest (for 16 cores/node) with a 43% run time improvement compared to the original code (which on 64 nodes performs best for 4 cores/node). It is also interesting to point out that for 64 nodes, while PARSEC manages to use an increasing number of cores—all the way up to $64 \times 16 = 1024$ cores—to improve performance, the original code

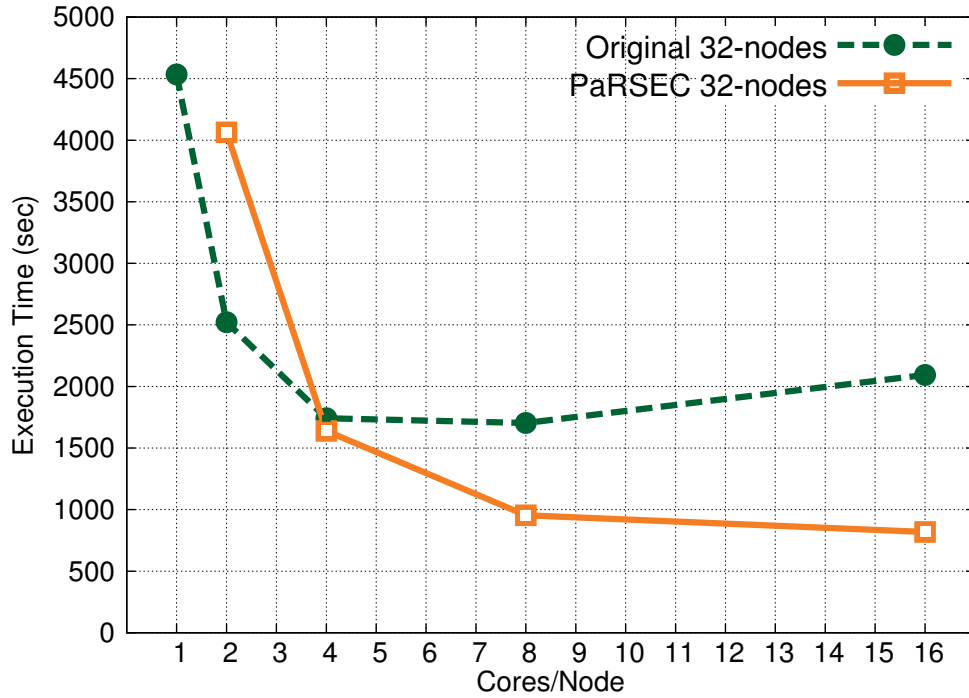


Figure 4.21: CC Execution time comparison using 32 nodes on EMSL/PNNL Cascade (for beta-carotene with tilesize=45).

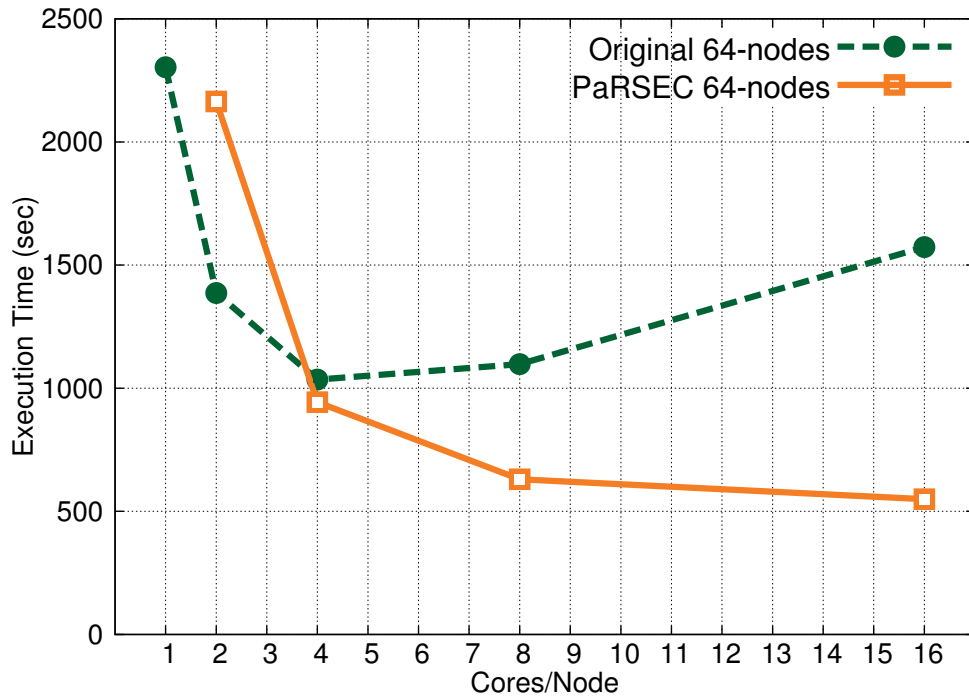


Figure 4.22: CC Execution time comparison using 64 nodes on EMSL/PNNL Cascade (for beta-carotene with tilesize=45).

exhibits a slowdown beyond 4 cores/node. This behavior is not surprising since (1) the unit of parallelism of the original code (chain of `GEMMs`) is much coarser than that of PARSEC (single `GEMM`), and (2) the original code uses a global atomic variable for load balancing while PARSEC distributes the work in a round robin fashion and avoids any kind of global agreement in the critical path.

4.5 Summary

This chapter presents the effort to utilize the explicit dataflow model (PTG) through the task-based execution system PARSEC to optimize part of NWChem, a legacy Quantum Chemistry application written using FORTRAN 77 and Global Arrays. A discussion on how PARSEC is used from within NWChem, followed by a detailed analysis of the different algorithmic choices of the algorithm as well as their impact on performance is given. The lessons learned from this analysis guide us in the effort to port a larger part of the application to run over PARSEC, but can also provide useful insight to other groups aiming to modernize legacy applications by converting them to a task-based form.

This work successfully demonstrates the feasibility of converting TCE-generated CC code into a form that can execute in a dataflow-based task scheduling environment, such as PARSEC. Finally, this effort substantiates that *utilizing dataflow-based execution for CC enables more efficient and scalable computation*—as our performance evaluation reveals a performance boost of more than 2x for the entire CC kernel—compared to the employment of the antiquated CGP programming model that, to this day, remains the prevalent model for developing scientific applications.

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 can be 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, GPU, Xeon Phi processors) that perform best for the type of task under consideration.

Chapter 5

Implicit Dataflow Execution of Coupled Cluster Methods

This chapter 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, as covered in [Chapter 4](#), are only supported by the PARSEC runtime, the *implicit dataflow* model can be used by any runtime a user chooses to use. That is, any runtime that supports the serial task-insertion interface can be used to submit tasks, and generate the DAG of the execution.

In the context of the present work, this means that, computational chemists are not forced to use the PARSEC runtime. Instead, this effort offers flexibility in terms of what dataflow paradigm to use—explicit or implicit—and it also offers users the choice between runtimes.

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 SD code and the PTG-enabled PARSEC implementation.

5.1 Design Decisions

The development of an *implicit dataflow* version of CC is less “painful” on the application developer, in the sense that it is easier 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 run-time, derives the data dependencies between tasks, builds the DAG of the tasks in memory, and executes the graph of tasks. Such programming model, 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 very different representation of the algorithm, like it is, for example, the case for the PTG representation.

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—and express additional parallelism by simply submitting tasks in the sequential flow of the program. This is not possible for explicit dataflow. For instance, the PTG-enabled version requires an initial “Discovery Phase” because the original CC code is not organized in pure tasks—i.e., functions with no side-effects to any memory other than arguments passed to the function itself—nor is the control flow affine (e.g. loop execution space has holes in it; branches are statically undecidable since their outcome depends on program data, and thus it cannot be resolved at compile time) [37]. However, even though 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 [37]. Therefore, it is technically feasible to express the CC code as a PTG, that is, by using lookups into the data of the program, either directly or indirectly. The PTG-enabled CC version, described in [Chapter 4](#), accesses the program data indirectly by building meta-data structures

in a preliminary discovery step. The technical details of this discovery phase are explained in [Section 4.1](#).

5.1.1 Insert_Task() Version

The implicit dataflow version of CC that is based on the serial task-insertion 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 5.1](#) shows the code snippet of one original CC subroutine (left), and compares it against the modified version (right).

<pre> 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) </pre>	<pre> my_next_task = SharedCounter() DO i DO j ! Validation of loop index values IF ... DFILL(dimc, dbl_mb(k_c),...) DO k DO l ! Validation of loop index values IF ... GET_HASH_BLOCK(GA,A) GET_HASH_BLOCK(GA,B) </pre>
<pre> 13 call DGEMM('T','N',..., 1.0d0 , ↪ dbl_mb(k_c),...) </pre>	<pre> call INSERT_TASK_CORE(..., 0.0d0 , ↪ k_c,...) </pre>
<pre> 14 END IF 15 END DO 16 END DO </pre>	<pre> END IF END DO END DO </pre>
<pre> 17 18 ADD_HASH_BLOCK(dbl_mb(k_c),...) 19 my_next_task = SharedCounter() 20 END IF 21 END DO 22 END DO </pre>	<pre> call INSERT_TASK_WAIT() ADD_HASH_BLOCK(dbl_mb(k_c),...) my_next_task = SharedCounter() END IF END DO END DO </pre>

Figure 5.1: Original CC subroutine code (left), compared to the `Insert_Task()` 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 5.2) that has as input values the `GEMM` parameters. Note, the parameter change for the scalar *beta* (highlighted in orange in 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 `GEMMs` 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 5.2.

```

1 recursive SUBROUTINE insert_task_core(...)
2     ...
3     double precision, dimension(:), pointer :: c_tmp
4     allocate(c_tmp(dimc))
5     call DGEMM('T','N',..., c_tmp, ...)

6     call lock_mutex()
7     DO i = 0, dimc-1
8         dbl_mb(k_c+i) = dbl_mb(k_c+i) + c_tmp(1+i)
9     END DO
10    call unlock_mutex()

11    deallocate( c_tmp )
12    RETURN
13 END

```

Figure 5.2: `Insert_task_core()` subroutine that is submitted to the runtime.

