Tensor contraction on distributed hybrid architectures using a task-based runtime system

George Bosilca
ICL, University of Tennessee
Knoxville, TN, USA

Damien Genet
ICL, University of Tennessee
Knoxville, TN, USA

Robert J. Harrison
IACS, Stony Brook University
Stony Brook, NY, USA

Thomas Herault
ICL, University of Tennessee
Knoxville, TN, USA

Mohammad Mahdi Javanmard
IACS, Stony Brook University
Stony Brook, NY, USA

Chong Peng
Department of Chemistry, Virginia Tech
Blacksburg, VA, USA

Edward F. Valeev
Department of Chemistry, Virginia Tech
Blacksburg, VA, USA

***Abstract*—The needs for predictive simulation of electronic structure in chemistry and materials science calls for fast/reduced-scaling formulations of quantum n-body methods that replace the traditional dense tensors with element-, block-, rank-, and block-rank-sparse (data-sparse) tensors. The resulting, highly irregular data structures are a poor match to imperative, bulk-synchronous parallel programming style due to the dynamic nature of the problem and to the lack of clear domain decomposition to guarantee a fair load-balance. TESSE runtime and the associated programming model aim to support performance-portable composition of applications involving irregular and dynamically changing data. In this paper we report an implementation of irregular dense tensor contraction in a paradigmatic electronic structure application based on the TESSE extension of *PaRSEC*, a distributed hybrid task runtime system, and analyze the resulting performance on a distributed-memory cluster of multi-GPU nodes. Unprecedented strong scaling and promising efficiency indicate a viable future for task-based programming of complete production-quality reduced-scaling models of electronic structure.**

***Index Terms*—**

I. Introduction

As the future of high performance computing is transitioning towards increasingly hybrid hardware, with deep memory hierarchies and a growing number of computational resources (CPU cores or different types of accelerators), it becomes unquestionable that for this transition to be successful it needs to be supported by a similar transition in the way these resources are managed and used. Future applications will not thrive due to core peak performance or physical frequency increases but by exposing a larger and more varied degree of parallelism from our applications and relying on the hardware and software infrastructure to maximize its performance. There will be no single path to success but a clever combination of techniques, with few common demands: an increase in asynchrony and a finer granularity of parallelism. As long as we don't have the tools to automatically extract these from sequential source code, the burden to expose the intrinsic algorithmic parallelism remains on software developers. In a traditional programming environment, software developers are required in addition of exposing parallelism from the algorithms, to manage the resources and also to decompose and express their computations in a way that is portable among shared and distributed memory machines with widely varying configurations. To address the challenges of efficiently utilizing this type of heterogeneous resources we need programming paradigms that provide the ability to express parallelism in more flexible and productive manners. MPI and OpenMP are two of the most popular programming models for parallel applications, and their impact on the computational science is undeniable. However, they both encourage a practice of parallel programming for hero-programmers, where the developers perform multiple jobs: express parallelism, manage the computational resources and communications, and programmatically provide the mapping between these two. These burdens become heavier with the increase in core and node count, in heterogeneity of computational resources and application size.

At the opposite of the spectrum, task based runtime systems have become popular in tackling such challenges and making it easier to write parallel HPC applications. Runtimes relieve the users from managing low-level resources and give them the opportunity to focus on writing parallel applications by describing the potential parallelism in a way that is comprehensible and exploitable by them. A task-based runtime expects the users to express their computations and the data on which the computations will be performed in a way where computations become entities (aka. kernels or tasks) and the data flowing among them are the dependencies. Runtimes then create a complete, or in some cases partial, Directed Acyclic Graph (DAG) of tasks based on these dependencies, and will map these tasks on the available resources to achieve a correct, and possibly efficient execution. Thus, the major challenge of using a runtime is not only on the capabilities of the runtime, but also on the expressivity of the language or API the runtime provides for expressing the graph of tasks, and on providing the needed parallelism to maximize resource occupancy.

As advanced science applications seek to expand the scale and fidelity of systems being simulated, they must reduce algorithmic/computational complexity while controlling or

NSF #1450300, #1450344, #1450262 and Exascale Computing Project (17-SC-20-SC)

even improving accuracy and robustness. In computational chemistry and material science these goals are achieved by exploiting dynamic (i.e., discovered during the course of computation) sparsity and low-rank structures that together combine to greatly increase both the complexity of the software and the irregularity of the computation, while also reducing the granularity of computation. There is thus a fundamental tension between adopting advanced algorithms and realizing high-performance on current supercomputer systems, such as hybrid systems accelerated with multiple GPUS that represent a path to exascale simulation.

In the TESSE (Task-based Environment for Scientific Simulation at Extreme ScalE) project we have employed application-driven design to create a general-purpose software framework that attacks the twin challenges of programmer productivity and portable performance for advanced scientific applications on massively-parallel, hybrid, many-core systems. TESSE (1) extends the successful *PaRSEC* runtime and execution model [1] to support more irregular and dynamic applications, (2) defines a new programming model for composing sparse algorithms in modern C++ that leverages concepts of general flow-based programming and more specifically the *PaRSEC* parameterized task graph [2] for dense linear algebra [3], and (3) develops a new generation of science applications that builds upon these tools. Crucial to success is the appropriate partitioning of responsibilities between the runtime, the parallel-programming model, and the application.

In this paper, we focus upon the TESSE runtime and demonstrate, for the first time, key components of a many-body chemistry application executing on a distributed-memory computer with fully distributed data and utilizing multiple GPUs per node. We describe and demonstrate progress towards an extension to the *PaRSEC* runtime for irregular applications executing on distributed memory, hybrid architectures. In particular, contractions of 4-index block-distributed (and eventually block-sparse) tensors are mapped to matrix-multiply operations with irregular tiles. Key challenges overcome and features demonstrated are

- obtaining high-performance with matrix operations on strongly non-square matrices,
- a model based approach starting from single node multi-GPU benchmark data,
- an intelligent runtime for irregularly-tiled data that load balances and routes work to CPU/GPU appropriately based upon size/shape,
- the use of futures to integrate the new component on the TESSE runtime into the existing application that uses the MADNESS parallel runtime,
- experiments that demonstrate and analyze the application performance on up to 16 nodes with multiple GPUs/node.

