One-Sided Interface for Matrix Operations using MPI-3 RMA: A Case Study with Elemental

Sayan Ghosh*, Jeff R. Hammond†, Antonio J. Peña‡, Pavan Balaji§, Assefaw H. Gebremedhin*, Barbara Chapman ¶

* School of Electrical Engineering and Computer Science,
Washington State University, Pullman, WA, USA {sghosh1, assefaw}@eecs.wsu.edu
† Parallel Computing Lab, Intel Corp., Portland, OR, USA jeff.r.hammond@intel.com
‡ Barcelona Supercomputing Center, Barcelona, Spain antonio.pena@bsc.es
§ Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA balaji@mcs.anl.gov
¶ Institute for Advanced Computational Science,
Stony Brook University, Stony Brook, NY, USA Barbara.Chapman@stonybrook.edu

***Abstract*—A one-sided programming model separates communication from synchronization, and is the driving principle behind partitioned global address space (PGAS) libraries such as Global Arrays (GA) and SHMEM. PGAS models expose a rich set of functionality that a developer needs in order to implement mathematical algorithms that require frequent multidimensional array accesses. However, use of existing PGAS libraries in application codes often requires significant development effort in order to fully exploit these programming models. On the other hand, a vast majority of scientific codes use MPI either directly or indirectly via third-party scientific computation libraries, and need features to support application-specific communication requirements (e.g., asynchronous update of distributed sparse matrices, commonly arising in machine learning workloads). For such codes it is often impractical to completely shift programming models in favor of special one-sided communication middleware. Instead, an elegant and productive solution is to exploit the one-sided functionality already offered by MPI-3 RMA (Remote Memory Access). We designed a general one-sided interface using the MPI-3 passive RMA model for remote matrix operations in the linear algebra library Elemental; we call the interface we designed *RMAInterface*. Elemental is an open source library for distributed-memory dense and sparse linear algebra and optimization. We employ *RMAInterface* to construct a Global Arrays-like API and demonstrate its performance scalability and competitivity with that of the existing GA (with ARMCI-MPI) for a quantum chemistry application.**

***Keywords*-Distributed-memory linear algebra, MPI-3, RMA, one-sided communication, Global Arrays, PGAS**

I. Introduction

Many scientific applications are a mixture of regular and irregular computations. For example, in quantum chemistry, methods such as density-function theory (DFT) are composed of a highly irregular step of forming the Fock matrix, which requires dynamic load balancing and unstructured communication in order to utilize all the processing elements, followed by parallel dense linear algebra to diagonalize this matrix. Other application domains have similar patterns of combining domain-specific matrix-formatting steps with standard linear algebra procedures. It is critical to allow application developers to combine domain-specific code with the best available dense linear algebra libraries without compromising performance by restricting the data layouts or communication patterns they can use in the domain-specific parts of their code.

Historically, the Global Arrays library (GA) [13] has met this need in quantum chemistry applications by providing a library that supports dense array data structures, a set of one-sided communication primitives that support arbitrary subarray access patterns, the necessary features for dynamic load-balancing, and an interface to parallel dense linear algebra capability from ScaLAPACK [5] (and, more recently, ELPA [11]). Thus, the domain scientist is able to write an efficient Fock matrix formation code by reading and updating distributed arrays, then calling the dense eigensolver, Cholesky, or other procedures from ScaLAPACK, without having to know anything about the ScaLAPACK interface.

An alternative approach to using GA, which is implemented in linear algebra libraries such as PETSc [1], PLAPACK [21], and Elemental [14], involves queuing up updates to a distributed array locally, then completing them with a collective operation. [1] This has the desirable property of being highly portable, since it can be implemented by using two-sided messaging, such as MPI Send-Recv. However, the distributed data structure cannot be touched until the collective step happens, and there is no opportunity for overlapping communication and computation, because unlike one-sided operations in GA, unmatched messages cannot complete asynchronously.[2]

With the release of the MPI-3 standard [12], the communication primitives required to implement GA have become widely available in both open-source and proprietary implementations of MPI. Furthermore, new distributed dense linear libraries such as Elemental and DPLASMA [3] have emerged as modern alternatives to ScaLAPACK. These changes motivate a fresh investigation of the interplay between irregular computations and dense linear algebra computations. Can we

[1] See http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Mat/MatSetValues.html and http://libelemental.org/documentation/0.81/core/axpy_interface.html for details on these interfaces.

[2] The MPI progress rules for `Send` apply only when a matching `Recv` has been posted. In the case here, matching Recv can be posted only after information about the local queue has been communicated with the destination processes.

achieve performance similar to Global Arrays and ScaLAPACK while providing more flexible interfaces, support for different matrix distributions, and greater portability through MPI-3?

GA provides a convenient data access interface that uses high-level array indices; it also provides a rich lower-level interface for managing data distribution, exploiting locality, and managing communication. GA is built on top of Aggregate Remote Memory Copy Interface (ARMCI), which is a low-level one-sided communication runtime system. ARMCI has implementations for several vendor-specific network conduits such as uGNI and LAPI. ARMCI is compared with MPI by Dinan et al. [6]. Elemental has a similar interface for accessing portions of matrices distributed in memory, called the AXPY interface (named after the BLAS *axpy* routine, which performs vector-vector multiplication and addition, i.e., $a \times x + y$, where a, x, and y are vectors). This interface provides a mechanism by which individual processes can independently submit local submatrices that will be automatically redistributed and added to the global distributed matrix. The interface also allows for the reverse: each process may asynchronously request an arbitrary subset of the global matrix. Both of these functionalities are effected by a single AXPY routine.

