PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution

Anthony Danalis*, Heike Jagode*, George Bosilca* and Jack Dongarra*†‡
*Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville
†Oak Ridge National Laboratory, Oak Ridge, TN
‡University of Manchester, Manchester, UK

***Abstract*—Task-based execution has been growing in popularity as a means to deliver a good balance between performance and portability in the post-petascale era. The Parallel Runtime Scheduling and Execution Control (PARSEC) framework is a task-based runtime system that we designed to achieve high performance computing at scale. PARSEC offers a programming paradigm that is different than what has been traditionally used to develop large scale parallel scientific applications.**

In this paper, we discuss the use of PARSEC to convert a part of the Coupled Cluster (CC) component of the Quantum Chemistry package NWCHEM into a task-based form. We explain how we organized the computation of the CC methods in individual tasks with explicitly defined data dependencies between them and re-integrated the modified code into NWCHEM.

We present a thorough performance evaluation and demonstrate that the modified code outperforms the original by more than a factor of two. We also compare the performance of different variants of the modified code and explain the different behaviors that lead to the differences in performance.

Keywords*—*PaRSEC; Tasks; DAG; PTG

I. INTRODUCTION

Large applications for parallel computers have been predominantly programmed in the Coarse Grain Parallelism model using MPI as the communication layer, or some abstraction layer built on top of MPI. The main drivers of this trend have been the relative simplicity of the CGP model, the high performance that such implementations can achieve, as well as the portability, 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. Task-based execution on-top of runtime systems has started emerging as an alternative programming paradigm to CGP and several success stories have encouraged multiple research groups to pursue this avenue. In this paper, we describe how we used PARSEC– the distributed task scheduling runtime system that we have developed in previous work [1] – to accelerate CCSD, a key component of the quantum chemistry software NWCHEM [2].

We argue that the traditional CGP programming model fails to deliver the expected performance scalability, especially with the increase in scale, complexity, and heterogeneity of modern supercomputers. While applications could be optimized further without departing from existing programming models, such optimizations would lead to significantly more complex code that is harder to port and maintain. In this paper we substantiate this claim by showing how a production strength, large scale, scientific application balances complexity and optimization in a way that leads to suboptimal performance.

The Parallel Runtime Scheduling and Execution Control (PARSEC) framework [3] 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. The main difference between PARSEC and other task engines is the way tasks, and their data dependencies, are represented, enabling PARSEC to employ a unique, symbolic way of discovering and processing the graph of tasks. Namely, PARSEC uses a symbolic Parameterized Task Graph (PTG) [4], [5] to represent the tasks and their data dependencies to other tasks.

PARSEC's programming paradigm proposes a complete departure from the way we have been designing and developing applications. However, as we demonstrate in this paper, the conversion from CGP to task 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.

The transformation discussed in this paper focuses on the iterative Coupled Cluster (CC) methods [6] of NWCHEM known as Coupled Cluster Single Double (CCSD), which is automatically generated by the Tensor Contraction Engine (TCE) [7]. Coupled Cluster constitutes highly accurate electronic methods and is used to address important topics such as renewable energy sources (e.g., solar and bio-renewables), efficient batteries, and chemical catalysis.

In the remainder of this paper, we briefly outline PARSEC and NWCHEM and describe the high level structure of the PARSEC integration into NWCHEM. Then we discuss the task-based implementation of the selected code and its various algorithmic variations, followed by a detailed analysis on the impact of these variations on performance. Finally, we offer a quantitative performance comparison between the original and modified CCSD code and qualitative explanations for this difference.

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.50

II. Overview of NWChem and PaRSEC

This section outlines NWChem and PaRSEC with its execution model that is based on the Parameterized Task Graph (PTG) abstraction.