Once the `GEMM` operation is completed, each task adds its computed `c_tmp` array to the `dbl_mb(k_c)` array (lines 6-10) that belongs to the main thread. This addition 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 `dlb_mb(k_c)` array (which is highlighted in green in the code snippets) is created and initialized by the main thread at the beginning of each chain of **GEMMs** (line 6 in Figure 5.1). 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 5.1). Even though the runtime can schedule and execute all **GEMMs** in parallel, because of the `ADD_HASH_BLOCK()` operation (in line 18 in Figure 5.1), all **GEMM** tasks need to be completed so that the final `dlb_mb(k_c)` array can be added to the Global Array.

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 (line 11-12 in Figure 5.1), 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 Global Arrays 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 Global Arrays is currently not thread-safe. We have been in contact with the Global Arrays developers and they are just now starting to work on multi-threading support for the GA library.

5.1.2 Design Variations for OpenMP Version

The OPENMP specification [17] is a well known standard for developing parallel shared-memory applications. The OPENMP version 3.0 [3] introduced a feature called *tasking*, which adds some support for task level parallelism. The main difficulty for users, however, is to ensure that there are no data dependencies between **PARALLEL** regions. That makes it difficult to express a complex DAG of tasks and their data dependencies to one another using OPENMP.

Three different design variations for the OPENMP task version of the CC code have been implemented and compared. Before describing the differences of these alternative versions, the code blocks that they all have in common will be 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 when a task runs is taken by the OPENMP runtime. A task is composed of three parts: The code to be executed, the data environment associated with the task, and 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, e.g., **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 creation time of a task. 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 5.3 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 the **!OMP TASK** and **!OMP END TASK**, which specifies a single **GEMM** operation as one task (lines 15-24 in Figure 5.3).

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

<pre> 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) </pre>	<pre> my_next_task = SharedCounter() DO i DO j ! Validation of loop index values IF ... DFILL(dimc, dbl_mb(k_c), ...) !OMP PARALLEL PRIVATE(...) !OMP SINGLE DO k DO l ! Validation of loop index values IF ... GET_HASH_BLOCK(GA,A) GET_HASH_BLOCK(GA,B) </pre>
<pre> 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 deallocate(c_tmp) 23 !OMP END TASK 24 </pre>	<pre> !OMP TASK PRIVATE(c_tmp,h) allocate(c_tmp(dimc)) call DGEMM('T','N',..., 0.0d0, c_tmp, ...) !OMP CRITICAL DO h=0,dimc-1 dbl_mb(k_c+h)=dbl_mb(k_c+h)+c_tmp(h+1) END DO !OMP END CRITICAL deallocate(c_tmp) !OMP END TASK </pre>
<pre> 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 </pre>	<pre> END IF END DO END DO !OMP END SINGLE !OMP END PARALLEL ADD_HASH_BLOCK(dbl_mb(k_c), ...) my_next_task = SharedCounter() END IF END DO END DO </pre>

Figure 5.3: Pseudocode of OPENMP atomic (**left**); and critical (**right**) versions.

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 which waits for all threads to be completed before moving on. This implicit barrier—which is equivalent to the `INSERT_TASK_WAIT()` calls in Figure 5.1—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 `dlb_mb(k_c)` array can be added to the Global Array via the `ADD_HASH_BLOCK()` operation (line 30 in Figure 5.3).

The three different design variations that have been implemented and studied for the OPENMP task version of the CC code are shown in Figure 5.3 and Figure 5.4. The difference between these alternative versions is in the way the C arrays from the GEMM computations are added to the `dlb_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 `dlb_mb(k_c)` array by multiple threads. This can be accomplished in different ways with the same effect on `dlb_mb(k_c)` but different results on the performance.

Atomic: The version in Figure 5.3 (left) uses an OPENMP `ATOMIC` operation (lines 19-20), which protects only the single assignment that immediately follows it. An atomic provides mutual exclusion but only applies to the load/update of a memory location. More precisely, atomic only protects the update of `dlb_mb(k_c)`. This is a lightweight, special form of a critical region, which suggests a lower overhead for atomic operations compare to critical regions.