Unlike GA, however, one needs collective synchronization before the global matrix or the local matrix—which is submitted to the global matrix or where a subset of the global matrix would eventually reside—can be reused or accessed. (A way to circumvent this for the local matrix would be to allocate a new matrix for every AXPY call, but the approach clearly is memory expensive.) This requirement is an artifact of the underlying implementation of the API using MPI point-to-point routines and in certain applications may require essentially replicating the matrix on every process. We note that the AXPY interface was developed at a time when the MPI RMA model was restrictive in its offerings. Prior to the recent release of MPI-3, MPI RMA lacked important atomic operations (e.g., fetch-and-add and compare-and-swap), had an inconvenient synchronization model (including lack of separation of local and remote completion), and had a memory model (especially the notion of public and private windows) that made MPI one-sided communication routines undesirable for one-sided operations in applications. Bonachea and Duell [2] discuss the limitations of MPI-2 RMA as compared to contemporary PGAS models.

Therefore, we redesigned the Elemental AXPY interface in a way that would reduce the amount of collective synchronization and bring the interface semantically closer to GA. RMA is a natural communication model for scientific applications involving multidimensional array accesses, and some of these applications have a need for robust Dense Linear Algebra (DLA). We have created a "union" of RMA and DLA, much as GA did, but we went in the opposite direction by building RMA into DLA, rather than building DLA into RMA.

The remainder of the paper is organized as follows. In Section II, we provide some background information and motivation for our work on Elemental. In Section III, we identify some limitations of the existing asynchronous matrix update API of Elemental, and, propose a new set of API which aims to improve the performance of the existing approach significantly. In Section IV, we discuss our implementation of the new API using MPI-3 RMA, and, establish the need for a distributed arrays interface to increase productivity. In Section V, we present performance evaluations using microbenchmarks and a quantum chemistry application. In Section VI, we draw conclusions and briefly discuss our future research plan.

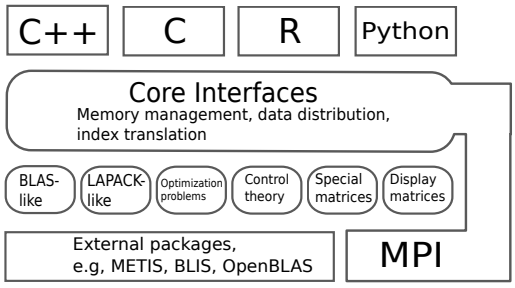

Fig. 1. Overall structure of the Elemental library.

II. Background and Motivation

Elemental is a C++11 library for distributed-memory algorithms for dense/sparse linear algebra and interior-point methods for convex optimization. Similar to PLAPACK [21], Elemental was designed around the idea of building different matrix distributions and providing a simple API for moving a matrix from one such distribution to another throughout a computation. Elemental has a thin abstraction layer on top of the necessary routines from BLAS, LAPACK, and MPI. Figure 1 shows the high-level organization of Elemental.

A. Data Distribution

One of the requirements for high performance dense matrix computations is scalability. The way in which the data is distributed (or decomposed) over the memory hierarchy is of fundamental importance to scalability. Data distribution impacts the granularity of the computation, which in turn impacts load balance and scalability. Most distributed-memory dense linear algebra packages differ in the way data distribution is performed. Unlike linear algebra libraries that distribute contiguous blocks of data to processes (e.g., PLAPACK [21] and ScaLAPACK [5]), in Elemental the default matrix distribution is designed to spread the matrix in an element-wise fashion. As depicted in Figure 2, in Elemental, individual elements of a matrix are distributed cyclically in column-major ordering following the 2D process-grid layout. For example, if there are 4 processors in total, then a 2×2 grid is used, and, elements are distributed in a round-robin fashion within each column/row of the 2D process grid.

Elemental offers a number of element-wise distributions over the process grid and provides a convenient mechanism

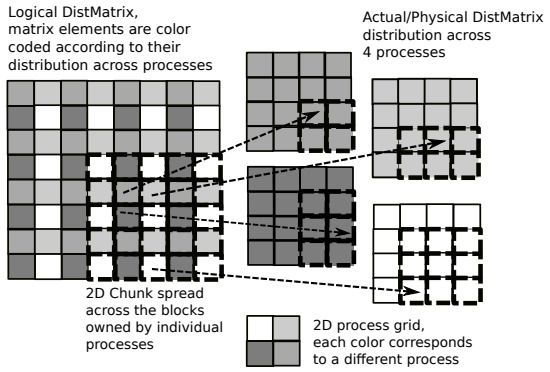

Fig. 2. Elemental element-wise cyclic distribution (MC×MR) of an 8×8 matrix on a 2×2 process grid (4 processes). Dark borders indicate local/ physical chunks corresponding to a global chunk.

for performing basic matrix manipulations. The `Matrix<T>` class builds a 2D matrix owned by only the process calling it, and `DistMatrix<T, U, V>` class is the distributed-memory variant of the former. Here, T stands for template substitution for datatypes, including complex types, and U and V signify the distribution pattern on each dimension. The default matrix distribution is known as MC×MR (matrix column by matrix row). This distributes the elements in the first dimension in a round-robin fashion over each column of the process grid, and in the second dimension in a similar way over each row of the 2D process grid. The logical 2D process grid is actually composed of different MPI communicators (for instance, there are communicators for processes forming the row of the process grid and that of the column) because of the need for MPI collective communication to distribute elements according to the specified distribution. Figure 2 shows the MC×MR logical and physical distribution of a matrix. Elemental has around 10 such distributions (distributions can also be paired). Throughout our work, we have used only the MC×MR distribution because it leads to the best scalability for a variety of cases.