II. Motivating Science Application

Although the objectives of TESSE runtime are domain-neutral, a particular key science application – namely, accurate simulation of the electronic structure of molecules and solids – was chosen to drive the development of TESSE. Predictive simulation of electronic structure involves first-principles solution of the quantum mechanical equation of motion for many (potentially, an infinite number of) electrons; exact solutions are not known for even 2 electrons (outside of few models) and even for a finite discretization the problem is NP-hard. Robust approximate methods exist, such as coupled-cluster [4] and many-body Green's function approaches, but they are *expensive*, i.e. they have high-order polynomial operation and space complexity; for the foundational Coupled Cluster Singles and Doubles method (CCSD) these are N^6 and N^4, respectively. The high complexity limits the applicability of conventional (naive) formulations of predictive methods to systems with a few (5-10) atoms on a single workstation, and a few dozen (50-100) atoms on a supercomputer [5]. However, recent emergence of robust fast/reduced-scaling formulations has greatly extended the applicability of such methods to hundreds of atoms on a single workstation [6]. Such formulations replace the usual dense tensors with block-sparse and/or block-rank-sparse tensors, generally referred to as data-sparse. Thus one of the concrete goals of TESSE was to allow high-level composition of performant data-sparse tensor algebra required by the reduced-scaling electronic structure methods on modern distributed-memory heterogeneous computer platforms.

Since there are dozens of terms in the nonlinear algebraic equations that define even the simplest target method, CCSD, in a finite basis number, to understand application performance it is sufficient to focus on the representative, and usually the most expensive term (accounting routinely 90% or more), in the CCSD equation,[1] often referred to colloquially as the *ABCD* term:

$$R^{ij}_{ab} = \sum_{cd} T^{ij}_{cd} G^{cd}_{ab} + \ldots, \qquad (1)$$

where the elements of tensor T are the model parameters to be determined iteratively (in typically 10-20 iterations) so that tensor R vanishes. Tensor G is fixed (does not change between iterations). Ranges of all indices are proportional to system size N, hence each tensor has N^4 space complexity, and the operation has N^6 operation complexity.

The tensor contraction in Eq. (1) can be viewed as a multiplication of matrix T (with fused indices ij and cd playing the role of row and column indices, respectively) with square matrix G (with cd and ab row and column indices). In practice the range of *unoccupied* indices ($abcd$) has rank U that's a factor of 5-20 larger than the corresponding rank O of the *occupied* indices ij, hence transposes of matricized tensors T and R are very tall and skinny matrices, with aspect ratios of 25-400! *Optimal formulation* of dense matrix multiplication on distributed-memory systems [7], including for rectangular matrices [8], is relatively well understood. However, translating these advances in dense linear algebra to advances in electronic structure involves several hurdles:

[1]The permutational symmetries of tensors T, G and R, which are essential for proper physics as well as optimal operation count, are neglected for simplicity.

- A *realistic implementation* of Eq. (1) may violate the assumption of regularity that lend to formally perfect load balance of traditional algorithms, e.g. ranges $abcd$ may be irregularly tiled due to particular structure of the basis, as in the so-called integral-driven formulation of the ABCD term in which G is evaluated on the fly.
- An *optimal* formulation of Eq. (1) calls for exploiting the data sparsity in T, G, and R that becomes apparent by an appropriate choice of the basis in which to express these tensors; this reduces the operation complexity from N^6 to N [6].
- Data sparsity of tensors in electronic structure assume a variety of forms, e.g. G is block sparse, whereas tensors T and R have blocks that are rank and eventually (for large enough systems) element sparse.

It is clear that high-performance computing with irregularly-tiled and/or data-sparse tensorial data structures is a poor match to imperative, bulk-synchronous parallel programming style and execution models due to the irregular (and potentially dynamic) structure of the data. The goal of this paper is to demonstrate how modern task-based dataflow-style execution can be used to achieve high performance on a distributed-memory heterogeneous cluster with multi-GPU nodes; as a first step towards fully data-sparse formulation, we consider an irregularly-structured dense tensor contraction of the ABCD term in Eq. (1), as implemented in the open-source Massively Parallel Quantum Chemistry (*MPQC*) program.

III. Background

This section describes the *PaRSEC* runtime and some of the user-level APIs used to interact with the runtime. We briefly describe the initial implementation of the integration of the electronic structure program *MPQC* with *PaRSEC*, and then delve further into some optimizations made to improve upon the original implementation.

a) PaRSEC: [1] is a task-based runtime for distributed heterogeneous architectures, capable of dynamically unfolding a concise description of a graph of tasks on a set of resources and satisfying all data dependencies by shepherding data between memory spaces (including between nodes) and scheduling tasks on heterogeneous resources. Instead of a fixed API, *PaRSEC* facilitates the design of Domain Specific Languages (DSL) that allow domain experts to focus on their science rather than on the computer science. These DSLs rely on a dataflow model to create dependencies between tasks, and are expected to maximize the description of the parallelism available at the algorithm level to allow the runtime to exploit the available parallelism present in application. *PaRSEC* is rich with many features aimed at helping developers express their application to the runtime correctly and efficiently, and is supported by a set of tools to debug, profile and optimize the *PaRSEC* codes. In the context of this paper, we will focus on the programmability of the runtime, and how domain scientists describe their algorithm and interact with the runtime.

Certainly the most exposed DSL, PTG, allow users to use a parameterized task graph (PTG) [2] known as Job Data Flow (JDF) which handles the dependencies between tasks. To enhance the productivity of the application developers, *PaRSEC* implicitly infers all the communication from the expression of the tasks, supporting one-to-many and many-to-many types of communications. The runtime has been designed to excel in heterogeneous distributed systems and has been extensively tested for performance in different contexts. From a performance standpoint, algorithms developed using PTG are capable of delivering a significant peak performance on many hybrid distributed machines, as highlighted in [2] where DPLASMA, a dense linear algebra (DLA) library using *PaRSEC*, yields superior performance compared with the most widely used DLA library, ScaLAPACK [9]; or compared with state-of-the-art computational chemistry applications [10], [11]. Other DSLs, such as Dynamic Task Discovery (DTD) [12], are less science domain oriented, and provide alternative programming models, to satisfy more generic needs by delivering an API that allows for sequential task insertion into the runtime. This programming model is simple and straightforward, and has been shown to deliver decent level of performance at regular scales, but it suffers some drawbacks that limit its scalability as described in more details in Section VI.

Multiple components constitute the *PaRSEC* runtime: programming interfaces (DSL), schedulers, communication engines and data interfaces. The runtime uses a modular component architecture (MCA), allowing different modules or instances to be dynamically selected during runtime, providing a varied set of capabilities to different instances of the runtime (such as scheduling policies, or support for heterogeneity). A well-defined API for these modules transforms them into black boxes, and allows interested developers or users to implement their own, application specific, policies. The different DSL share the same runtime, data representation, communication engine, scheduler, cohabiting over the same set of hybrid resources and seamlessly inter-operate in the context of the same application.