A. NWChem

Computational modeling has become an integral part of many research efforts in key application areas in chemical, physical, and biological sciences. NWChem [2] is a molecular modeling software developed to take full advantage of the advanced computing systems available. NWChem provides many methods to compute the properties of molecular and periodic systems by using standard quantum-mechanical descriptions of the electronic wave function or density. The Coupled Cluster (CC) theory [6], [8], [9], [10] is considered by many to be a gold standard for accurate quantum-mechanical description of ground and excited states of chemical systems. Its accuracy, however, comes at a significant computational cost. One of the goals of this project is to strengthen the NWChem CC methods by enabling more powerful ways through dataflow-based task scheduling and execution, much better resource management, and a robust path to exploit hybrid computer architectures.

Tensor Contraction Engine: An important role in designing the optimum memory vs. cost strategies in CC implementations is played by the program synthesis system, the Tensor Contraction Engine (TCE) [7], which abstracts and automates the time-consuming and error-prone processes of deriving the working equations of second-quantized many-electron theories and synthesizing efficient parallel computer programs on the basis of these equations. All TCE CC implementations take advantage of the Global Arrays (GA) Toolkit [11], which provides a "shared-memory" programming interface for distributed-memory computers.

CC Single Double (CCSD): Especially important in the hierarchy of the CC formalism is the iterative CC model with single and double excitations (CCSD) [12]. This paper focuses on the CCSD version that has a computational cost of $O(N^6)$ and storage cost of $O(N^4)$, where N is a measure of the molecular system size. This version takes advantage of the alternative task scheduling (ATS) and the details of these implementations have been described in previous publications [13].

B. PaRSEC and the PTG programming model

The PaRSEC framework [3] is a task-based dataflow-driven system designed as a dynamic platform that can address the challenges posed by distributed heterogeneous hardware resources. The central component of the system, the *runtime*, orchestrates the execution of the tasks on the available hardware. Choices regarding the execution of the tasks are based on information provided by the user regarding the tasks that comprise the user application, and the dataflow between those tasks. This information is provided in the form of a compact, symbolic representation of the tasks and their dependencies known as a Parameterized Task Graph (PTG) [4], [5]. The runtime combines the information contained in the PTG with supplementary information provided by the user – such as the distribution of data onto nodes, and priorities, that define the relative importance of different tasks – in order to make efficient scheduling decisions.

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

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

  : descRR(L1)

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

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

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

Fig. 1: PTG for GEMM tasks organized in a chain.

The PTG can be understood as a compressed representation of the DAG that describes the execution of a task-based application. While a thorough description of the PTG is outside the scope of this paper (and can be found in [5]), we will use the example shown in Figure 1 to outline some important aspects that relate to the work presented in this paper.

This code snippet defines the GEMM task class. Tasks of this class are parameterized using parameters `L1` and `L2`. This class mimics the original CCSD code which has the GEMM operations organized in multiple parallel chains, with each chain containing multiple GEMMs that execute serially. In this PTG, `L1` corresponds to the chain number and `L2` corresponds to the position of a GEMM inside the chain to which it belongs. As can be seen in the figure, the number of chains and the length of each chain do not have to be known a-priori. PaRSEC can dynamically look them up in metadata structures that can be filled by an inspection phase during the execution of the program. 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` (the lines with the arrows), 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. Figure 2 shows the one line that must replace the four lines that define the dataflow of matrix C in order to change the organization of the GEMMs from a serial chain to a parallel execution followed by a reduction.

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

Fig. 2: PTG snippet for parallel GEMM tasks.

PARSEC is an event driven system. When an event occurs (i.e., a task completes), the runtime reacts by examining the dataflow defined in the PTG of the task. This reveals what future tasks can be executed based on the data generated by the completed task. Beyond scheduling tasks, the runtime also handles the data exchange between distributed nodes, thus it reacts to the events triggered by the completion of data transfers as well. When the hardware is busy executing application code – and thus no events are triggered – the runtime does not incur overhead.

Due to the PTG representation, all communication becomes implicit and thus is handled automatically by the runtime without user intervention. In MPI (or other Coarse Grain Programming models), 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()`.

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

III. CODE STRUCTURE

In this section, we highlight the basics necessary to understand the original parallel implementation of CC through TCE. In the second part, we then describe our design decisions of the dataflow version of the CC code and how it is integrated into the original workflow of NWCHEM.