B. Elemental AXPY Interface

We started the work by investigating the performance and suitability of the Elemental AXPY interface, which offers functionalities to perform operations on globally distributed matrices. The AXPY interface is implemented by using MPI point-to-point communication routines. This interface has the look and feel (in terms of the API) of Global Arrays, which is extensively used in the NWChem computation chemistry package [20]. The AXPY interface has only three routines: ATTACH, AXPY, and DETACH; the individual functionalities are explained below.

1) ATTACH (collective) – Performs vector resizing, buffer allocation.
2) AXPY (point-to-point) – Sends or receives data to/from the globally distributed matrix based on the direction parameter specified by the user (LOCAL_TO_GLOBAL or GLOBAL_TO_LOCAL). This operation is analogous to the ARMCI scaled accumulate – *dst += scale * src* (where *dst* and *src* are the origin buffer or the target buffer, depending on the direction).
3) DETACH (collective) – Finishes all outstanding communication and tracks progress.

We identified several issues with the existing AXPY interface of Elemental that limit its capabilities considerably (these are discussed in the next section in detail). Hence, we redesigned the existing AXPY interface and created an entirely new interface, which we have called *RMAInterface*, with the aim of improving the asynchrony of remote operations and thereby enhancing performance significantly.

III. BEYOND THE ELEMENTAL AXPY INTERFACE

In this section, we offer general guidelines on designing an interface for asynchronous matrix operations. We begin by enumerating some of the notable issues with the existing Elemental AXPY interface:

- *Asynchrony*: The existing interface does not allow overlapping of operations. When DETACH returns, all communication to/from the distributed matrix completes. This enforces a bulk synchronous model.
- *Overallocation*: Although one could issue multiple AXPY calls within an ATTACH-DETACH epoch (note that these are point-to-point nonblocking operations), different local matrices would have to be allocated as well, because local/remote completion of operations is not guaranteed until DETACH is called.
- *Restrictive synchronization*: In DETACH, every process must exchange "end-of-message" (EOM) messages to mark communication end; also, the ATTACH-DETACH pair of calls marks access epochs.
- *Expression*: The put/get/accumulate operations are all expressed through a single function – AXPY, where one must specify *alpha* (scale factor) and *direction* (local to global or vice versa) to select the particular operation.

In Section III-A, we discuss a strategy to improve the performance of existing Elemental AXPY interface and, in Section III-B we discuss the design of *RMAInterface*.

A. Nonblocking Consensus

Since AXPY operations are point-to-point and nonblocking, the communication may not even begin until DETACH is called. Apart from ensuring MPI progress, DETACH also needs a mechanism to mark the end of current data communication. This is because there is no way of assessing in advance the number of messages to be received. Hence, each process sends an EOM message to every other process to complete the ATTACH-DETACH epoch.

This mechanism poses a nonnegligible overhead. For instance, in a simple test program, in which each MPI process updates different locations of the distributed matrix, we found that around 80% of the total time was spent on DETACH.

We can improve the performance of DETACH, however, if there is a way to improve the synchronization

logic. Fortunately, MPI-3 introduces nonblocking barriers (`MPI_Ibarrier`[3]), which can be used to implement a synchronization scheme for cases when the number of messages to receive is not known in advance. This is facilitated by alternately checking inside a loop for any incoming message (via `MPI_Iprobe`) and testing whether the synchronous sends have completed (via `MPI_Testall`).

In order to improve the performance of the end of data communication, we introduced a consensus mechanism using a nonblocking barrier (`MPI_Ibarrier`) instead of explicitly sending messages to mark the end of communication (during DETACH). This scheme is referred to as *nonblocking consensus* and is inspired by prior research on data exchange protocols[9, Algorithm 2]. By leveraging this protocol we were able to significantly improve performance, but also to save memory by not allocating buffers associated with EOM synchronization. The pseudocode of the enhanced DETACH is listed in Algorithm 1. The HANDLEDATA function handles the data posted during the AXPY call.

Algorithm 1 DETACH: Nonblocking barrier for determining end of communication

```
1: done ← false
2: while not done do
3:    HANDLEDATA()
4:    if nonblocking barrier is active then
5:       done = test barrier for completion
6:    else
7:       if all sends are finished then
8:          activate nonblocking barrier
```

We note that only MPI synchronous sends will work in this particular case, because testing on an MPI request handle associated with a nonblocking synchronous send returns only when a corresponding MPI receive is posted. By completely bypassing the requirement to send EOM packets, we were able to save approximately $3 \times p$ buffer allocations (where p is the total number of processes) and obtain a performance improvement of up to 14x (see Figure 7 in SectionV).