Traditionally, application developers have a propensity to write sequential code. *PaRSEC*, with the help of a pre-compiler, transforms some form of sequential code to PTG, with the limitation that the sequential code must be affine [13]. However, neither the runtime, nor the PTG language itself are bound by the same constraints, providing a way to overcome the loop-affine limitations.

b) MADNESS: started as an environment for fast and accurate numerical simulation in chemistry [14], [15], but rapidly expanded to include applications in nuclear physics [16], boundary value problems [17], solid state physics. Computations beyond 3D include time evolution in an intense laser pulse in 4D [18], and in 6D the first ever numerical computation of the MP2 energy of a nonlinear molecule [19]. A prototype high-level DSL [15] demonstrates the feasibility of elevating programs to a very high level. To guarantee precision, every function (and there may be thousands of these in a large electronic structure calculation) has an independent and dynamically refined "mesh," and composing functions or applying operators can change the mesh refinement. These meshes are represented as 2^d-trees, where d is the dimensionality of the problem. These trees are typically poorly balanced and change dynamically.

Other projects such as *TiledArray*, over which *MPQC* was originally implemented, also employ the *MADNESS* parallel runtime. The *MADNESS* runtime has evolved into a powerful environment for the composition of a wide range of parallel algorithms on many distributed data structures including trees in *MADNESS* and the sparse tensors in *TiledArray*. The central elements of the parallel runtime are a) futures for hiding latency and managing dependencies, b) global namespaces with one-sided access, c) remote method invocation in objects in global namespaces, and d) dynamic load balancing and data redistribution. An SPMD model is provided with a single logical main thread per process, a thread pool to execute tasks, and a thread dedicated to serving remote active messages. *MADNESS* can be configured to use its own thread pool implementation, or to use Intel TBB or *PaRSEC*. An application in the *MADNESS* runtime can be viewed as a dynamically constructed DAG, with futures as edges. One of the goals of TESSE is then to complement the *MADNESS* runtime by enabling much more powerful ways to specify and schedule the DAG, much better resource management, and a robust path to exploiting hybrid computer architectures.

c) TiledArray: is a modern C++ framework for parallel tensor algebra [20]. The original motivation for *TiledArray* was to support the development of reduced-scaling electronic structure methods in *MPQC* and other platforms, and to explore the potential of task-based computation style for data-sparse tensor algebra. *TiledArray* is implemented on top of task-based parallel runtime of *MADNESS*. Not only does the task-based formulation make it possible to design scalable algorithms for data-sparse tensor computation [21], but has advantages even for dense tensor algebra by overlapping communication and computation [21], *TiledArray* is the foundation for high-performance implementation of coupled-cluster methods in the latest reengineering of the *MPQC* program, with demonstrated utilization of 128,000 cores of IBM BlueGene/Q for the largest-to-date canonical coupled-cluster singles and doubles computation with essentially exact numerics [5].

The current form of *TiledArray*, already useful in practice, is limited by the relatively low-level explicit task composition style of the *MADNESS* runtime. This makes formulation of even simple data flows like that involved in SUMMA [7], a general matrix multiplication algorithm used to implement tensor contraction in *TiledArray*, involves relatively complex manual composition of DAG of tasks, with programmer fully responsible for computation placement and resource management. For example, to manage for finite bandwidth the parallelism is throttled manually by inserting artificial task dependencies in the SUMMA DAG [21]. Another issue is the lack of native support in *MADNESS* for heterogeneous execution (although as part of this and other [22] recent efforts we demonstrated how it is possible to extend *MADNESS* directly for efficient execution on GPUs) The TESSE runtime and programming model promise numerous benefits to *TiledArray*, including 1) improved performance for dense and block-sparse tensor algebra, 2) improved resource management, 3) robust support for computation on CPU/accelerator platforms (this is the topic of this paper), and 4) easier implementation of unstructured data and computation flow patterns characteristic of data-sparse tensor algebra.

IV. Integration of *PaRSEC*, *MADNESS* and *TiledArray*

Porting large applications, like *MPQC*, over a new runtime system is a complex task that requires significant developer involvement, and may prevent adoption. For this reason, in this paper we focused on porting the *critical* part of the CCSD chemistry application to the new runtime, namely the ABCD contraction in Eq. (1), and analyzing its performance. This already posed a substantial technical challenge of extending the tensor framework *TiledArray* to seamlessly integrate with *multiple* task runtimes: the default *MADNESS* task runtime and the *PaRSEC* runtime. This section outlines how the integration was accomplished.

TiledArray represents tiled tensors as distributed hash tables of *MADNESS* futures to tiles. The use of futures allows to fully decouple scheduling and execution of tensor operations as well as support a variety of data-sparse operations naturally. There are 2 ways that *PaRSEC* runtime has been integrated into *TiledArray*: (1) by seamless offload of *MADNESS* tasks to *PaRSEC* runtime, and (2) by replacing entire native implementations of *TiledArray* tensor algebra operations by their *PaRSEC*-based counterparts. In this paper we are using the latter route, i.e. *TiledArray* implementation of tensor contraction (based on task-based formulation SUMMA [21]) was replaced by the *PaRSEC*-based variant. This required orchestrating transfer of {in,out}put (futures to) data between *MADNESS* and *PaRSEC*. The *PaRSEC* implementation of SUMMA for *TiledArray* exposes the *MADNESS* futures that represent each tile of input and output tensors as *PaRSEC* tasks that are not ready at the beginning of the execution, and registers a callback with each future to schedule a corresponding input task in *PaRSEC*. As the futures are set by *MADNESS* during the execution, these tasks become executable. They are selected for execution by the *PaRSEC* runtime, and when executed, they broadcast their data (coming directly from the memory set by the future) to the successor tasks in the distributed GEMM operation written in *PaRSEC*.

From this point, the data flows within the *PaRSEC* runtime and is untouched by the *MADNESS* runtime or the *TiledArray* application, until the data is used by all tasks in the *PaRSEC* operation. The last use of each data (in read or read-write mode) triggers a final output for *PaRSEC*, in which we set corresponding *MADNESS* futures. This triggers the rest of the operation, back at the *MADNESS* runtime level.

Both runtimes progress simultaneously: as the integration is done at the lowest level of granularity, it is possible that *MADNESS* operations are required to release futures that are waiting to make *PaRSEC* operations progress, and vice-versa. The integration does not require strong synchronization, neither between the nodes of a distributed run or between the threads of a single node. As a consequence, both runtimes need to use threads simultaneously.