Critical: The version in Figure 5.3 (right) uses an OPENMP `CRITICAL` region, which surrounds the entire DO loop that adds the `c_tmps` array to `dlb_mb(k_c)`

```

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 5.4: Pseudocode of buffered OPENMP Task implementation.

(lines 18-22). In OPENMP, all the unnamed critical regions are mutually exclusive. Because a critical region can surround any arbitrary block of code, this generality “typically” comes with a higher price tag, by incurring a significant overhead every time a thread enters and exits the critical section. Therefore, it is surprising that the critical region implementation in CC results in the best performance (see [Section 5.2](#)).

Buffered: The version in [Figure 5.4](#) creates a temporary C array for each OPENMP thread. This means, there are “`number_of_omp_threads`” temporary C arrays, and all are 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 `dlb_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 due to the use of thread-id and critical regions in the current tasks.

Even though tasks are tied to threads in the aforementioned implementations, threads can be bound to CPU cores for performance reasons. Forcing threads to run on a specific processor core (or subset of processors) 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 5.1](#) summarizes its different settings.

All three implementations have been run and compared against each of the four OMP_PROC_BIND options.

Table 5.1: Thread binding options with OMP_PROC_BIND environment variable.

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 behavior).
CLOSE	Binds threads as close to the main thread as possible.
SPREAD	Binds threads as evenly distributed as possible among the processing cores.

5.2 Performance Evaluation for entire 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 contrast it with the performance of (a) the original code, and (b) the “explicit” dataflow version that has been covered in [Chapter 4](#).

5.2.1 Methodology

The input data is the beta-carotene molecule ($C_{40}H_{56}$) in the 6-31G basis set, composed of 472 basis set functions. In these tests, all core electrons are kept frozen, and 296 electrons are correlated.

The subroutines targeted for the “implicit” dataflow effort are the same 29 T2 subroutines—in this text referred to as `Orig:T2` and `Dataflow:T2`—that comprise approx. 91% of the execution time of all 41 T2 subroutines. Note, the unique monikers “Orig:T2” and “Dataflow:T2” are introduced in [Section 2.3.3](#), and details about the specific set of CC subroutines are covered there. In a nutshell, this is the set of subroutines that has been identified in the statistics in [Figure 4.19](#), [Figure 4.20](#)

and [Table 4.2](#) as the computationally most expensive work when computing the CC correlation energy.

The performance tests were performed on the Cascade computer system at EMSL/PNNL. Each node has 128 GB of main memory and is a dual-socket Intel Xeon E5-2670 (Sandy Bridge EP) system with a total of 16 cores running at 2.6 GHz. Various scalability tests have been performed, utilizing 1, 2, 4, 8, and 16 cores per node. At the time of the execution of these experiments, a newer version of the Intel compiler and MKL library were available on cascade compared to the tests described in [Section 4.4.1](#). NWChem v6.6 was compiled with the Intel 15.0.0 compiler, using the optimized BLAS library MKL 14.0, provided on Cascade.

5.2.2 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 is used for the (earlier mentioned) 29 `Dataflow:T2` subroutines. Each of the experiments were run three times; the variance between the runs, however, is so small that it is not visible in the figures. Also, the correctness of the final computed energies have been verified for each run, and differences occur only in the last digit or two (meaning, the energies match for up to the 14th decimal place). The graphs depict the behavior of the original code using the back line and the behavior of the dataflow implementations using bars. Once again, the execution time of the dataflow runs does not exclude any steps performed by the modified code.

OpenMP Tasks

All three CC implementations with OPENMP Tasks have been run and compared against each of the four `OMP_PROC_BIND` options. The results are shown in [Figure 5.5](#), [Figure 5.6](#), [Figure 5.7](#) for the “atomic”, “critical”, and “buffered” version,

respectively. For all three cases, the CC execution for runs on 32 nodes performs best when the OPENMP threads are bound to processing cores and as evenly distributed as possible among the cores (`OMP_PROC_BIND = SPREAD`).

Consecutively, for the analysis of the three different versions, shown in [Figure 5.8](#), the performance of the three OPENMP versions with `OMP_PROC_BIND = SPREAD` is compared.

The implementation that uses a `CRITICAL` region (to protect the addition of `c_tmp` to the `dbl_mb(k_c)`) performs best over the other two versions. Intuitively, one would expect a `CRITICAL` region to be outperformed by an `ATOMIC` operation. However, it clearly depends on how the compiler/OPENMP-runtime implements the `ATOMIC` directive. If it is able to generate atomic assembly instructions for the update, then one might expect the atomic version to be faster, as it will allow different elements of the array to be updated concurrently by different threads. On the other hand, if the compiler is not able to generate these instructions, then it will most likely fall back to using a mutex lock to protect the updates, in which case it will be very similar to having a critical region inside the loop. This, however, will have more overhead than the “critical” version because the lock has to be acquired and released many times (from inside the `DO` loop). Instead, for the “critical” implementation, a lock needs to be acquired and released only once because the critical is outside the `DO` loop.

In the third version, referred to as “buffered”, each OPENMP task accumulates results into a per-thread copy of (part of) `dbl_mb`, and once all tasks are completed, the main thread accumulates these per-thread results together. 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 [Table 4.2](#) 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 reduction of the per-thread copies. This may not vectorize well due to the non-unit stride on `c_tmps`, depending on how smart the

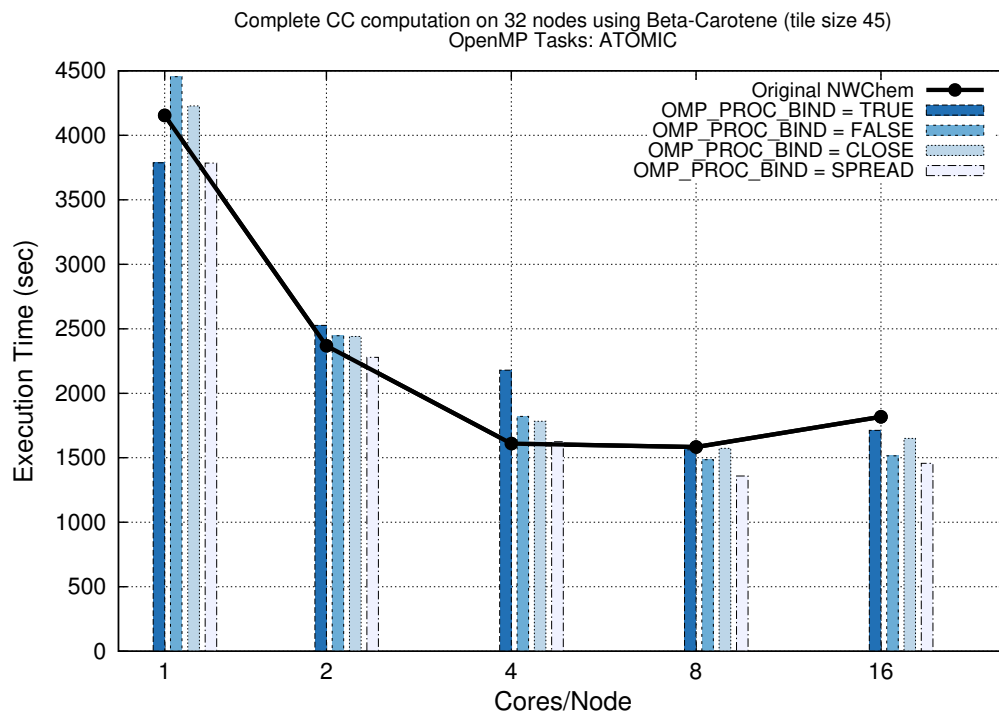


Figure 5.5: Comparison of CC execution time with OpenMP Tasks with “ATOMIC summation of C” using different OMP_PROC_BIND.

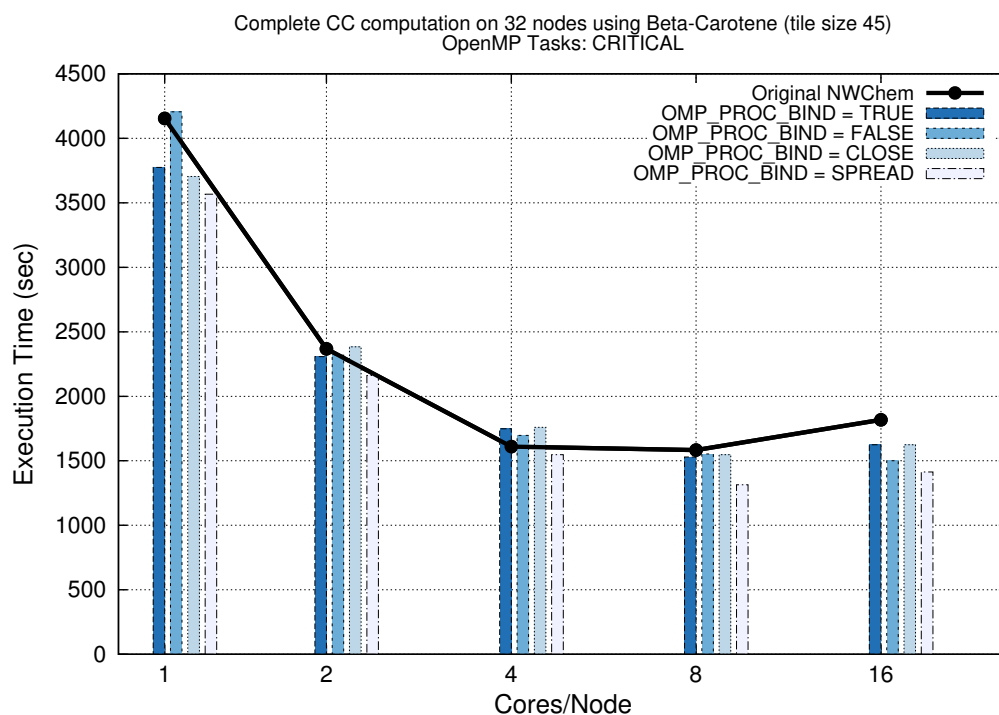


Figure 5.6: Comparison of CC execution time with OpenMP Tasks with “CRITICAL summation of C” using different OMP_PROC_BIND.

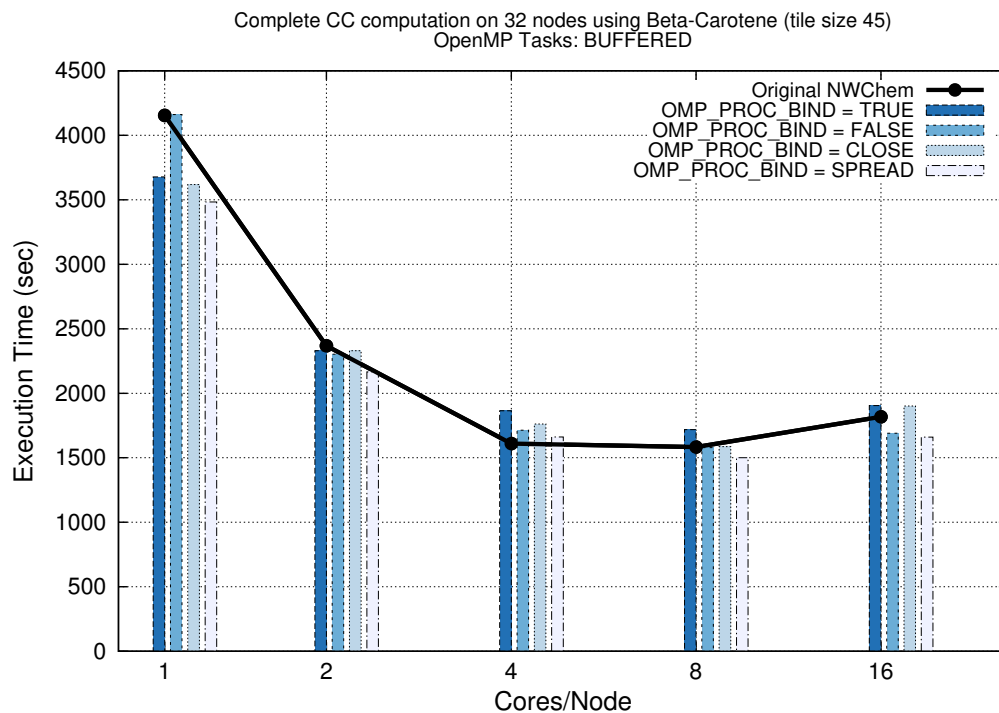


Figure 5.7: Comparison of CC execution time with OpenMP Tasks with “Buffered summation of C” using different OMP_PROC_BIND.

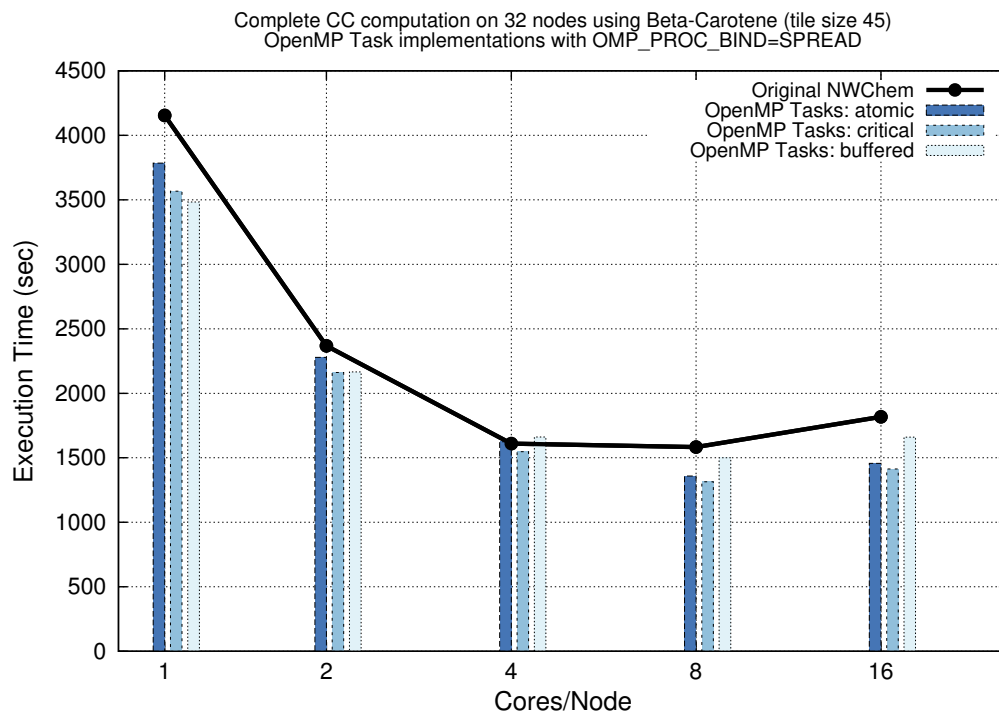


Figure 5.8: Comparison of CC execution time of all three OpenMP implementations using OMP_PROC_BIND=SPREAD.

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.

Insert_Task()

For the `Insert_Task()` version of the implicit dataflow implementation of NWChem CC, 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 investigations in this work. However, because both runtimes use a different scheduler “by default”, the performance of three schedulers has been evaluated, and the scheduler delivering the best performance has been chosen for subsequent performance tests. [Figure 5.9](#) shows the differences in CC execution time for runs on 32 nodes when STARPU is configured with different schedulers:

lws: The Locality Work Stealing (lws) scheduler uses a queue per worker, and schedules a task on the worker which 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 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.

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 executing at that time. The [Figure 4.20](#) confirms that 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

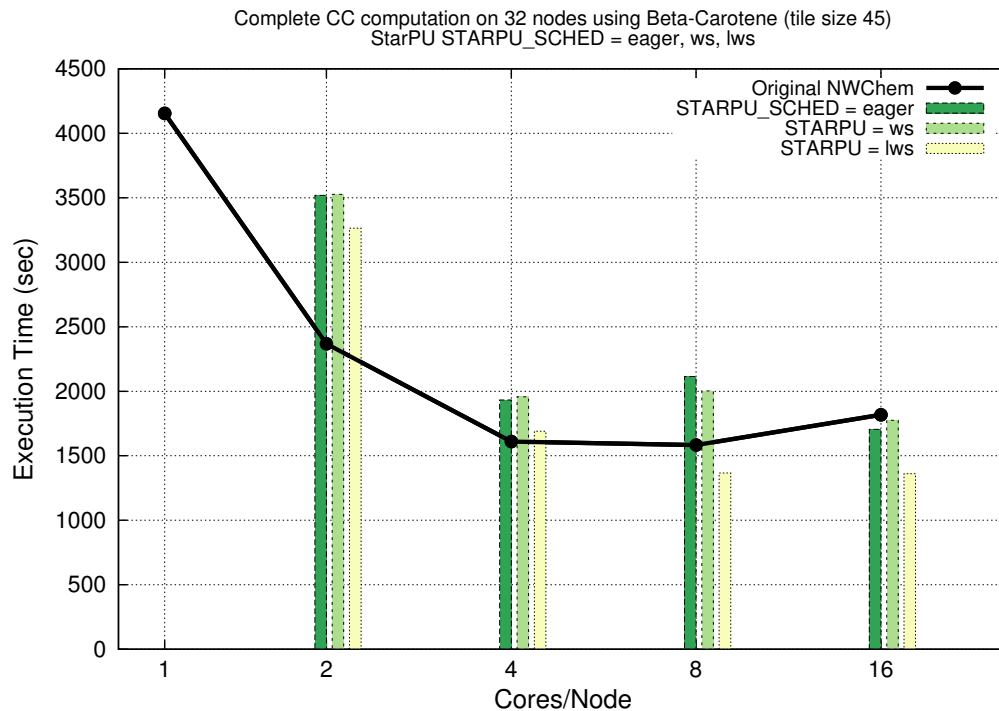


Figure 5.9: Comparison of CC run-time with StarPU using different schedulers.

queue that requires a lock to be acquired and released each time, causing an overhead that is avoided with the other schedulers, which do not use a global queue.

Figure 5.9 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.

Results