B. *Introducing* RMA Interface

The design of the existing AXPY interface offers little possibility of overlapping computation and communication within the application using it. We therefore designed a new interface, *RMAInterface*, offering one-sided semantics that overcomes the strict synchronization requirements in the existing interface while expressing the required functionality in a more natural manner. Following are the design highlights of this new interface.

- ATTACH-DETACH should be required to be called only once per distributed matrix, instead of every time we need to (re)use the buffers in use by a prior AXPY call.

[3]A non-blocking barrier (`MPI_Ibarrier`) functions in the following way: A request object associated with a barrier evaluates to *true* upon calling `MPI_Test` only when *all* the processes in the communicator have started the nonblocking barrier.

```
// Management
void Attach( DistMatrix<T,MC,MR>& Y );
void Detach();

// Remote Transfer
void Put( Matrix<T>& Z, Int i, Int j );
void Acc( Matrix<T>& Z, Int i, Int j );
void Get( Matrix<T>& Z, Int i, Int j );

// Synchronization
void Flush(      Matrix<T>& Z );
void LocalFlush( Matrix<T>& Z );
```

Fig. 3. Elemental *RMAInterface* API. Locally nonblocking remote transfer APIs begin with an *I, such as `IPut`, `IAcc` and `IGet`.*

- We expose remote operations in the API, such as PUT/GET/ACCUMULATE.
- Instead of bulk synchronization facilitated through DETACH, we introduce operation-wise (noncollective) synchronization functions. We add a number of synchronization API routines (referred to as *Flush* operations), for enforcing local/remote completion.

We will henceforth refer to the Elemental AXPY interface as *AxpyInterface* and the new RMA interface using MPI-3 RMA as *RMAInterface*.

IV. PROPOSED ONE-SIDED APIS

We begin this section by discussing the implementation details of *RMAInterface* using MPI-3 RMA. Then we introduce the Distributed Arrays interface (we refer to it as EL::DA) that we designed on top of *RMAInterface*. Distributed Arrays expose an interface similar to Global Arrays, but with the ability to extend the functionality of GA significantly because of being tightly integrated to Elemental. By having EL::DA similar to GA, we enable the application programmers to easily develop GA-based applications using EL::DA.

A. *RMAInterface Design and Implementation*

1) Design Overview: The basic idea behind *RMAInterface* is to expose a set of one-sided communication and synchronization functions to enable fetching and updating portions of a distributed matrix. A distributed matrix is semantically similar to a global array, but is laid out across processes according to Elemental's `MC`×`MR` distribution. Low-level details such as processes involved in communication and MPI-related types are abstracted away from the API. *RMAInterface* users will only need to specify the local and distributed matrix handles and the (2D) coordinate axes of the distributed matrix, in order to fetch or update a part of the globally distributed matrix.

To familiarize the readers with the RMA interface functionality, we list the definitions of some of its basic functions in Figure 3.

To demonstrate the efficacy of *RMAInterface*, we explore a common parallel programming motif – distributed blocked matrix multiplication. We will show that such a programming task could be easily developed using *RMAInterface*, whereas the existing AXPY interface or any bulk-synchronous mechanism will not work in this case. Figure 4 includes the corresponding

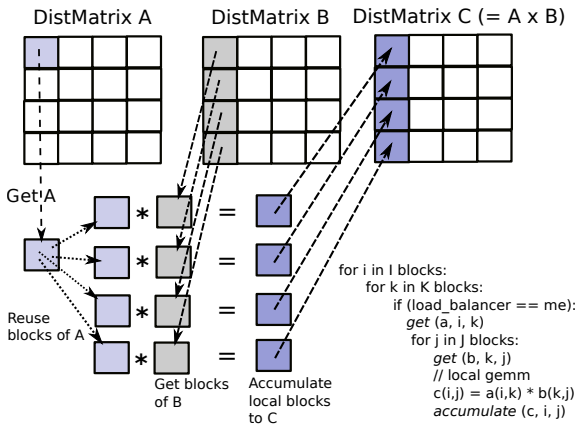

Fig. 4. Logical diagram of a 2D blocked distributed matrix multiplication using *RMAInterface*. Each block of `DistMatrix<T, MC, MR>` A, B and C contains non-contiguous elements. a, b and c are local matrices (i.e, `Matrix<T>`).

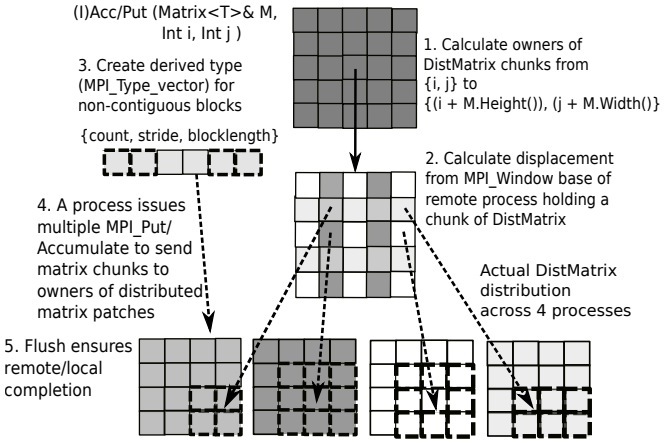

Fig. 5. Steps involved in a put/accumulate operation of *RMAInterface*. An 8×8 distributed matrix (as shown in Figure 2) is updated starting at position $(3, 3)$ by a local 5×5 matrix M. A get operation would show the arrows in the opposite direction. Step 3 (MPI DDT creation) is optional.