Instead of oversubscribing the cores with duplicate threads, risking a high level of involuntary context switches, or space sharing the computing resources by dedicating some cores to *PaRSEC* threads and others to *MADNESS* threads, we provide a *PaRSEC* backend runtime for *MADNESS*: *MADNESS* uses *PaRSEC* to schedule tasks on different cores, and we used the state machine of *PaRSEC* tasks to trigger the different steps of a task progress in the *MADNESS* runtime. All computing resource are under the control of a single runtime, that provide computing capabilities to the other runtime system.

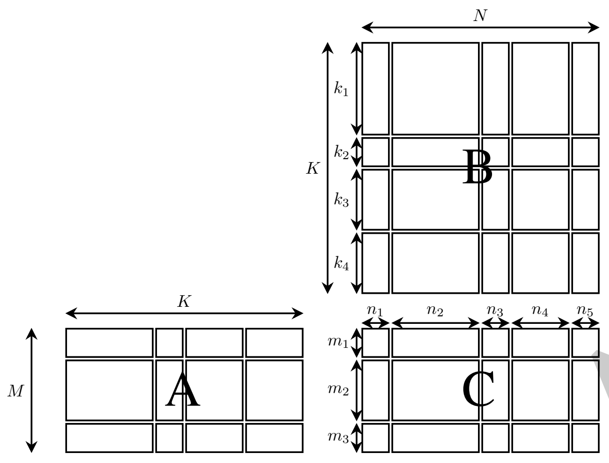

Fig. 1. Example of a Irregular Tiled GEMM operation

V. Runtime and Algorithm Optimizations

A. Optimized Matrix Multiply Algorithm

The ABCD operation is implemented in *TiledArray* as a distributed-memory matrix multiplication (GEMM) [23] on matrices with irregular tiles. GEMM computes $C' = A \times B + C$, where A is a matrix of size $M \times K$, B is a matrix of size $K \times N$, and C is a matrix of size $M \times N$. A, B, and C are tiled – they are divided into submatrices such that $A_{ij}, 1 \leq i \leq m, 1 \leq j \leq k$ are submatrices of A of size $M_i \times K_j, \sum_{i=1}^{m} M_i = M, \sum_{j=1}^{k} K_j = K$. We define similarly B_{ij} and C_{ij} as the tiles of B and C, of size $K_i \times N_j$ and $M_i \times N_j$ respectively. (M_i, N_j, K_k) defines a cartesian tiling of A, B, C such that the tiles remain compatible for the GEMM operation: $C'_{ij} = C_{ij} + \sum_{l=1}^{k} A_{il} \times B_{lj}, 1 \leq i \leq m, 1 \leq j \leq n$. Figure 1 illustrates this irregular tiling for 3 matrices.

As described in Section II, the shape of the input matrices is a consequence of the specific problem context: in typical runs, $K = N >> M$. As a consequence, the traditional version of SUMMA that rotates A and B over the blocks of C in a coordinated manner becomes communication intensive: an approach that moves the data of A and C above the location of B is much more efficient. Another challenge is that the chemistry context determines the tilings of A, B, and C. Irregularity of the tiling creates additional scheduling challenges for the runtime system, as each task has a different load.

The SUMMA operation is implemented over the PTG Domain Specific Language of *PaRSEC*. In PTG, there are multiple levels of parallelism: between nodes, tasks are bound to data, and will execute where specific data are located. This binding is static, and decided by the developer. Inside a node, the distribution of tasks between the cores and the accelerators is decided dynamically by the runtime. Multiple strategies are operating simultaneously: when tasks can be scheduled on accelerators (like the GEMM update operation, for which we provided a MKL-based for the CPU and a cuBLAS-based for the GPUs), the scheduler computes the current load of the CPUs and GPUs, and distributes ready-tasks based on the corresponding number of floating point operations required, the computing capability of the device, and the location of the data. Once some data starts to be modified by an accelerator, it remains hosted by the accelerator until the next update requires a CPU execution, binding all subsequent local GEMM operations on this GPU. Tasks assigned to CPUs, on the other hand, may be executed by any computing thread bound to any core, using a job-stealing approach. The dynamic schedulers of *PaRSEC* aim at optimizing cache reuse by sorting tasks in local queues as a function of the recent use of data by tasks, and job stealing follows a hierarchical strategy that maps the hardware memory hierarchy. However, these constraints are only heuristics, and tasks assigned to CPUs may be executed by any core.

The algorithm used for the integrated software aims at minimizing communications and exposing the highest degree of parallelism. As the order in which the updates of each tile of C does not impact the quality of the result for the targeted application, we explore the data distribution of A, B, and C at initialization time, and build an execution plan that defines where each $GEMM$ kernel is going to be executed, and in what relative order.

```
Input       : $A$, matrix of $M \times K$, tiled on $M_m, K_k$
Input       : $B$, matrix of $K \times N$, tiled on $K_k, N_n$
Input/Output: $C$, matrix of $M \times N$, tiled on $M_m, N_n$
Parallel for $1 \leq i \leq m$ {
    Parallel for $1 \leq j \leq n$ {
        Let $Chains_{i,j}$ be a partition of $[1, \ldots, k]$;
        Parallel for $c \in Chains_{i,j}$ {
            Let $C_c$ be an empty matrix of size $m_i \times n_j$;
            For each $l \in c$ {
                Compute $C_c = C_c + A_{il} \times B_{lj}$
            }
        }
        Reduce the sum of $C_c, c \in Chains_{i,j}$ into $C_{ij}$
    }
}
```

Algorithm 1: Tiled Matrix Multiply Algorithm

A generic tiled matrix multiply algorithm is presented in

Algorithm 1. It consists in a collection of parallel chains of tasks that compute a partial sum of $GEMM$ updates, followed by a chain reduction that sum these partial results to the corresponding tile of C. The execution plan defines these chains, and the order of the reduction. For *MPQC*, as noted above, B is significantly larger than A and C. Thus, the execution plan considers the location of each contribution of tiles of B, and chains them together, to execute on the same process. The partial result is then sent to the next rank hosting any contribution to that sum, completing the reduction at the location of the final target tile in C.

As each $GEMM$ task defines the data it requires, the runtime automatically broadcasts the tiles of A that are required to execute the different operations. We did not insert control flow between the different independent chains, so executions can degenerate by requiring all tiles of A on all ranks, effectively triggering a replication of the matrix A on all ranks. However, since A is $M/K << 1$ only a small fraction of B, this memory overhead is typically negligible for *MPQC* (*e.g.*, in our experimental section, some application runs require A and C to be 5GB, while B is 260GB). The data distributions in *MPQC* for A, B, and C is a block distribution (each matrix is seen as a single contiguous memory block, and each node gets the same number of contiguous tiles). The GEMM update tasks are distributed following the distribution of B, to reduce communications.