Performance studies have been performed on 32, 64 and 128 nodes, and results are shown in Figure 5.10, Figure 5.11, Figure 5.12, respectively. Each of these graphs depicts the results of the three best-performing implicit dataflow versions, the explicit dataflow version, and the original CC code. Both dataflow versions outperform the original code on all three node partitions. While the speedup of the explicit over original decreases, the speedup of implicit over original increases as the number of nodes grows from 32 to 128. Table 5.2 lists the exact speedups for each node partition.

Table 5.2: Speedup for dataflow versions over original CC.

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

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 performs always best for 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:

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. The statistics in [Figure 4.20](#) show that 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 ≈ 11 .
2. The implicit code inherits the limitations from the original code that 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 Global Arrays layer. This communication is blocking and leaves no room for communication and computation to overlap within the same thread of the execution. The explicit dataflow model uses non-blocking communication and allows for communication-computation overlap.

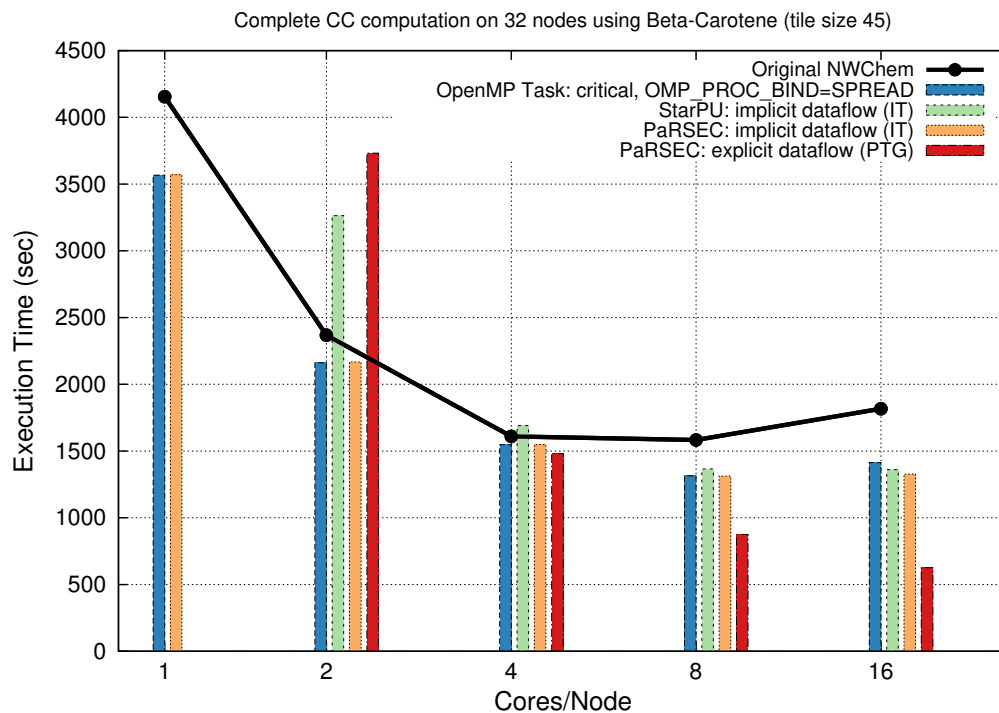


Figure 5.10: CC Execution time comparison using 32 nodes (using up to 512 cores) on EMSL/PNNL Cascade (for beta-carotene with tilesize=45).

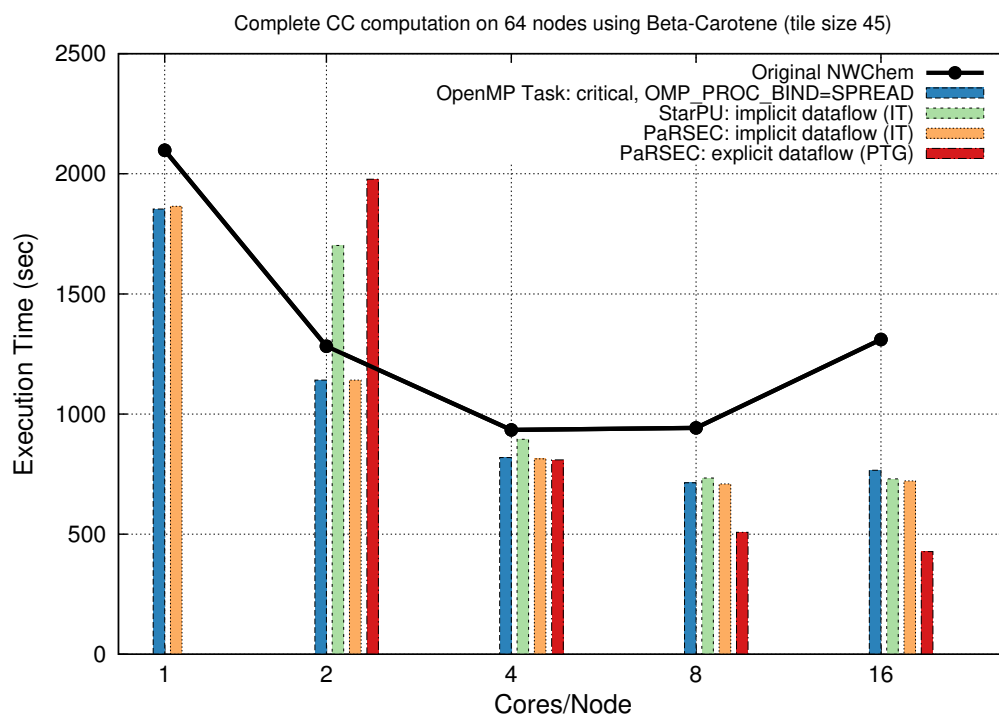


Figure 5.11: CC Execution time comparison using 64 nodes (using up to 1,024 cores) on EMSL/PNNL Cascade (for beta-carotene with tilesize=45).

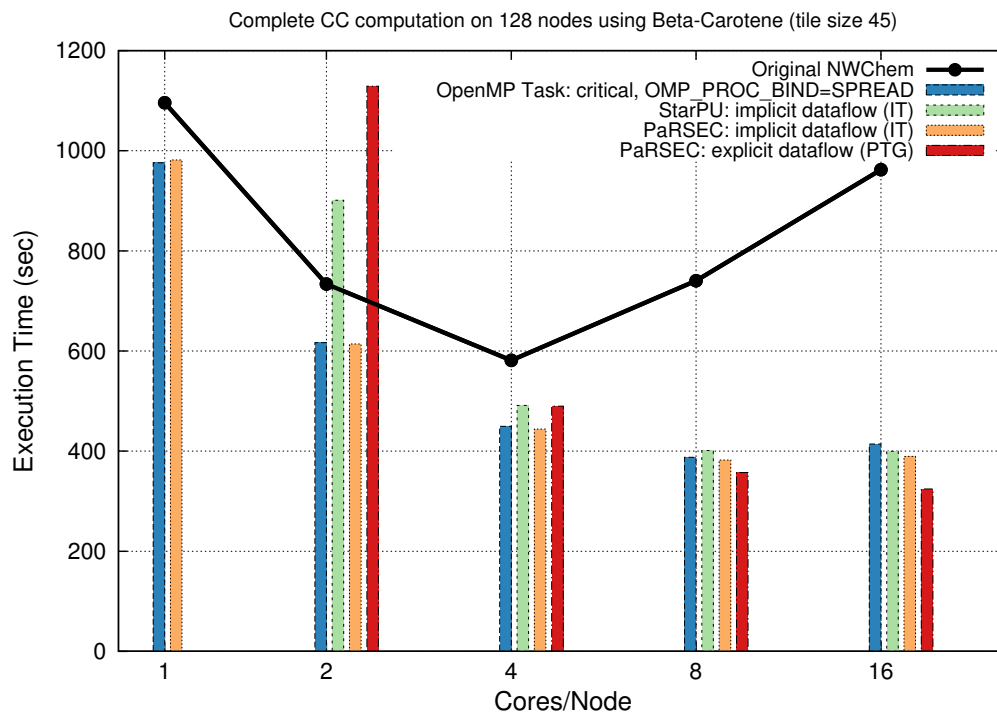


Figure 5.12: CC Execution time comparison using 128 nodes (using up to 2,048 cores) on EMSL/PNNL Cascade (for beta-carotene with tilesize=45).

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.

Chapter 6

An Abstract Model for Dataflow Comparison

Dataflow programming paradigms are an alternative approach with significant potential to the Coarse Grain Parallelism programming model. This chapter compares “explicit” dataflow—and its Parameterized Task Graph (PTG) abstraction as it is used in the dataflow task-based runtime system, PARSEC—with “implicit” dataflow models of runtimes that support the serial task-insertion interface. It presents an abstract model that quantifies the differences between “explicit” and “implicit” dataflow models.

6.1 Affine Programs

Suppose there is a program that is affine. For instance, dense linear algebra results in largely affine implementations. Such affine codes have the following restrictions:

- Loops that have fixed bounds and step that may not change during execution (but could change from one execution to the next). If, for instance, the bounds or steps are changed in the loop body, then a static analysis is not possible at compile time.

- Array indices with expressions that are limited to addition and subtraction of the loop induction variables, constants, numeric literals (as well as multiplication by numeric literals) and branches (if-then-else) that contain only similar arithmetic expressions and comparison operators.

Such an affine program can be fully analyzed using polyhedral analysis, and the explicit dataflow representation can be automatically generated as a parameterized task graph (PTG). The PTG is the compact representation of a DAG where 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 requirements to store a parameterized task graph (PTG) does not grow as the number of tasks grows but remains constant: 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 a node p ,
- $T_{compute}^{t_i}$ is the execution time it takes to compute a task t_i ,
- $T_{comm}^{t_i}$ is the communication time it takes to transfer data from another node that is required for a 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.

Note, that $0 \leq T_{overlap}^{t_i} \leq T_{comm}^{t_i}$ and that $T_{overlap}^{t_i} = T_{comm}^{t_i}$ when the data is on the same node or when communication is completely overlapped with computation.

Furthermore, $T_{wait}^{t_i} = T_{comm}^{t_i} - T_{overlap}^{t_i}$ corresponds to the time a task waits for data to arrive.

Since the communication is performed by a node p , the above equations can be simplified and provided by “node” rather than by “tasks assigned to a node”. Thus, one can denote $T_{compute}^p = \sum_{t=0}^{N_p} T_{compute}^{t_i}$ and $T_{wait}^p = \sum_{t=0}^{N_p} T_{wait}^{t_i}$ then the time required to complete all tasks on a node derives as:

$$T_p = T_{compute}^p + T_{wait}^p \quad (6.1)$$

$T_{wait}^p = 0$ if there is a complete overlap between communication and computation, and where $T_{wait}^p = T_{comm}^p$ if there is no overlap at all and the data transfer between nodes happens in a sequential fashion with the computation. Also, in the case where the time for data transfer is larger than the time for computation (e.g., for very fine grained tasks or for communication bound algorithms) T_{wait}^p can be expressed as the remaining portion of time that has not been overlapped.

For simplicity, it will be denoted that all tasks are similar and require similar amount of time, say T_w , to be computed. Also, we can assumed that the workload is equally balanced over the P nodes. Furthermore, the DAG exhibits a lot of parallelism with low dependencies on communicated data; and since communication is non-blocking, it can be perfectly overlapped with computation. As for communication, let B be the total data in bytes that is transferred at the maximum throughput of the network, which has a peak bandwidth of BW_{peak} bytes per second.

In terms of time requirements for the explicit dataflow, [Equation 6.1](#) can then be rewritten as:

$$T_p^{expl} = \max \left(\frac{T_w \cdot N}{P \cdot C}, \frac{B}{P \cdot BW_{peak}} \right) \quad (6.2)$$

where N defines the total number of tasks in a DAG, P is the number of nodes, and C is the number of cores per node. Since communication and computation are fully overlapped, T_p^{expl} will be the maximum of the two.

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. In order 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 run-time. This “discovery of the DAG” adds an overhead $T_O^{(1)}$ of $O(N)$ for implicit dataflow models, which is not the case for the explicit dataflow version.

For the explicit dataflow, DAGs are represented in a compact, problem-size independent, format (a PTG) that is evaluated on demand to discover data dependencies. The PTG allows for the evaluation of any part of the DAG without having to store the DAG. For the implicit dataflow, the space requirements to store the DAG grows as the number of tasks grow, resulting in a memory requirement of $O(N)$.

Once the DAG is discovered for implicit dataflow, the sole time requirements for the completion of all tasks in a DAG is identical to [Equation 6.2](#). Note, for both models, the communication is non-blocking and performs at the maximum throughput of the network. In terms of total time per node for the implicit dataflow, this results in the following:

$$T_p^{impl} = \max \left(\frac{T_w \cdot N}{P \cdot C}, \frac{B}{P \cdot BW_{peak}} \right) + T_O^{(1)} \quad (6.3)$$

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 an affine code will execute with a speedup of $O(P \cdot C)$:

$$Speedup = \frac{T_p^{impl}}{T_p^{expl}} = \frac{\frac{T_w \cdot N}{P \cdot C} + T_O^{(1)}}{\frac{T_w \cdot N}{P \cdot C}} = 1 + \frac{T_O^{(1)}}{T_w \cdot N} \cdot P \cdot C \quad (6.4)$$

This demonstrates that the implicit dataflow model—even if sufficient memory is available to store the entire DAG—will have a constant overhead due to the sequential task-insertion at the beginning of the execution. In contrast, the explicit dataflow model always discovers the same amount of parallelism but without the overhead of traversing the entire execution space in order to build the DAG, and without the heavy memory requirements [20].

6.2 Non-affine Programs

On the other end of the spectrum waits the reality that affine codes are a very 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 affine 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.

Nevertheless, it is still possible to generate PTGs for highly dynamic applications, like NWChem, that are not algebraic, and that have meta-data the execution depends on. In such cases, the program data needs to be inspected and stored in meta-data structures in a preliminary step before a PTG can be generated. This program data can then be accessed in the PTG. Unlike a fully algebraic PTG of an affine code that has no overhead, a more dynamic PTG that spends some time traversing meta-data structures in memory will add an overhead $T_O^{(2)}$ to the explicit dataflow model for non-affine programs. In that case, a skeleton program needs to run at