pseudocode and a schematic diagram. As shown in the figure, in carrying out a matrix-matrix multiplication, an MPI process acquires a 2D tile of the distributed matrices, performs a local GEMM on the tile it owns, and asynchronously modifies the distributed matrix with the locally updated values. A distributed counter serves as a load balancer by mapping tasks to processes, and ensuring that no two processes access the same tile. Load imbalance occurs if there are more processes than tiles—some of the processes need to wait on a barrier for others to finish. This type of communication pattern, involving asynchronous updates to different portions of a distributed matrix, is not possible in the existing AXPY interface. This is because, AXPY (which simulates put/get/accumulate operation) is not locally complete unless DETACH is called. Since DETACH is a collective operation, it cannot be invoked (inside the innermost loop in Figure 4) if there is no guarantee that every process will obtain a task. Even if this were true, every AXPY call would need to be placed in the middle of ATTACH-DETACH functions in order to ensure local completion as the local matrix c is reused.

2) Implementation Details: RMAInterface is implemented on top of the MPI-3 RMA API. By MPI-3 RMA/*one-sided*, we always mean *passive target communication* in which only the *origin* process—that is, the process initiating RMA calls—is actively involved in data transfers at the user level. This is in contrast with *active target communication* in MPI RMA, wherein both processes, *origin* and *target*, are explicitly involved in the RMA communication at the user level. Implementing PGAS concepts into *RMAInterface* using MPI-3 RMA was a straightforward exercise, as has also been demonstrated in prior research projects of ARMCI-MPI [6] and OSHMPI [8].

ATTACH initializes the *RMAInterface* environment for a particular distributed matrix, starts an MPI RMA epoch (by invoking `MPI_Win_lock_all`), and creates the necessary buffers. We support either `MPI_Win_allocate` or `MPI_Win_create` for creating MPI windows for RMA communication in ATTACH, as per user configuration. While supporting `MPI_Win_create` just requires associating a pointer to a data buffer with the RMA window, most of the core Elemental interfaces had to be modified to support `MPI_Win_allocate`. Like most modern C++ projects, core data structures in Elemental are resized dynamically as needed. This situation had to be prevented when `MPI_Win_allocate` was used, because a predefined amount of memory needs to be available for RMA operations. However, one can dynamically attach memory to an MPI window object via `MPI_Win_create_dynamic`. Although `MPI_Win_create_dynamic` allows exposing memory without needing an explicit remote synchronization, this usually results in low performance. The use of C++ makes the design more expressive, since ATTACH and DETACH can be realized as constructors and destructors of an *RMAInterface* object, respectively. We denote `Put` as locally blocking (when the function returns, the input matrix may be reused), whereas `IPut` is locally nonblocking (the input matrix cannot be reused upon return until a synchronization call is properly performed). The underlying interface uses the input coordinates axes (passed to the put/get/accumulate functions) to determine the target process for the individual elements in the input `Matrix<T>` object. The coordinate axes are also used to calculate the displacement (from the window base) in the MPI window of the determined target process (steps 1 and 2 in Figure 5 depict these operations). Therefore, a single put/get/accummulate call usually results in a number of `MPI_Get/Put/Accumulate` calls to fetch/update data from/to the memory of several remote processes.

Figure 5 shows the various steps involved in a remote

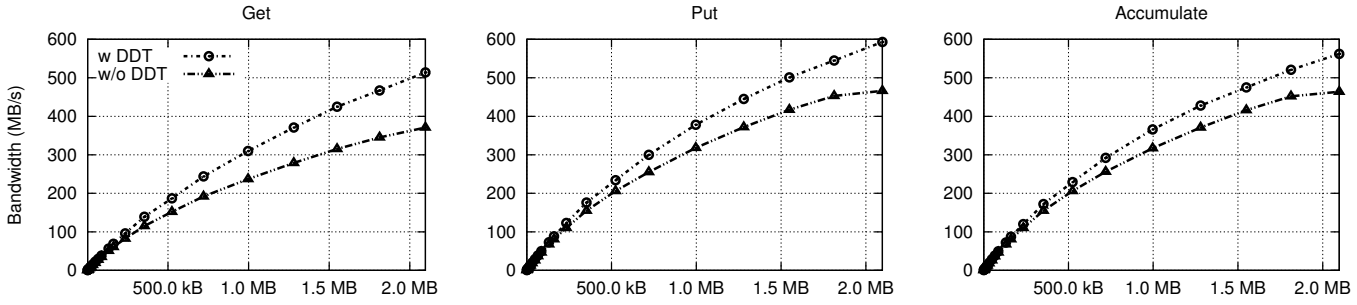

Fig. 6. Bandwidth of put/get/accumulate operations with/without MPI DDT on 16 processes (n=4:ppn=4) on Cori (higher is better). The X-axis shows the size of the data transferred.

operation until completion (effected by calling the appropriate synchronization functions). Since the target data layout (chunks of distributed matrix in individual processes) consists of contiguous blocks separated by fixed strides, we found the usage of MPI derived data types (DDT) to be appropriate in this case. We use `MPI_Type_vector` to designate the target data layout, which helps limit the number of RMA operations.