A. Code Structure of NWChem CCSD

In NWCHEM, the iterative CCSD method, among other kernels, is generated through the TCE into multiple (more than 60) sub-kernels that are divided into so-called "T1" and "T2" subroutines for equations determining `T1` and `T2` amplitude matrices. These amplitude matrices embody the number of excitations in the wave function, where `T1` represents all single excitations and `T2` all double excitations. The underlying equations of these theories are all expressed as contractions of many-dimensional arrays or tensors (generalized matrix multiplications (`GEMM`)), of which there are typically many thousands in any one problem. The generated FORTRAN 77 code for the `T1` and `T2` subroutines contains most work in deep loop nests, including local memory management (e.g., via `MA_PUSH_GET()`, `MA_POP_STACK()`), calls to GA functions that transfer data over the network (i.e., `GET_HASH_BLOCK()`, `ADD_HASH_BLOCK()`), and last but not least, calls to the subroutines that perform the actual computation on the data (i.e., `SORT()`, `GEMM()`).

Parallelism of the current TCE generated CC code is coarse grained. The work inside the CC kernels is grouped into chains of multiple GEMM operations. The operations within each chain are executed serially while different chains are independent and can execute in parallel. In other words, the unit of work (i.e., a task) in the TCE generated code is an entire chain. To achieve load balancing, the different MPI ranks perform global work stealing in order to decide which chains will execute by each rank. However, the entire work is divided into seven different levels and there is an explicit synchronization step between those levels. This implies that the task-stealing model applies only within each level; and 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 work levels is achieved through shared variables that are atomically updated (read-modify-write) using Global Arrays operations. Reliance on atomic variables that are shared across all participating nodes in a distributed memory execution is bound to become inefficient at large scale, becoming a bottleneck and causing significant overhead.

B. Code Structure of CCSD over PaRSEC

In an effort to evaluate the suitability of a task scheduling runtime such as PARSEC for executing the CC methods, we conducted a study where we transformed part of NWCHEM's CCSD into a dataflow version. In this paper we focus our discussion on one of the CCSD subroutines, *icsd_t2_7()*. The rest of the NWCHEM code is oblivious to this change and the execution of the PARSEC-enabled CCSD subroutines is as seamless as the call to an external library procedure. Figure 3 provides a high-level overview of how the transformed subroutines, that run over PARSEC, are integrated into the original structure of NWCHEM.

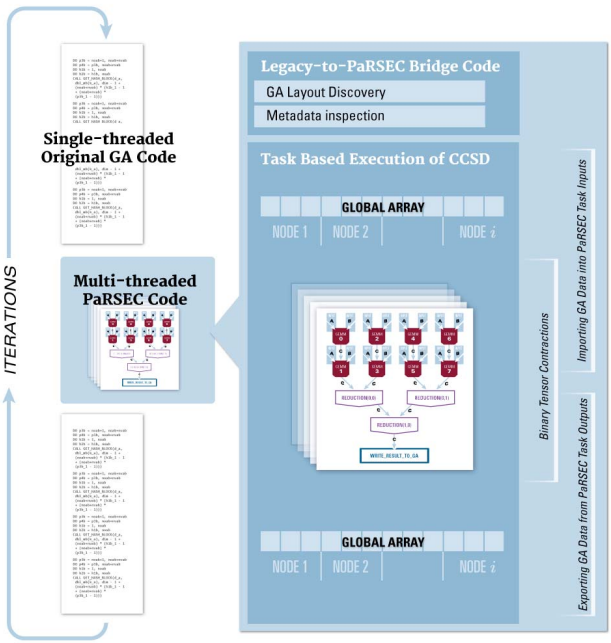

Fig. 3: High level view of PARSEC code in NWCHEM.

The loop nests that implement the chains of GEMMs in the original code contain `IF` branches. It follows, that each GEMM executes only if the conditions of the branches that enclose it evaluate to true.