B. Scheduling Out-of-GPU-Memory Operations

Another challenge raised by the *MPQC* application comes from the size of the B matrix. Even when distributed, the B matrix might not fit in the accumulated accelerators memory: it is necessary for the runtime to schedule tasks and data movements on the GPUs in a way that tolerates this fact and that aims at high data reuse to increase efficiency.

PaRSEC GPU scheduling is an opportunistic process: when a task can be executed on the GPU, a first load balancing algorithm decides on which resource it should be sent (the cores, or one of the available GPUs). A model-based function provides the number of floating point operations issued by the task to schedule, and hardware capability read at run time is used to balance the load between the different computing resources. If the task selected is targeted toward a GPU, and this GPU is not managed by *PaRSEC* at the time, the calling thread enters GPU management, and will remain in this mode until all tasks sent to this GPU are completed. As long as the GPU manager is in use, all tasks sent to that GPU from any thread are delegated to this GPU manager.

The GPU manager uses multiple CUDA streams: two are dedicated to data movement (one for movements from host to device, the other for movements from device to host), and there is a variable number of streams dedicated to the execution of tasks. An internal finite state machine is used to track the completion of each operation on each stream and progress to the next part of the operation: first, data that needs to be pushed down to the device is scheduled on the input stream, then the task execution is scheduled on one of the execution stream, last, if needed the output data can be copied back in main memory using the output stream.

To cope with out-of-GPU-memory operations, it is necessary to evict data that was copied onto the GPU back into the main memory. This is done during the first phase of a task execution on the GPU: while looking to pull data from main memory to the GPU, the GPU manager can detect that no memory is available on the GPU. It then iterates over two Least Recently Used queues: the read LRU and the modified LRU. In the read LRU are stored (in order of usage) all the data elements that reside on the GPU, but are not required by scheduled tasks neither in read or write mode, and that have not been modified. In the write LRU are stored (in order of usage) the data elements that are not in use, but have been modified by the execution of previous tasks. *PaRSEC* prioritizes modified data to read-only data: if, a data is available in the read-LRU, it will be evicted first. Only if the read-LRU is empty will a write-LRU data be evicted.

This can lead to tasks sent to be executed on the GPU, but whose data are currently being evicted, or for which no data can be allocated because too many tasks execute on the GPU. These tasks will wait first-in first-out that enough tasks complete (or their data transfer completes) to acquire the memory necessary to their input data and initiate the data movement required for their execution.

VI. Related Work

a) Runtimes: With the increase in hierarchy and complexity of the underlying hardware, maintaining a potential for high performance while abstracting the hardware to a simpler expression became critical. The literature is not short of proposals addressing this problem, including many evolutionary solutions that seek to extend the capabilities of current message passing paradigms with intra-node features (MPI+X). A different, more revolutionary, solution explores dataflow task-based runtime systems as a substitute to both local and distributed data dependencies management. The ideas behind this are similar to the concepts put forward in workflow, parallelizing an algorithm over a heterogeneous set of distributed resources, by dividing it in sets of dependent tasks and organizing the data transfers to maximize the occupancy of most resources. But the scale, in terms of number and heterogeneity of resources, and the duration of the potential tasks set the new programming model well apart from workflows. Numerous efforts to provide such an abstraction via a fine-grain task-based dataflow programming exists, adding to those that have transitioned from a grid-based workflow toward a task-based environment. Some of the recent task-based runtimes like Legion [24], *StarPU* [25], QUeuing And Runtime for Kernels (QUARK) [26], HPX [27], Open Community Runtime (OCR) [28], *OmpSS* [29], SuperMatrix [30], and *PaRSEC* [1] abstract the available resources to isolate application developers from the underlying hardware complexity and simplify the process of writing massively parallel scientific applications.

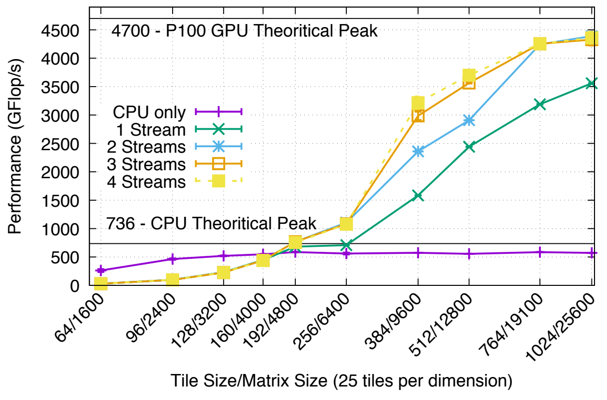

Fig. 2. Performance of double precision GEMM on P100 according to the number of submission streams and the tile size for a fixed number of tiles per dimension (25). CPU (Haswell) performance is according to MKL. The peak of the respective device (CPU, P100are also represented as a dashed line).

The most straightforward approach is to provide a task insertion application programming interface (API) and dynamically build the dependency graph between the developer inserted tasks by tracking the type of usage made with the tasks parameters. QUARK, *OmpSS*, and *StarPU* provide such a task insertion API, supported by different methods to facilitate the scheduling and help with the profiling and debugging. To interact with the runtime, the developer expresses his algorithm as a set of elementary tasks, and inserts the tasks in the runtime. The main advantage of this approach is the ease of porting applications to the task-based runtime, but the easiness comes with drawbacks on distributed environments, as it forces all participating processes to discover the entire set of tasks (in order to identify data movements between processes), before reducing to the set of locally executed tasks, and neighbor tasks that deliver or acquire input or output for the local tasks. This pruning phase limits potential scalability [12]. QUARK has no implicit support for heterogeneous nor distributed architectures. *StarPU* provides support for heterogeneous architectures, and covers distributed execution via the insertion of explicit communication tasks [31], which places the burden of organizing communication back on the application developer and on the communication library. *OmpSS* follows a master-slave model allowing nesting of tasks in individual nodes to relieve the master; however the master-slave model may suffer from scalability issues on large scale distributed systems.

Recent versions of the OpenMP specification [32] introduce the *task* and *depend* clauses which can be employed to express dataflow graphs. OpenMP is widely used and supports homogeneous, shared memory systems, and its *target* extension to support accelerators is quickly gaining traction. A limitation of the OpenMP model is that distributed memory and internode communication needs to be explicitly described and performed with the use of an external communication library.

OCR, still in early development stages, only supports homogeneous architectures and have some nascent capabilities for dealing with distributed environments. Legion describes logical regions of data and uses those regions to express the dataflow and dependencies between tasks, and defers to its underlying runtime, REALM [33], the scheduling of tasks, and data movement across distributed heterogeneous nodes.