MPI implementations have been known to suffer from performance penalties when working with DDT, even though a substantial body of research has focused on improving them [19], [15], [4], [7]. Therefore, we made it user configurable for the code path to use MPI DDT in RMA operations, which otherwise would fall back to the version that uses MPI standard data types. To assess the benefit of using MPI DDT, we use a simple test case that performs a number of one-sided put/get/accumulate operations on varied data sizes, from 8 B to 2 MB, with increments of 128 B on 16 processes. Figure 6 shows the comparative bandwidth (in MB/s) of *RMAInterface* put/get/accumulate functions with and without MPI DDT. We observe an improvement of up to 20% in bandwidth on average for the NERSC Cori platform (see Section V for platform details) as a result of using MPI DDT. This improvement is due to the reduction in the number of overall RMA operations. In terms of synchronization, a flush operation ensures remote/local completion of all outstanding operations initiated from (or targeted toward) the current process. `Flush` translates to `MPI_Win_flush_all`, while `LocalFlush` enforces the local completion of all operations (i.e., `LocalFlush` translates to `MPI_Win_flush_local_all`). A relevant distinction between *RMAInterface* and the original *AXPYInterface* is that DETACH is no longer responsible for collective synchronization, as a flush will complete outstanding operations in an asynchronous fashion. In *RMAInterface*, DETACH marks the end of an RMA epoch (by issuing `MPI_Win_unlock_all`) and clears the associated buffers.

B. Distributed Arrays Interface (EL::DA)

We create the Distributed Arrays interface (EL::DA) on top of the Elemental `DistMatrix` and *RMAInterface* interfaces. Both regular and irregular (GA) distributions internally map to Elemental's cell-cyclic data layout. EL::DA supports the most fundamental GA operations, and hence most applications written in GA may be easily ported to EL::DA. We also created a C interface to EL::DA (although Elemental is written in C++11, it offers C interfaces for almost all core modules), which allows it to be used with applications written in C as well. In EL::DA, one-sided functions (such as `NGA_Put` or `NGA_Acc`) are implemented by using *RMAInterface*, whereas for any other collective operation (such as `GA_Add` or `GA_Symmetrize`), the core Elemental API is used. However, supporting GA local access functions (e.g., `NGA_Access`) is not possible in an efficient manner because of the underlying element-cyclic data distribution. For example, consider the case of `NGA_Access`, which returns a pointer to the local portion of the global array. In EL::DA, `NGA_Access` is no longer a local operation; it requires a remote *get* to pull the relevant portions of data from the global array to a local buffer. The reason is that the local portion of an Elemental `DistMatrix` contains elements that are globally noncontiguous (see Figure 2), and hence it cannot be used for performing computations that expect elements to be in their logical order.

`NGA_Release_update` is another case when EL::DA is nonconformant to the original GA. `NGA_Release_update` releases access to a local copy (owned by a particular process) of a GA, in case the local copy was accessed for writing. For cache-coherent machines, `NGA_Release_update` is essentially a *no-op*. Nevertheless, since the local buffer (exposed by `NGA_Access`) was updated, it needs to be sent back to the global array distributed in Elemental's cell-cyclic fashion. Therefore, for EL::DA, `NGA_Release_update` is a remote *put* operation to update the global matrix with local data. Since Elemental offers an unparalleled range of functionality pertaining to linear algebra and optimization, a rich set of the Elemental framework is exposed to the Distributed Arrays interface. In contrast, this is not possible in the existing GA, which offers limited linear algebra functionality.

V. EXPERIMENTAL EVALUATION

Our experiments are motivated by operations frequently arising in quantum chemistry applications. Quantum chemistry simulation of small or large molecular systems requires substantial computation resources and suitable parallel programming models for scalability. In practice, quantum chemistry

codes are not perfectly scalable, because of the significant volume of communication that dominates the total time.

A. Experimental Setup

We used two platforms for our experiments.

1) Argonne's Blues – 310-node cluster with dual-socket Intel® Xeon® E5-2670 processors per node and a QLogic InfiniBand QDR interconnect.
2) NERSC Cori (Phase 1) – 1,630-node Cray XC40 machine with dual-socket Intel® Xeon® E5-2698v3 CPUs per node and the Cray XC series interconnect (Aries).

We used MVAPICH2 (version 2.2.1) on Blues and Cray MPI (version 7.2.5) on Cori (both of them are MPICH derivatives). MVAPICH2 uses hardware for contiguous put/get operations and implements accumulate and strided one-sided operations using software. Following the vendor recommendations, we use the regular mode of Cray MPI, where RMA operations are implemented in software (as opposed to the DMAPP mode [18], which implements one-sided contiguous Put/Get operations in hardware). We found no noticeable difference in the performance of Cray MPI runs with or without DMAPP for our test cases (all of which use accumulate operations). We compare EL::DA with Global Arrays (version 5.4). The native communication conduit of GA is ARMCI, which uses low-level network APIs for point-to-point communication and MPI for collective operations. ARMCI-MPI [6] is a completely rewritten implementation of the original ARMCI using MPI RMA (specifically, it supports both MPI-2 RMA and MPI-3 RMA) for one-sided communication. We used ARMCI-MPI with MPI-3 RMA (referred as GA in the plots) for our evaluations. Also, Cray Aries (Cori's interconnect) cannot perform accumulate or atomic update operations in hardware and must use software. The MPI standard does not mandate RMA to be asynchronous: although there is no need for remote processes to be involved in passive RMA communication, in practice the remote/target process may issue calls to the MPI runtime system to ensure progress in communication.