the beginning of the execution which populates the meta-data structures (details are described in [Section 4.1](#)). 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 T_p^{expl} it takes per node p to complete the computation of all tasks in the PTG for CC is then:

$$T_p^{expl} = \max \left(\frac{T_w \cdot N}{P \cdot C}, \frac{B}{P \cdot BW_{peak}} \right) + T_O^{(2)} \quad (6.5)$$

The current *implicit* dataflow version of NWChem CC comes with much higher overhead than the PTG version of NWChem CC. `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. These overheads, which will be referred to in the text as $T_O^{(3)}$, can be classified as follows:

- Calls to functions (e.g., `GET_HASH_BLOCK()`, `ADD_HASH_BLOCK()`) that transfer data over the network via the Global Arrays layer. This communication is blocking and 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 seven levels of the 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 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.

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, the fixed-size window of tasks that 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.

In this implementation, the runtime can never exploit more parallelism than the fixed-size window of tasks, which is the maximum number of tasks in a chain. Therefore, one must replace $(P \cdot C)$ with:

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

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

$$T_p^{impl} = \max\left(\frac{T_w \cdot N}{\min\left(\frac{N}{N_{chain}}, C\right) \cdot P}, \frac{B}{BW_{peak} \cdot P}\right) + T_O^{(3)} \quad (6.6)$$

One can see that for communication-bound algorithms, the second term of [Equation 6.6](#) 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 CC 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-barotene case study, the chain sizes are always less than or equal to the number of cores per node. More specifically, the statistics in [Figure 4.20](#) confirm that the chain size is either 8 or 16 for the most expensive subroutines, and smaller chain sizes (4, 6, 8) for less expensive subroutines.

Thus, the equation can be simplified to:

$$T_p^{impl} = \max\left(\frac{T_w \cdot N_{chain}}{P}, \frac{B}{BW_{peak} \cdot P}\right) + T_O^{(3)} \quad (6.7)$$

This means, 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 5.10 and Figure 5.11 that show that the scalability of all three implicit dataflow implementations (PARSEC, STARPU, OPENMP) stagnates after 8 cores per node.

Combining Equation 6.5 and Equation 6.7, we see that the explicit dataflow version of NWChem CC will execute with a speedup of:

$$Speedup = \frac{T_p^{impl}}{T_p^{expl}} = \frac{\frac{T_w \cdot N_{chain}}{P} + T_O^{(3)}}{\frac{T_w \cdot N}{P \cdot C} + T_O^{(2)}} \quad (6.8)$$

over the implicit dataflow version. This demonstrates the following:

1. For an increasing number of nodes P and a fixed number of cores C per node, the speedup of the explicit over the implicit CC code will converge to a constant $\propto \frac{T_O^{(3)}}{T_O^{(2)}}$. This can also be observed in the experiment when comparing the performance change when moving from 32 to 64 to 128 nodes. With an increasing size of P , while both dataflow versions continue to show a speedup:

$$32 \text{ to } 64 \text{ nodes: } Speedup^{expl} = 1.5 \quad \text{and} \quad Speedup^{impl} = 1.8$$

$$64 \text{ to } 128 \text{ nodes: } Speedup^{expl} = 1.3 \quad \text{and} \quad Speedup^{impl} = 1.8$$

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 a fixed number of nodes P and increasing number of cores C per node, the speedup of the explicit dataflow over the implicit dataflow version for NWChem CC will be proportional to:

$$\frac{T_p^{impl}}{T_p^{expl}} \approx \left(\frac{N_{chain}}{N} + \frac{T_O^{(3)} \cdot P}{T_w \cdot N} \right) \cdot C \quad (6.9)$$

Since for real-world problems the total number of tasks N is much greater than $\frac{T_O^{(3)} \cdot P}{T_w}$, Equation 6.9 can be further simplified because the second summand becomes negligible. Note, this does not apply for the first summand where numerator and denominator are both functions of N , thus, the fraction remains large for big N . Hence, the ratio of $\frac{T_p^{impl}}{T_p^{expl}}$ will become proportional to C :

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

Note, this relationship is only accurate if in Equation 6.8 the following holds true for the denominator $\frac{T_w \cdot N}{P \cdot C} \gg T_O^{(2)}$, which implies:

$$C \ll \frac{T_w \cdot N}{P \cdot T_O^{(2)}}$$

In the beta-carotene case study, $\frac{T_w \cdot N}{P \cdot T_O^{(2)}} \approx 1,000$, which is indeed much greater than C for most modern machines. It can be observed in the performance graphs for 32 (Figure 5.10), 64 (Figure 5.11), and 128 nodes (Figure 5.12), that, as the number of cores per node increases for a fixed P , the ratio of the implicit over the explicit dataflow version continues to grow with the 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 puts already 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 skeleton program ($T_O^{(1)} > T_O^{(2)}$).

This abstract model is not limited to NWChem CC and can be applied to any non-affine applications that exhibit a comparable control flow, or uses similar techniques and code structures.

Chapter 7

Related Work

Portions of this chapter are drawn from the following publications:

- Jagode, H., Danalis, A., Dongarra, J. “**Accelerating NWChem Coupled Cluster through dataflow-based Execution,**” *International Journal of High Performance Computing Applications (IJHPCA)*, SAGE Publications, pp. 1-13, <http://dx.doi.org/10.1177/1094342016672543>, January 2017.
- Jagode, H., Danalis, A., Bosilca G., Dongarra, J. “**Accelerating NWChem Coupled Cluster through dataflow-based Execution,**” *Parallel Processing and Applied Mathematics: 11th International Conference*, Krakow, Poland, September 6-9, 2015, Springer International Publishing, R. Wyrzykowski et al. (Eds.): PPAM 2015, Part I, LNCS 9573, pp. 366-376, 2016.
- Danalis, A., Jagode, H., Bosilca G., Dongarra, J. “**PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution,**” *2015 IEEE International Conference on Cluster Computing*, Chicago, Illinois, USA, IEEE, pp. 304-313, September 8-11, 2015.

I was responsible for the design and implementation of the CC dataflow versions corresponding to each of these publications. In addition, I served as the primary author.

An alternate approach for achieving better load balancing in the CC code is the Inspector-Executor methods [52]. This method applies performance model based cost estimation techniques for the computations to assign tasks to processors. 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 are applied to assign the aggregated tasks to processors. This technique focuses on balancing the computational cost without taking data locality into consideration, and without using a task-execution runtime.

ACES III [43] is another method that has been used effectively to parallelize CC codes. In this work, the CC algorithms are designed in a domain specific language called the Super Instruction Assembly Language (SIAL) [23]. This serves a similar function as the Tensor Contraction Engine (TCE) [6, 33], but with an even higher level of abstraction to the equations. The SIAL program, in turn, is run by a Multiple Program Multiple Data (MPMD) parallel virtual machine, the Super Instruction Processor (SIP) [44]. SIP has components that coordinate the work by tasks, communicate information between tasks for retrieving data, and then for execution. SIP is an MPI based program that uses multiple POSIX threads for communication.

The Dynamic Load-balanced Tensor Contractions framework [42] 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 Global Arrays (GA) [50] does not explicitly manage data redistribution, the communication pattern resulting from one-sided accesses is often irregular [59]. All of these solutions offer efforts to improve load balancing and scalability of tensor contractions, and are orthogonal approaches to the TCE altogether.