As shown in the figure, our modified code starts with an inspection phase. During this phase the code computes the set of iteration vectors that lead to task executions – in other words, we discover all the loop iterations whose `IF` statements evaluate to `true`. In addition, the code queries the Global Array library to discover the physical location of the program data on which the GEMMs will operate on. The code that performs this inspection is derived from the original code of each subroutine. Specifically, 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. For example, a call to `GEMM()` is replaced with code that stores the pointers to the data that is involved in this GEMM operation as well as the iteration vector (the values of the induction variables of all loops enclosing the call) into a meta-data array. The location in this array where this information is stored is determined by the location of each GEMM in the chain of GEMMs and the chain number. The location in the chain and the chain number are computed by the same inspection code based on the occurrence of calls to subroutines such as `DFILL()` which initiates a chain of GEMMs and `ADD_HASH_BLOCK()` which terminates it.

After the inspection phase is finished, our 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, and the actual pointers to these data. At this point PaRSEC starts the execution of the tasks. The execution starts with "read" tasks that pull the data from the Global Array into PaRSEC-managed memory and pass this memory to the tasks that execute the actual tensor contractions. As the tensor contractions finish, "writer" tasks take the output data and push it back into the GA memory. At this point, PaRSEC returns control back to the original NWChem code.

It is important to note that part of the aforementioned work that is necessary in the current version of the code, will become unnecessary as soon as the complete CCSD code has been converted to PaRSEC. Specifically, data will not need to be pulled and pushed into the GA at the beginning and end of each subroutine if all subroutines execute over PaRSEC. Instead, the different PaRSEC tasks that comprise a subroutine will pass their output to the tasks that comprise another subroutine using the internal communication engine of PaRSEC – and will do so implicitly, since PaRSEC handles communication internally, without user involvement.

IV. Design Decisions

A. Algorithmic Variations

When we designed the PaRSEC implementation of CCSD 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. In this section we present several variations of our algorithm, each using a different ordering of tasks, and we discuss the impact of these variations on performance. 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 CCSD 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 the scope of a subroutine. Matrix multiplication itself is noncommutative (i.e., $A \cdot B \neq B \cdot A$), however the GEMM kernel does not perform only a multiplication, but also an addition, $C = C + A \cdot B$. 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, increases available parallelism, but decreases locality, since different chains work on different private C matrices.

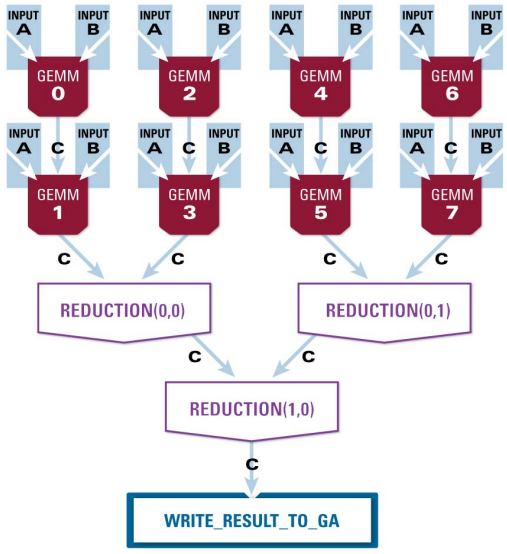

Fig. 4: Parallel GEMM operations followed by reduction.

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 paper 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. 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 V.

In Figure 4 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.

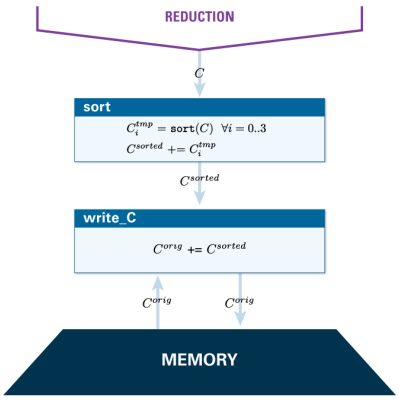

Fig. 5: Serialized sort and single write.

The simplest variation, which is depicted in Figure 5, executes all operations in serial, but in a different order than the original CCSD 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.