Asynchronous progress in MPI implementations is typically enabled by using a communication helper thread per process to handle messages from other processes or by utilizing hardware interrupts. Both MVAPICH2 and Cray MPI (in regular mode) implement asynchronous progress using a background thread per process as an optional feature. Because of requiring deployment of as many helper threads as MPI processes, this scheme leads to either processing core oversubscription or devoting half the cores to ensuring MPI progress. Both approaches result in losing a considerable amount of compute power in polling for incoming messages.

To maximize the benefits of the performance potential of leveraging MPI RMA communication instead of its two-sided counterpart, we use Casper [17], a new process-based progress engine for MPI one-sided operations, which aims to alleviate most of the crucial drawbacks of thread-based or hardware interrupt-based communication progress and to favor scalability [16]. Casper decouples the number of helpers devoted to progressing MPI RMA communications from the number of MPI processes—the optimal number of helper (*ghost*) processes is application dependent and currently is specified by the user at launch time. We leverage Casper in our evaluations involving MPI-3 RMA (both Elemental and GA with ARMCI-MPI) and observe significantly improved performance with respect to the cases based on the original thread-based asynchronous progress models.

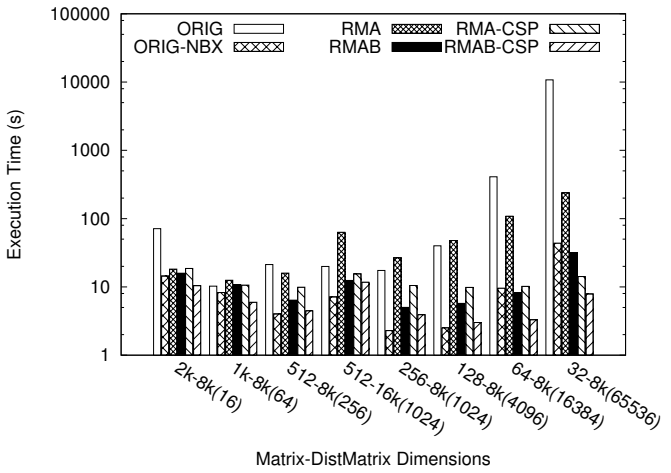

Fig. 7. Hartree-Fock proxy microbenchmark comparing the Elemental AXPY interface versions on 256 processes of Blues. The number inside braces denotes the number of tasks.

B. Microbenchmark Evaluation

We use two microbenchmarks to evaluate the performance of *RMAInterface*. The first microbenchmark loosely simulates a Fock matrix construction used commonly for electronic structure calculation. This microbenchmark is designed to compare *RMAInterface* with the original and improved *Axpy-Interface*. The second microbenchmark is a distributed matrix-matrix multiplication, to compare the performance of EL::DA with that of GA.

1) Hartree-Fock Proxy: This microbenchmark features two phases. In the first phase, each process requests a task, and upon receiving a task it issues a remote *accumulate* operation to different tiles of a 2D matrix distributed in Elemental's element-wise cyclic fashion. In the second phase, each process requests a task again, and upon receiving a task it issues a remote *get* from a different tile of the global matrix to a local matrix. Processes that do not receive a task (because of insufficient tasks) just wait on a barrier. Accesses to different blocks are made possible via a distributed global counter, which ensures that at a time only a single process is accessing a tile of the global array. We compare the performance of six *AxpyInterface* versions in Figure 7. In this microbenchmark, we fix the number of processes to 256 and vary the workload (expressed as number of tasks, each task involving exactly one *accumulate* or a *get* operation). The various versions in Figure 7 are defined as follows.

1) ORIG – Original Elemental AXPY interface.

2) ORIG-NBX – Original Elemental AXPY interface with a nonblocking consensus mechanism to test communication completion in DETACH.
3) RMA – Locally nonblocking API of *RMAInterface* (e.g., IPut/IAcc, see Figure 3).
4) RMAB – Locally blocking API of *RMAInterface* (e.g., Put/Acc, see Figure 3).
5) RMA-CSP – RMA with Casper.
6) RMAB-CSP – RMAB with Casper.