Thus far, generic dataflow runtimes have been used to either investigate irregular algorithms on shared memory (occasionally with accelerators), or, alternatively, to deploy dense, regular algorithms on distributed systems. This research provide a tangible base to address sparse irregular applications that have so far been out of reach.

b) Chemistry: Distributed-memory algorithms for coupled-cluster and other many-body electronic structure methods have been in development since late 1980s and are now available in several packages (see Ref. [5] for a recent review of CCSD implementations), most notably in NWChem (a flagship distributed-memory quantum chemistry code), ACESIII, and GAMESS. Unfortunately very little of this capability can be executed on distributed-memory heterogeneous platforms. NWChem has a CUDA-based implementation of perturbative triples correction to CCSD, also known as (T), that has been demonstrated on a GPU-equipped distributed-memory platform and can take advantage of multiple GPUs and multiple CPU cores on each node (however, the CCSD code is CPU only) [34]. Very recently some of us demonstrated a distributed memory implementation of (T) in *MPQC* that can take advantage of multiple GPUs per node [22]. GAMESS has demonstrated a GPU-capable implementation of select terms in the CCSD code on 1 node with 1 GPU [35].

VII. Experimental Results

A. Single Node Experiments

Figure 2 presents a benchmark of the GPU engine designed to help understanding the performance impact of using the cuBLAS library with multiple streams, executed on a recent NVIDIA accelerator (P100). This benchmark consists of a tiled general matrix multiply in double precision. As described in Section V-B, the *PaRSEC* GPU manager users one or more streams to execute kernels. When using a single execution stream, the CUDA engine serializes the kernels calls, thus, the call overhead has little opportunity for overlap, decreasing the SM occupancy and negatively impacting the performance. As the tile size grows, the performance asymptotically reaches the peak, but larger tiles are rare for the target application (*MPQC*). The interesting point is that as soon as the engine uses two execution streams, the cost of calling a kernel mostly overlaps with the execution of a kernel on another stream, and the performance increases up to a sizable percentage of the peak performance. After a given tile size, increasing the number of streams have no measurable benefits for the GEMM operation. In conclusion, in order to extract a reasonable amount of performance (more than 80%) from the use of accelerators via cuBLAS we need tiles of at least 384x384 with at a minimum of 3 submission streams. On the Haswell E5-2650 v3, the MKL library reaches 80% of the theoretical peak for tile size of around 96x96.

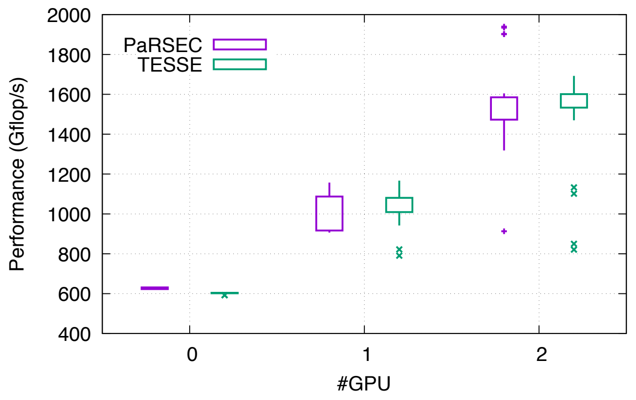

Fig. 3. Comparison of performance between the direct *PaRSEC* driver and the TESSE runtime that integrates *TiledArray*, *MADNESS*, and *PaRSEC*, for a large problem size, on a Haswell E5-2650 v3 with 20 cores and 64GB of RAM, as a function of the number of accelerators

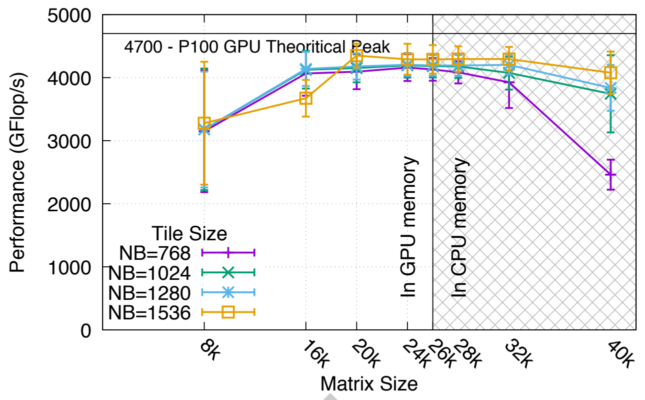

Fig. 4. Problem scaling performance of regular tile size GEMM operation in double precision. Computations limited to NVIDIA GPU (P100). Problem sizes on the right of the 26k cutoff line do not fit in the GPU physical memory and require data movement strategies to minimize/optimize transfers.

To evaluate the overhead due to the integration of the software stack, we implemented the matrix multiply algorithm directly above a *PaRSEC* driver: using *PaRSEC*, we initialize 3 matrices, with shapes similar to the matrices produced by *MPQC*, and call the same algorithm implemented with *PaRSEC* as the one used in the *MPQC* integration. We measure the performance, on an Intel Xeon E5-2650v3 with 20 cores, zero to two NVIDIA P100, and 64GB of RAM, and for the largest problem that fits in this environment. The parameter values $O = 30, n_O = 1, U = 280, n_U = 5$ (see Section II), which corresponds to a matrix multiply of a matrix $A(900 \times 78400)$ by $B(78400 \times 78400)$ onto a matrix $C(900 \times 78400)$, where A, B, and C are tiled with tiles of size 900×3136. Figure 3 shows the distribution of performance for 20 runs, comparing the performance obtained directly with the *PaRSEC* driver and the performance obtained using the entire stack within *MPQC*, as a function of the number of accelerators used. The Tuckey box plots show the minimal value, the 1st and 3rd quartiles, and the maximal values of the distributions, with measurements further than twice the interquartile range shown as additional points.

The software stack overhead is measurable with statistical significance only for the CPU-only run where it remains well below 1%. For the accelerator-supported runs, the variance increases significantly, but for a different reason: the dynamic scheduling in *PaRSEC* introduces non-determinism in the order of execution of kernels between executions. This variance remains under 5%, but this is high enough to make the overhead for the full integration non measurable.

Figure 4 presents the performance of *PaRSEC* executing a tiled matrix multiply operation in double precision. This experiment is run on a single Haswell (E5-2650v3 20 cores) node with one NVIDIA Tesla P100 with 16GB of onboard memory. The cutoff size beyond which the problem does not fit in the GPU memory is depicted as a vertical line around 26k. Below that point, all three matrices will be transferred once to the GPU for the duration of the operation. Beyond that point, the engine is moving data back and forth on the GPUs as tasks become ready.