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 [12, 13, 25, 51, 65], where the compiler is expected to perform the parallelization of the input program. Habanero [31] 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 [49] is another compiler tool that utilizes the prototype runtime system Tarragon [14] 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 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 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 [30, 38, 57]. Unfortunately, most of these systems are impossible to use and evaluate today. Newer approaches, such as PM2 [47], SMARTS [61], Uintah [22], and Mentat [30] 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 “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 [53], Cilk [9], Thread Building Blocks (TBB) [Intel]), or accelerator based systems (StarPU [1], CellSs, GPUSs [8]). 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 Parameterized Task Graph (PTG) representation, in order to avoid the unnecessary overheads associated with DTD.

Chapter 8

Conclusions and Future Directions

The most popular programming model for current large scale computing is Coarse Grained 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 in the hardware landscape, featuring increasing complexity and heterogeneity of today’s and projected future high performance computers, one of the challenges for the mainstream programming model is to sustain the expected performance scalability on these systems. For instance, most of the computational chemistry methods are unable to take full advantage of current computer resources at national supercomputer centers, and are doomed to drop back even more on post-petascale systems.

This dissertation substantiates the hypothesis that *dataflow programming models* are well suited for computational chemistry applications, and provide a viable way to *achieve and maintain* computation at scale, especially on distributed, many-core, heterogeneous architectures. With the present discrepancy between the vast advances in *hardware* and without a corresponding match at the *software level*, there is little room left to dynamically manage diverse levels of parallelism and deliver performance portability across different systems. The movement toward more powerful and flexible programming models, away from traditional programming concepts, will enable easier transitions to emerging novel architectural designs, and more portable applications.

While the focus in this work is on Coupled Cluster methods (CC), the insights gained from this research are much more broadly applicable. The DAG execution models for Coupled Cluster algorithms, developed in this dissertation, will allow for broader and more flexible availability to more software levels of computational chemistry codes. The modernized codes are applicable to several key science drivers, such as studies of aerosol particles, soil chemistry, bio-systems, hormone-cofactor functionality in proteins, ionic liquids in cells, spectroscopies, new materials, light harvesting systems, and large-scale reaction mechanisms. We expect that the availability of dataflow-based CC implementations will transform the landscape of highly-accurate simulations; will provide an important example for employment with the multiple domain-specific, tensor-based methods; and will enable users to achieve an unprecedented level of accuracy in molecular modeling.

8.1 Conclusions

The main goal 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 allows capturing the essential properties of the algorithms (like the data dependencies). A task-scheduling runtime will then map the algorithms from the DAG representations to the hardware platforms. It is the responsibility of the runtime to manage architecture-aware scheduling of the tasks on a broad range of computing platforms, which, ultimately, lifts the burden of traditional programming models—namely, “programming at the communication layer”—from the domain scientists and allows them to focus on the efficiency of the *numerical algorithms* (and not *individual implementations* of the algorithms).

Chapter 3 introduced “implicit” and “explicit” dataflow programming models, and displayed the benefits of adopting dataflow-based executions over CGP in terms of: (a) communication-computation overlap, which is achieved automatically with explicit dataflow as communication is implicit; (b) support for heterogeneity of

compute resources, which is handled by the dataflow runtime as it dynamically schedules tasks based on the workload and available compute resources; and (c) freedom from control flow based limitations of CGP, which are often a result of the application being structured as a sequential code.

This work adopts dataflow programming models to the current state-of-the-art NWChem Coupled Cluster methods and evaluates different dataflow executions in terms of scalability, resource utilization, programmability. [Chapter 4](#) presented the *explicit dataflow* extensions of CC with the PARSEC framework, while the *implicit dataflow* extensions over different task-scheduling runtimes were discussed in [Chapter 5](#).

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. It also allows for the utilization of long established features from the original implementation, without diverging too much into an entirely different representation, like it is the case for the explicit dataflow model.

The implicit dataflow versions are executed with 3 runtimes: OPENMP for task-level parallelism in shared memory, and PARSEC and STARPU for task-level parallelism in distributed memory. 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. The OPENMP implementation has limits, though, for the reason that no additional parallelism can be afforded with the shared memory programming model. Consequently, the main difficulty is to ensure that there are no data dependencies between PARALLEL regions. In contrast, the implicit dataflow versions over PARSEC and STARPU come with several advantages as (1) there is no limitation to shared memory, and (2) programmability is more

transparent and less error-prone than OPENMP tasks (which depend on explicit attribute declarations for variables in the `PARALLEL`, `SINGLE`, `TASK` regions, etc.).

On the other hand, the explicit dataflow model demands a much bigger engineering job compared to the implicit dataflow models. The PTG programming paradigm 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 [20]. However, as demonstrated in [Chapter 4](#), even though the explicit dataflow model departs entirely from the original control flow of CC, the conversion from a CGP to PTG execution can happen gradually. In this work, the performance critical parts of CC have been selectively ported to execute over PARSEC and are then integrated back into the original NWChem framework to run seamlessly until completion.

Despite the lower startup overhead, in terms of software engineering, of implicit dataflow paradigms (by purely submitting tasks in the sequential flow of the original code), the significance of the increased implementation effort of the PTG becomes visible when comparing the superior performance of the explicit dataflow version of CC with the implicit dataflow and traditional CC computation. The PTG version of CC outperforms the original CC version by a significant margin—to be precise, by a factor of 2.6 on 32 nodes—even with only half of the CC kernels running over PARSEC while the remaining kernels are unchanged. 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/node)—demonstrating not only *a significant performance boost but also better scaling and greater utilization of compute resources due 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/node and are not able to take full advantage of the 16 available cores/node because both versions are tied to the limitations of the original control flow, such as 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 limits 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.

Chapter 6 presented a theoretical model that quantifies the differences and limitations of implicit and explicit dataflow models for affine and non-affine codes. The model demonstrates that explicit dataflow for affine codes will execute with a speedup proportional to the total number of compute cores on a machine. For implicit dataflow, where tasks are sequentially inserted, the DAG for a given execution is not known a priori. And so, the runtime needs to build the (entire or partial) DAG of the tasks in memory, which results in a constant overhead at the beginning of the execution.