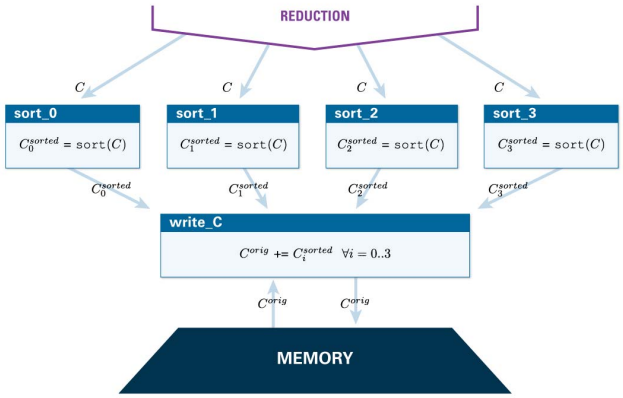

Fig. 6: Parallelized sort and single write.

The next variation involves parallelization of the `SORT` task. Namely, as shown in Figure 6, 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 `SORT_`i 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

SORT_i 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.

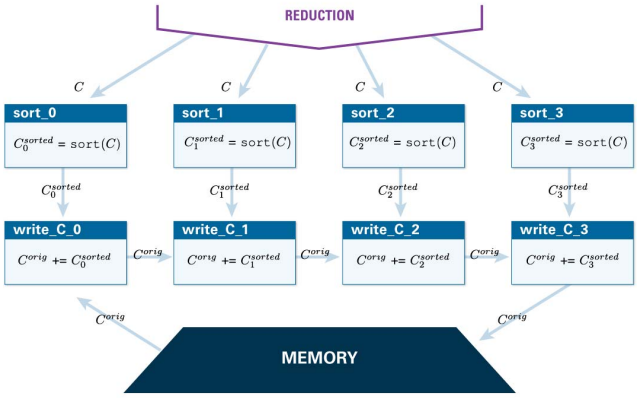

Fig. 7: Parallelized sort and write

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 7. In this variation, there are multiple WRITE_C_i tasks, in addition to the multiple SORT_i tasks. Each SORT_j task (that executes) sends its output matrix to the corresponding WRITE_C_j 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.

B. *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 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 5, 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 8 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 is implemented such that each task instance only

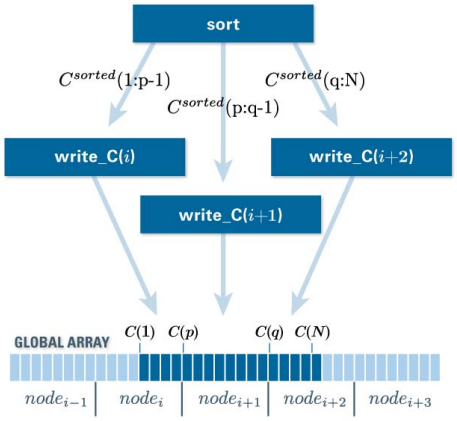

Fig. 8: Exporting data to GA.

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.

C. *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[1]. 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:

```
max_L1 - L1 + offset*P
```

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 $5 * P$.

D. Load Balancing

The original NWChem code aims to achieve load balancing through a work stealing scheme that is often referred to as NXTVAL [14]. 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.

V. Performance Evaluation

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 IV-A. In particular we timed the following five variants of the algorithm:

[1]As we mentioned in Section III, 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.

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 9 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.

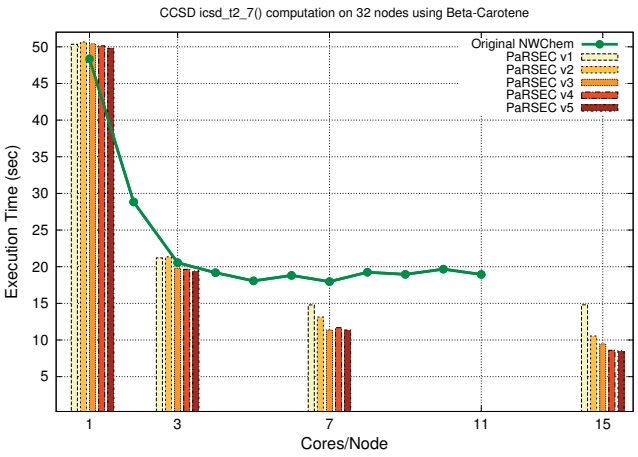

Fig. 9: Comparison of algorithm variations and original code.