The impact of the data transfers is also visible in Figures 4, most notably after the cutoff point. As the matrix size grows, the need for data reuse increases, adding to the pressure on the heuristic of selecting the best candidate for replacement (see Section V-B). An optimal reuse would guarantee a minimal need for multiple back-and-forth between different memories (CPU and GPU) for the same data, but optimal scheduling decisions are intractable. From these results it is clear that *PaRSEC* data transfer decisions are sub-optimal, and result in a significant decrease in accelerator occupancy and performance. Addressing this problem is outside the scope of this study, and remains open for future work.

B. Distributed Application Experiments

Performance analysis of the *PaRSEC*-based matrix multiplication algorithm on a distributed memory system utilized the 173-node Dell *NewRiver* cluster at the Virginia Tech Advanced Research Computing. Each nodes in the GPU-enabled subset of NewRiver is equipped with 2 14-core Intel Xeon E5-2680v4 CPUs (1075 GFlop/s peak in double precision) with 512 GB RAM and two NVIDIA P100 GPUs (total of 9400 GFlop/s peak in double precision), each with 12 GB HBM2 memory. Both *TiledArray*, *MADNESS*, and *PaRSEC* on NewRiver were compiled with GCC 8.1.0 and CUDA 9.0 with Intel MPI and MKL 2018 libraries.

Tensor contraction in Eq. (1) was implemented in *TiledArray* using two variants of distributed-memory SUMMA, one the native task-based implementation of SUMMA previously described elsewhere [21] and extended to issue tasks to a single CUDA-enabled GPU (no useful computational work is being done by the CPU in this version), and second the *PaRSEC*-based implementation described in Section V-A. Strong scaling of the two variants of SUMMA was studied for a representative model problem: a cluster of 12 water molecules in aug-cc-pVDZ basis. This problem corresponds to occupied and unoccupied ranges of size $O = 60$ and $U = 432$,

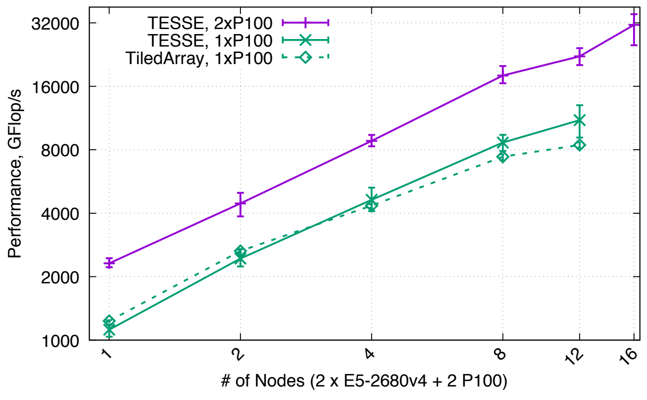

Fig. 5. Strong-scaling performance of the ABCD term in the coupled-cluster doubles equation for $(H_2O)_{12}$ in aug-cc-pVDZ basis set.

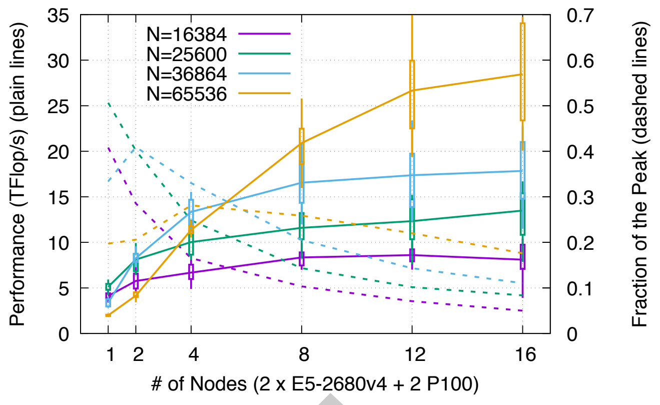

Fig. 6. Strong-scaling performance of SUMMA for a square matrix multiplication as a function of problem size. Constant tile size 1024 is used throughout.

respectively, each split into $n_O = 2$ and $n_U = 12$ tiles of varying sizes.

Excellent strong scaling was demonstrated by the *PaRSEC*-based SUMMA (Figure 5), both when using 1 GPU and 2 GPUs per node. A speedup of $\times 12.8$ was observed when the node count increased from 1 to 16, which translates into 80% parallel efficiency. Good strong scaling was attained by the native SUMMA implementation in TiledArray: a speedup of 6.8 was observed when the node count increased from 1 to 12, which translates into 57% parallel efficiency. However the current default implementation of SUMMA in *TiledArray* does not permit to efficiently utilize more than 1 GPU per MPI rank (with 2 GPUs per node a speedup of only 3.5 was observed when the node count increased from 1 to 16 nodes). Thus the new *PaRSEC*-based SUMMA implementation is a huge improvement over the default *TiledArray* implementation, and on 16 nodes it allows to reduce time to solution by more than a factor of 3.5, due to its efficient parallel scalability and the ability to efficiently utilize multiple P100 GPUs *as well as* the CPU cores.

The excellent strong scalability notwithstanding, there is still room for improvement. Specifically, the absolute performance of 31.3 TFlop/s on 16 nodes corresponds to approximately 20% of the peak hardware performance. As the data in Section VII-A suggests, *PaRSEC*-based GEMM implementation is perfectly capable of reaching high percentage of peak. To understand the origin of lower performance of the distributed ABCD benchmark we used the same *PaRSEC*-based SUMMA implementation to evaluate product of *square* matrices of various sizes. The hypothesis in this experiment was that the performance degradation could be traced to the "stationary" matrix in SUMMA (i.e., matrix B) no longer fitting into the high-bandwidth memory on the GPU. Indeed, as the data in Figure 6 suggests, for smaller problem sizes efficiency decreases monotonically with the number of nodes, whereas for the largest problem sizes the efficiency actually *increases* with the number of nodes when the number of nodes is small; one square matrix of size $N = 65,536$ in double precision requires $\sim$34GB of space, which greatly exceeds 12GB of high-bandwidth memory on a single card. Also note than on 1 node performance reaches peak for $N = 25,600$ (5.2GB) at 5.3 TFlop/s (or $> 50\%$ of peak) and then drops to ~ 2.1 TFlop/s for the largest problem size, which is similar to the ABCD benchmark performance on 1 node. This suggests that incorporating resource awareness into the *PaRSEC*-based SUMMA implementation should allow to improve performance significantly.