For non-affine codes, such as NWChem CC, the explicit dataflow adds an overhead, which does not exist for affine codes, due to a more dynamic PTG that spends time traversing meta-data structures, etc. However, after paying that overhead the runtime has a PTG and communication is performed at the maximum throughput of the network. On the other hand, the current implicit dataflow version of CC comes with a much higher overhead because it inherits the overheads of the original control flow. Therefore, with an increasing number of nodes on a machine, the speedup of explicit over implicit dataflow CC will be proportional to a constant that depends on the two overheads $\frac{Overhead_{impl}}{Overhead_{expl}}$.

The abstract model in Chapter 6 is not limited to NWChem CC but is much more widely applicable to any non-affine applications with similar control flow and patterns.

8.2 Future Directions

This dissertation is a stepping stone toward more portable performance across a broad range of diverse computing platforms. Heterogeneity is a challenge that parallel

application scientists have to address in order to modernize their applications given the pervasiveness of accelerators and co-processors in HPC. In the CC dataflow models, presented in this work, work is naturally decomposed in tasks and therefore it can be divided between heterogeneous resources more readily than in the CGP model. A natural next step for the CC investigations would be to use the flexibility afforded by the additional parallelism to enable facile exploitation of new and interesting computing architectures (such as the Intel Xeon Phi processor, GPU accelerators) that perform best for the type of task under consideration.

In the explicit dataflow version of CC, tasks are currently scheduling in a round-robin fashion without taking data locality into consideration. Different scheduling methodologies could be investigated by considering the trade-offs between locality and work stealing. Also prioritizing tasks with specific characteristics can result in significant performance differences if, for instance, data movement in the critical path can be avoided with carefully assigned priorities and scheduling of these tasks.

With the emergence of power efficiency as a primary design constraint of anticipated compute systems, and reduced data movement as a primary programming goal, the need for energy efficient algorithms is inevitable. With the DAG model, groups of independent tasks can be treated as individuals entities, or aggregated into a single super-task. The judicious grouping of tasks with particular characteristics into such a super-task can be investigated to obtain the fastest execution time under a given power bound.

Bibliography

- [1] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198. [4](#), [10](#), [20](#), [102](#)
- [2] Aupy, G., Faverge, M., Robert, Y., Kurzak, J., Luszczek, P., and Dongarra, J. (2013). Implementing a Systolic Algorithm for QR Factorization on Multicore Clusters with PaRSEC. In *Euro-Par Workshops*, pages 657–667. [3](#), [10](#)
- [3] Ayguade, E., Copt, N., Duran, A., Hoefflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418. [74](#)
- [4] Bartlett, R. J. and Musial, M. (2007). Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291–352. [2](#), [12](#)
- [5] Bartlett, R. J. and Musiał, M. (2007). Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291–352. [12](#)
- [6] Baumgartner, G., Auer, A., Bernholdt, D., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C., Lu, Q., Nooijen, M., Pitzer, R., Ramanujam, J., Sadayappan, P., and Sibiryakov, A. (2005a). Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proceedings of the IEEE*, 93(2):276–292. [101](#)
- [7] Baumgartner, G., Auer, A., Bernholdt, D. E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R. J., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C.-C., Lu, Q., Nooijen, M., Pitzer, R. M., Ramanujam, J., Sadayappan, P., and Sibiryakov, A. (2005b). Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2). special issue on "Program Generation, Optimization, and Adaptation". [2](#)

- [8] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. (2006). CellSs: A programming model for the Cell BE architecture. In *SC'2006 Conference CD*, Tampa, FL. IEEE/ACM SIGARCH. [102](#)
- [9] Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., and Zhou, Y. (1995). Cilk: an efficient multithreaded runtime system. In *Proceedings Symposium on Parallel Algorithms and Architectures*. [102](#)
- [10] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaeif, H., Luszczek, P., YarKhan, A., and Dongarra, J. (2011). 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)*, pages 1432–1441. IEEE. [3](#), [10](#), [21](#)
- [11] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2012). DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38:37 – 51. Extensions for Next-Generation Parallel Programming Models. [4](#), [10](#)
- [12] Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21:291–312. [102](#)
- [13] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An Object-oriented approach to non-uniform Clustered Computing. *SIGPLAN Not.*, 40:519–538. [102](#)
- [14] Cicotti, P. (2011). Tarragon: a programming model for latency-hiding scientific computations. Ph. D. Dissertation, Department of Computer Science and Engineering, University of California, San Diego. [20](#), [102](#)

- [15] Coester, F. (1958). Bound states of a many-particle system. *Nuclear Physics*, 7:421–424. [12](#)
- [16] Cosnard, M. and Loi, M. (1995). Automatic Task Graph Generation Techniques. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, Washington, DC. IEEE Computer Society. [10](#)
- [17] Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science Engineering*, 5(1):46–55. [74](#)
- [18] Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2014a). PTG: an abstraction for unhindered parallelism. In *Proceedings of the International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. [10](#)
- [19] Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2014b). Ptg: An abstraction for unhindered parallelism. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14*, pages 21–30. IEEE Press. [10](#)
- [20] Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2014c). PTG: An Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. [94](#), [107](#)
- [21] Danalis, A., Jagode, H., Bosilca, G., and Dongarra, J. (2015). PaRSEC in Practice: Optimizing a Legacy Chemistry Application through Distributed Task-Based Execution. In *2015 IEEE International Conference on Cluster Computing*, pages 304–313. IEEE. [37](#)
- [22] de St. Germain, J. D., McCorquodale, J., Parker, S. G., and Johnson, C. R. (2000). Uintah: A massively parallel problem solving environment. In *HPDC'00*:

- [23] Deumens, E., Lotrich, V. F., Perera, A., Ponton, M. J., Sanders, B. A., and Bartlett, R. J. (2011). Software design of aces iii with the super instruction architecture. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(6):895–901. 101
- [24] Dongarra, J., Faverge, M., Hérault, T., Jacquelin, M., Langou, J., and Robert, Y. (2013). Hierarchical QR Factorization Algorithms for Multi-core Clusters. *Parallel Comput.*, 39(4-5):212–232. 3, 10
- [25] El-Ghazawi, T. A., Carlson, W. W., and Draper, J. M. (2003). Upc specification v. 1.1. <http://upc.gwu.edu/documentation>. 102
- [26] Faverge, M., Herrmann, J., Langou, J., Lowery, B. R., Robert, Y., and Dongarra, J. (2014). Designing LU-QR hybrid solvers for performance and stability. In *Proceedings of the 23rd International Heterogeneity in Computing Workshop, IPDPS 2014*, IEEE Computer Science Press, Phoenix, AZ. 3, 10
- [27] Ferreira, K. B., Bridges, P., and Brightwell, R. (2008). Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08*, pages 19:1–19:12, Piscataway, NJ, USA. IEEE Press. 9
- [28] Forum, M. P. I. (1997). MPI-2: Extensions to the message passing interface standard. <http://www.mpi-forum.org/>. 8
- [29] Gordon, M. and Schmidt, M. (2005). Advances in electronic structure theory: GAMESS a decade later. *Theory and Applications of Computational Chemistry: the first forty years*, pages 1167–1189. 2

- [30] Grimshaw, A., Strayer, W., and Narayan, P. (1993). Dynamic object-oriented parallel processing. In *IEEE Parallel and Distributed Technology: Systems and Applications*, pages 33–47. <http://www.cs.virginia.edu/~mentat/>. 102
- [31] Guo, Y., Barik, R., Raman, R., and Sarkar, V. (2009). Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA. IEEE Computer Society. 102
- [32] Hartono, A., Lu, Q., Henretty, T., Krishnamoorthy, S., Zhang, H., Baumgartner, G., Bernholdt, D. E., Nooijen, M., Pitzer, R., Ramanujam, J., and Sadayappan, P. (2009). Performance Optimization of Tensor Contraction Expressions for Many-Body Methods in Quantum Chemistry. *The Journal of Physical Chemistry*, 113(45):12715–12723. PMID: 19888780. 16