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 Figures 10 and 11 respectively. These traces were generated using PARSEC's native performance instrumentation module.

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

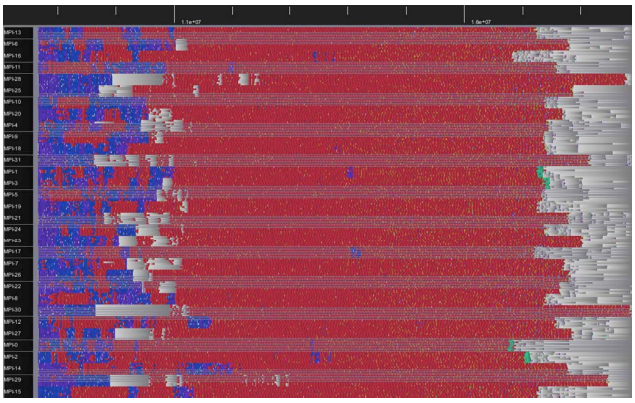

Fig. 10: Trace of v4 (priority decreasing with chain number).

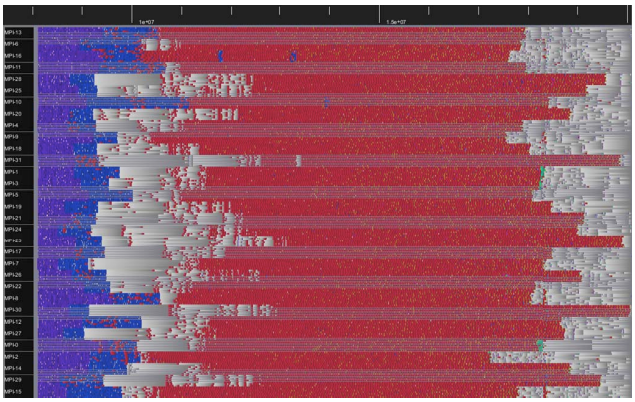

Fig. 11: Trace of v2 (no task priorities).

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.

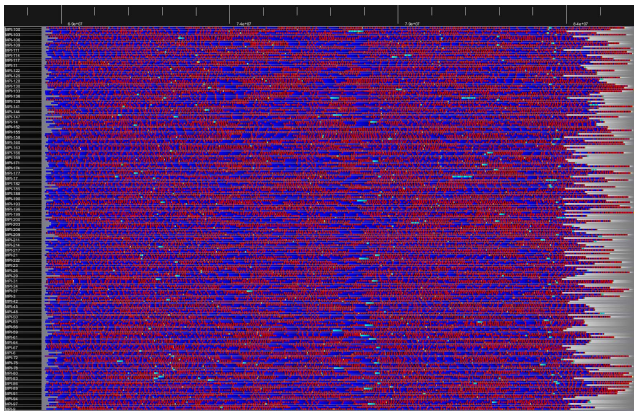

Fig. 12: Trace of original NWChem code.

Figure 12 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.

In Figure 13 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.

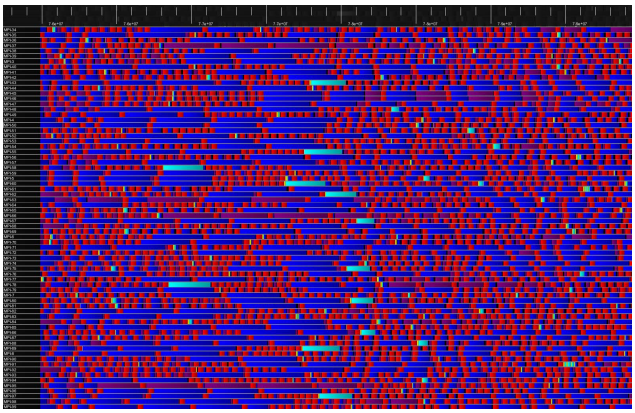

Fig. 13: Zoomed in trace of original code.