VIII. Conclusion

In this paper, we present a new approach of increasing applications efficiency on heterogeneous environments by the means of an integrated software stack, supported by a task-based runtime, *PaRSEC*. We depicted TESSE, the resulting software infrastructure, as well as algorithmic modification and runtime alterations necessary to improve the performance of the target application *MPQC*, a quantum chemistry application. The results show, for different matrix multiplication operations (with both square and non-square matrices and regular and irregular tiling), unprecedented levels of performance for tensor product applications on a distributed heterogeneous environment, with a sustained efficiency and scalability significantly higher that the state-of-the-art. As such it validates TESSE's application-driven design to create a general-purpose software framework that attacks the twin challenges of programmer productivity and portable performance for advanced scientific applications on massively-parallel, hybrid, many-core systems. This study also highlights the need to develop specialized DSL to facilitate computational scientists interaction with new programming concepts, and emphasize the capabilities of the underlying runtime, PaRSEC, to efficiently handle intricate and dynamic workloads on complex architectures without making compromises regarding the performance of the resulting applications. In same time this study exposed technical issues with the current implementation that limit the exposed parallelism and have a negative impact on performance, limitations that will be addressed in the near future.

Acknowledgment

This research was supported partly by NSF awards #1450300, #1450344 and #1450262, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We also acknowledge Advanced Research Computing at Virginia Tech (www.arc.vt.edu) for providing computational resources and technical support that have contributed to the results reported within this paper.

References

[1] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Comp in Sc. and Eng.*, vol. 99, p. 1, 2013.

[2] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," *Proceedings of WOLFHPC 2014: 4th Intl Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pp. 21–30, 2014.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, and et al., "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

[4] I. Shavitt and R. Bartlett, *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*, ser. Cambridge Molecular Science. Cambridge University Press, 2009.

[5] C. Peng, J. A. Calvin, F. Pavošević, J. Zhang, and E. F. Valeev, "Massively Parallel Implementation of Explicitly Correlated Coupled-Cluster Singles and Doubles Using TiledArray Framework," *J. Phys. Chem. A*, vol. 120, no. 51, pp. 10 231–10 244, Dec. 2016.

[6] C. Riplinger, P. Pinski, U. Becker, E. F. Valeev, and F. Neese, "Sparse maps—A systematic infrastructure for reduced-scaling electronic structure methods. II. Linear scaling domain based pair natural orbital coupled cluster theory," *J Chem Phys*, vol. 144, no. 2, Jan. 2016.

[7] R. A. Van De Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[8] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication," in *2013 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, Jan. 2013.

[9] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997.

[10] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra, "PaRSEC in Practice: Optimizing a Legacy Chemistry Application through Distributed Task-Based Execution," in *2015 IEEE International Conference on Cluster Computing*, Sept 2015, pp. 304–313.

[11] H. Jagode, A. Danalis, G. Bosilca, and J. Dongarra, *Accelerating NWChem Coupled Cluster Through Dataflow-Based Execution*. Springer International Publishing, 2016, pp. 366–376.

[12] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime," in *Proceedings of ScalA'17*, 2017, pp. 6:1–6:8.

[13] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, and J. Dongarra, *From Serial Loops to Parallel Execution on Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 246–257.

[14] R. J. Harrison, G. I. Fann, T. Yanai, Z. Gan, and G. Beylkin, "Multiresolution quantum chemistry: Basic theory and initial applications," *J. Chem. Phys.*, vol. 121, no. 23, pp. 11 587–11 598, Dec. 2004.

[15] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M. Y. Ou, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Á. Vázquez-Mayagoitia, N. Vence, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S123–S142, 2016.

[16] J. C. Pei, G. I. Fann, R. J. Harrison, W. Nazarewicz, J. Hill, D. Galindo, and J. Jia, "Coordinate-Space Hartree-Fock-Bogoliubov for Superfluid Fermi Systems in Large Boxes," *J. Phys. Conf. Ser.*, vol. 402, 2012.

[17] M. G. Reuter, J. C. Hill, and R. J. Harrison, "Solving PDEs in irregular geometries with multiresolution methods I: Embedded Dirichlet boundary conditions," *Comput. Phys. Commun.*, vol. 183, no. 1, 2012.

[18] N. Vence, R. Harrison, and P. Krstić, "Attosecond electron dynamics: A multiresolution approach," *Phys. Rev. A*, vol. 85, no. 3, Mar. 2012.

[19] F. A. Bischoff and E. F. Valeev, "Computing molecular correlation energies with guaranteed precision," *J. Chem. Phys.*, vol. 139, no. 11, p. 114106, 2013.

[20] J. Calvin and E. Valeev, "TiledArray: A massively-parallel, block-sparse tensor framework written in C++," https://github.com/ValeevGroup/tiledarray, 2018.

[21] J. A. Calvin, C. A. Lewis, and E. F. Valeev, "Scalable task-based algorithm for multiplication of block-rank-sparse matrices," in *IA3 '15*. ACM Press, 2015, pp. 1–8.

[22] C. Peng, J. Calvin, and E. F. Valeev, "Coupled-cluster singles, doubles and perturbative triples with density fitting approximation for massively parallel heterogeneous platforms," submitted to *Int. J. Quant. Chem.*, 2018.

[23] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38 – 53, 2009.

[24] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[25] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Conc. Comp. Pract. Exper.*, vol. 23, pp. 187–198, 2011.

[26] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[27] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX execution model to stencil-based problems," *Computer Science - Research and Development*, vol. 28, no. 2-3, pp. 253–261, 2013.

[28] J. Dokulil, M. Sandrieser, and S. Benkner, "Implementing the Open Community Runtime for Shared-Memory and Distributed-Memory Systems," *Proceedings of PDP 2016*, pp. 364–368, 2016.

[29] A. Duran, R. Ferrer, E. Ayguade, R. M. Badia, and J. Labarta, "A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks," *Intl. Journal of Parallel Programming*, vol. 37, no. 3, pp. 292–305, 2009.

[30] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *Proc. of SPAA '07*, 2007, pp. 116–125.

[31] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Harnessing Supercomputers with a Sequential Task-based Runtime System," vol. 13, no. 9, pp. 1–14, 2014.

[32] "OpenMP 4.0 Complete Specifications," 2013. [Online]. Available: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[33] S. J. Treichler, "Realm: Performance portability through composable asynchrony," Ph.D. dissertation, Stanford University, 2014.

[34] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, "Optimizing tensor contraction expressions for hybrid CPU-GPU execution," *Clust. Comput*, vol. 16, no. 1, pp. 131–155, 2013.

[35] A. Asadchev and M. S. Gordon, "Fast and Flexible Coupled Cluster Implementation," *J. Chem. Theory Comput.*, vol. 9, no. 8, pp. 3385–3392, Jul. 2013.