- [33] Hirata, S. (2003). Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A*, 107(46):9887–9897. 12, 101
- [34] Hoefer, T., Schneider, T., and Lumsdaine, A. (2010). Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA. IEEE Computer Society. 9
- [Intel] Intel. Intel threading building blocks. 102
- [36] Jagode, H., Danalis, A., Bosilca, G., and Dongarra, J. (2016). *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I*, chapter Accelerating

- NWChem Coupled Cluster Through Dataflow-Based Execution, pages 366–376. Springer International Publishing, Cham. [37](#)
- [37] Jagode-McCraw, H and Danalis, A and Herault, T and Bosilca G and Dongarra, J and Kowalski, K and Windus, T.L. (2014). Utilizing Dataflow-based Execution for Coupled Cluster Methods. pages 296–297. [12](#), [17](#), [39](#), [71](#)
- [38] Kale, L. (1993). Parallel programming with charm: An overview. Technical Report 93-8, Dept. of Computer Science, University of Illinois at Urbana-Champaign. [102](#)
- [39] Kowalski, K., Krishnamoorthy, S., Olson, R., Tipparaju, V., and Aprà, E. (2011). 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*, pages 1–10. [8](#), [12](#)
- [40] Lacoste, X., Faverge, M., Ramet, P., Thibault, S., and Bosilca, G. (2014a). Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes. [3](#)
- [41] Lacoste, X., Faverge, M., Ramet, P., Thibault, S., and Bosilca, G. (2014b). Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Proceedings of the 23rd International Heterogeneity in Computing Workshop, IPDPS 2014*, IEEE Computer Science Press, Phoenix, AZ. [10](#)
- [42] Lai, P.-W., Stock, K., Rajbhandari, S., Krishnamoorthy, S., and Sadayappan, P. (2013). A framework for load balancing of tensor contraction expressions via dynamic task partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13*, pages 13:1–13:10. ACM. [101](#)

- [43] Lotrich, V., Flocke, N., Ponton, M., Yau, A., Perera, A., Deumens, E., and Bartlett, R. (2008). Parallel implementation of electronic structure energy, gradient and hessian calculations. *J. Chem. Phys.*, 128, 194104, page 15 pages. [101](#)
- [44] Lotrich, V., Ponton, J., Perera, A., Deumens, E., Bartlett, R., and Sanders, B. (2010). Super instruction architecture of petascale electronic structure software: the story. *Molecular Physics*, 108(21-23):3323–3330. [101](#)
- [45] McCraw, H., Danalis, A., Herault, T., G, B., Dongarra, J., Kowalski, K., and Windus, T. (2014). Utilizing Dataflow-based Execution for Coupled Cluster Methods. In *Proceedings of IEEE Cluster 2014*, pages 296–297. IEEE. [37](#)
- [46] Medvedev, M. G., Bushmarinov, I. S., Sun, J., Perdew, J. P., and Lyssenko, K. A. (2017). Density functional theory is straying from the path toward the exact functional. *Science*, 355(6320):49–52. [11](#)
- [47] Namyst, R. and Méhaut, J.-F. (1996). PM^2 : Parallel multithreaded machine. A computing environment for distributed architectures. In D’Hollander, E., Joubert, G. R., Peters, F. J., and Trystram, D., editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo’95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam. Elsevier, North-Holland. [102](#)
- [48] Nataraj, A., Morris, A., Malony, A. D., Sottile, M., and Beckman, P. (2007). The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC ’07, pages 29:1–29:12, New York, NY, USA. ACM. [9](#)
- [49] Nguyen, T., Cicotti, P., Bylaska, E., Quinlan, D., and Baden, S. B. (2012). 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*, SC '12, pages 39:1–39:11, Los Alamitos, CA, USA. IEEE Computer Society Press. [102](#)
- [50] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Apra, E. (2006). Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231. [8](#), [13](#), [101](#)
- [51] Numrich, R. W. and Reid, J. K. (1998). Co-Array Fortran for parallel programming. *ACM Fortran Forum* 17, 2, 1-31. [102](#)
- [52] Ozog, D., Shende, S., Malony, A., Hammond, J. R., Dinan, J., and Balaji, P. (2013). Inspector/executor load balancing algorithms for block-sparse tensor contractions. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 483–484. ACM. [49](#), [101](#)
- [53] Pérez, J. M., Badia, R. M., and Labarta, J. (2008). A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 142–151, Tsukuba, Japan. [20](#), [102](#)
- [54] Purvis, G. and Bartlett, R. (1982). A Full Coupled-Cluster Singles and Doubles Model - the Inclusion of Disconnected Triples. *Journal of Chemical Physics*, 76(4):1910–1918. [12](#)
- [55] Raghavachari, K., Trucks, G. W., Pople, J. A., and Head-Gordon, M. (1989). A 5th-order Perturbation Comparison of Electron Correlation Theories. *Chemical Physics Letters*, 157(6):479–483. [12](#)
- [56] Rezac, J. and Hobza, P. (2013). Describing Noncovalent Interactions beyond the Common Approximations: How Accurate Is the "Gold Standard," CCSD(T)

- at the Complete Basis Set Limit? *Journal of Chemical Theory and Computation*, 9(5):2151–2155. [11](#)
- [57] Rinard, M. (1994). *The Design, Implementation, and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA. [102](#)
- [58] Schmidt, M., Baldrige, K., Boatz, J., Elbert, S., Gordon, M., Jensen, J., Koseki, S., Matsunaga, N., Nguyen, K., Su, S., Windus, T., Dupuis, M., and Montgomery, J. (1993). General Atomic and Molecular Electronic Structure System. *Journal of Computational Chemistry*, 14:1347–1363. [2](#)
- [59] Solomonik, E., Matthews, D., Hammond, J., and Demmel, J. (2013). Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 813–824. [101](#)
- [60] Taylor, D. E., ngyn, J. G., Galli, G., Zhang, C., Gygi, F., Hirao, K., Song, J. W., Rahul, K., von Lilienfeld, O. A., Podeszwa, R., Bulik, I. W., Henderson, T. M., Scuseria, G. E., Toulouse, J., Peverati, R., Truhlar, D. G., and Szalewicz, K. (2016). Blind test of density-functional-based methods on intermolecular interaction energies. *The Journal of Chemical Physics*, 145(12):124105. [2](#), [11](#)
- [61] Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malonv, A., Shende, S., Oldehoeft, R., and Smith, S. (1999). SMARTS: Exploting temporal locality and parallelism through vertical execution. In *Proceedings of the 1999 ACM SIGARCH International Conference of Supercomputing (ICS '99)*, pages 302–310. [102](#)
- [62] Valiev, M., Bylaska, E. J., Govind, N., Kowalski, K., Straatsma, T. P., Van Dam, H. J. J., Wang, D., Nieplocha, J., Aprà, E., Windus, T. L., and de Jong, W. (2010). NWChem: A comprehensive and scalable open-source solution for large

- scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489. [2](#), [12](#)
- [63] Čížek, J. (1966). On the correlation problem in atomic and molecular systems. calculation of wavefunction components in Ursell-type expansion using quantum-field theoretical methods. *J. Chem. Phys.*, 45:4256. [12](#)
- [64] YarKhan, A., Kurzak, J., and Dongarra, J. (2011). QUARK Users’ Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee. [20](#)
- [65] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. (1998). Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. [102](#)

Vita

Heike Jagode was born in Radebeul, Saxony, Germany to Karin and Bernd Jagode.

In 2001, Heike earned her Bachelor of Science degree in Applied Mathematics and her Master of Science degree in Applied Techno-Mathematics, both from the University of Applied Sciences Mittweida, Germany. Upon graduation, she joined the research team of Prof. Dr. Wolfgang Nagel at the Center for Information Services and High Performance Computing (ZIH) at the Dresden University of Technology, Germany. During this time, Heike worked on solutions in the field of scalable performance analysis on high performance computers, which inspired her to pursue a graduate degree in high performance computing. In 2006, Heike earned her second Master of Science degree in High Performance Computing from the University of Edinburgh, Scotland, UK.

In 2008, she joined the Innovative Computing Laboratory (ICL) research group of Prof. Dr. Jack Dongarra at the University of Tennessee, Knoxville, USA. She is a full time research scientist and specializes in high performance computing and the efficient use of advanced computer architectures; focusing primarily on developing methods and tools for performance analysis and tuning of parallel scientific applications. Her research interests include the multi-disciplinary effort to convert computational chemistry algorithms into a dataflow-based form to make them compatible with next-generation task-scheduling systems, such as PaRSEC. While working at ICL, Heike joined the doctoral program in computer science and graduated with her Doctorate of Philosophy in Computer Science from the University of Tennessee in May 2017.