VI. Related Work

Task based execution is being pursued by several research groups and software companies around the world. As a result, several solutions are being currently actively developed [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. While their level of maturity, feature set and similarity with PARSEC vary, none of them supports the PTG programming model. They largely rely on some form of "Dynamic Task Discovery (DTD)", or in other words building the entire DAG of execution in memory using skeleton programs. While PARSEC also uses an inspector phase to collect information about the meta data of the program, this is hardly equivalent to the DTD approach. Our inspector phase does not build a DAG in memory and does not need to discover the way tasks depend on one another by matching input and output data. Rather, this information is expressed by the developer in the PTG and the inspector merely collects information such as the number of chains and the size of each chain. As a consequence, using PARSEC gives us the flexibility to readily express and test different algorithmic variations (as discussed in Section IV-A).

The use of an inspector-executor model for optimizing NWCHEM has been proposed before [14]. However, Ozog et al. use this approach to assess the cost of different tasks and focus on improving the load balance of the application. They do not use a task-execution runtime, nor do they explore variations of the base algorithm, as we do in this paper. Other efforts to improve load balancing and scalability of tensor contractions, such as the Cyclops Tensor Framework [25], or the Dynamic Load-balanced Tensor Contractions framework [26] offer orthogonal approaches to the TCE altogether.

VII. Conclusions

In this paper we described our effort to utilize our task-based execution system PARSEC to optimize part of NWCHEM, a legacy Quantum Chemistry application written using FORTRAN 77 and Global Arrays. We discussed how PARSEC is used from within NWCHEM and perform a detailed analysis of the different algorithmic choices of our algorithm as well as their impact on performance. 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.

Finally, we demonstrated a significant performance improvement (2.69x speedup) that we achieved by executing the NWCHEM Coupled Cluster code over PARSEC instead of employing the antiquated Coarse Grain Parallelism programming model that, to this day, remains the prevalent model for developing scientific applications.

ACKNOWLEDGMENT

This material is based upon work supported in part by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476, and the DOE Office of Science, Advanced Scientific Computing Research, under award No. DE-SC0006733 "SUPER - Institute for Sustained Performance, Energy and Resilience", and in part by the Russian Scientific Foundation, Agreement N14-11-00190. A portion of this research was performed using EMSL, a DOE Office of Science User Facility sponsored by the Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory.

REFERENCES

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

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

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

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

[5] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "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)*, Nov 2014.

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

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

[8] R. J. Bartlett and M. Musiał, "Coupled-cluster theory in quantum chemistry," *Reviews of Modern Physics*, vol. 79, no. 1, pp. 291–352, 2007.

[9] F. Coester, "Bound states of a many-particle system," *Nuclear Physics*, vol. 7, pp. 421–424, Jun. 1958.

[10] J. Čížek, "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.*, vol. 45, p. 4256, 1966.

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

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

[13] K. Kowalski, S. Krishnamoorthy, R. Olson, V. Tipparaju, and E. Aprà, "Scalable implementations of accurate excited-state coupled cluster theories: Application of high-level methods to porphyrin-based systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov 2011, pp. 1–10.

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

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

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

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

[18] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

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

[20] O. A. R. Board, "OpenMP Application Program Interface, Version 4.0," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[21] S. Chatterjee, S. Tasrlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating Asynchronous Task Parallelism with MPI," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 712–725.

[22] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *Int. J. High Perf. Comput. Applic.*, vol. 23, no. 3, pp. 284–299, 2009.

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

[24] The OCR team, "The OCR project," https://01.org/open-community-runtime.

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

[26] P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, and P. Sadayappan, "A framework for load balancing of tensor contraction expressions via dynamic task partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013, pp. 13:1–13:10. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503290