With fewer than 256 tasks (the number of processes), all versions suffer from load imbalance, since some of the processes do useful work while the others wait on a barrier. With a larger number of tasks, the performance of the original *AxpyInterface* (ORIG) suffers because of the large number of messages to mark the end of communication. We improve the performance of these situations by modifying the DETACH to use a nonblocking consensus mechanism (ORIG-NBX) using `MPI_Ibarrier` (see Section III-A). This improved the performance of the original *AxpyInterface* significantly—up to 40x for a large number of tasks and on average by 5x. The AXPY function of the *AxpyInterface* (which is analogous in terms of functionality to `IPut`/`IAcc` in the *RMAInterface*) issues synchronous nonblocking sends to the target process that owns a patch of noncontiguous elements. The target process handles the sends by posting matching receives (when DETACH is called) and places each element at its correct position with respect to the Elemental cyclic distribution. Because of the active participation of the target process, AXPY is able to pack all the (noncontiguous) elements in a single `MPI_Issend`. In the case of *RMAInterface*, since we use MPI-3 passive RMA, we cannot involve the remote process in the communication. Hence, the origin process has to calculate the displacement in the remote process MPI window and issue multiple RMA operations. Therefore, the RMA versions issue many one-sided accumulates (with comparatively small data size as compared with ORIG/ORIG-NBX) over the network, when ORIG/ORIG-NBX could essentially pack elements to limit the number of messages. With an increasing number of tasks, the number of one-sided operations also increases, and RMA/RMAB suffer. In particular, the performance of the RMA versions starts degrading significantly because of the lack of progress of one-sided operations, especially for greater than 4K tasks; and in the worst case it is 5x slower than ORIG-NBX. In contrast, version RMAB is around 10x better on average (for tasks greater than 4K), because it enforces local completion (via `MPI_Win_flush_local`). Hence, we assert the importance of asynchronous progress in RMA communication. The performance of RMAB increases significantly when a number of "ghost" processes are used for asynchronous progress (for Figure 7 we use 4 ghost processes per node). A performance improvement of up to 4x is observed with the use of the *RMAInterface* locally blocking API in combination with the Casper progress engine (referred as RMAB-CSP in Figure 7) as compared with the optimized *AxpyInterface* (i.e., ORIG-NBX).

2) Distributed Matrix-Matrix Multiplication: The matrix-matrix multiplication operation, that is, $C = A \times B$, entails multiplying matrices (A and B) and storing the result on a third matrix (C). Since *AxpyInterface* cannot be used to simulate truly one-sided operations because of its bulk-synchronous nature (see Section IV-A), we compare only RMA approaches for this microbenchmark. Specifically, we compare the relative performance (demonstrated by an average of a million floating point operations per second, or MFLOPS) of a distributed matrix multiplication microbenchmark written in GA and EL::DA in Figure 8. We emphasize that the data distribution of Elemental is element-wise cyclic, whereas GA in this case uses a regular distribution (each PE or process receives contiguous chunks of the global array). We use square matrices as input, with all (four) combinations of transpose operations for A and B matrices. For instance, in Figure 8, "8192-NT" corresponds to a matrix multiplication version where the input matrix dimensions are 8192×8192 and "NT" indicates A was not transposed (N), whereas B was transposed (T).

An important factor for scalability in DLA computation for distributed-memory architectures is the matrix distribution over processes. Predicting the best distribution is challenging because on the one hand we want the tile size per process to be sufficiently large for BLAS efficiency and, on the other hand, to be sufficiently small to avoid load imbalance due to communication. Elemental tries to attain a good compromise by removing the constraint on deciding the block sizes per process by making it 1, which means essentially that each process gets a noncontiguous element of the input matrix. Element-wise cyclic data distribution has been proven to be more scalable than previous approaches that partition the matrices into contiguous blocks and distribute the block to the processes [14]. As shown in Figure 8, the performance of EL::DA suffers when the cost associated with communication exceeds the cost associated with arithmetic operations (for instance, when 512 processes are used for multiplying 8K matrices, particularly with A/B transposed). Because of the data distribution, however, EL::DA shows scalability for large (16K) matrices and is 6%–40% better than GA in terms of performance.

C. Application Evaluation – GTFock

In computational chemistry, the Hartree-Fock (HF) or SCF (Self-Consistent Field) method is used in approximating the energy of a quantum many-body systems in a stationary state. This is an iterative method; in each SCF iteration, the most computationally intensive part is the calculation of the Fock matrix. GTFock [10] is a new parallel algorithm for Hartree-Fock calculations that uses fine-grained tasks to balance the computation. It also enables dynamically assigning tasks to processes to reduce communication. GTFock is an excellent application candidate because it exploits domain (MPI and GA), thread (OpenMP), and data (vector loops) parallelism, which are necessary to achieve effective parallel efficiency on the next generation of supercomputers. Because of the simi-

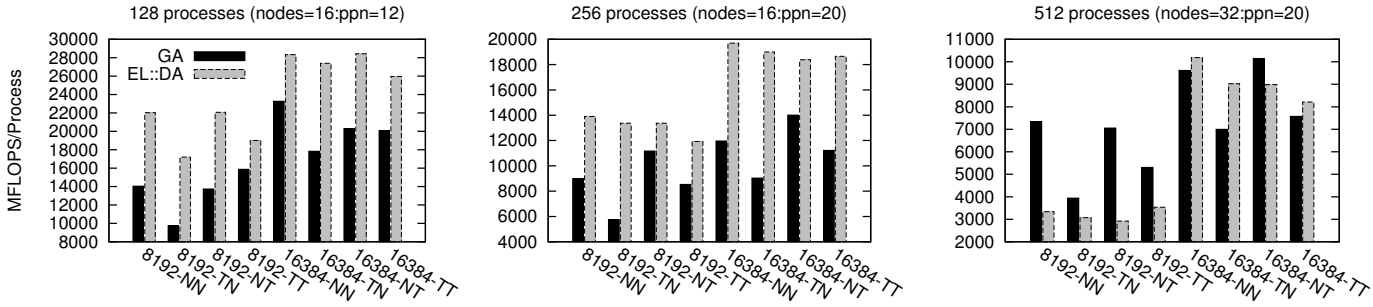

Fig. 8. Performance comparison of EL::DA and GA for the distributed matrix-matrix multiplication microbenchmark on Cori. Four processes per node are used by Casper for asynchronous progress.

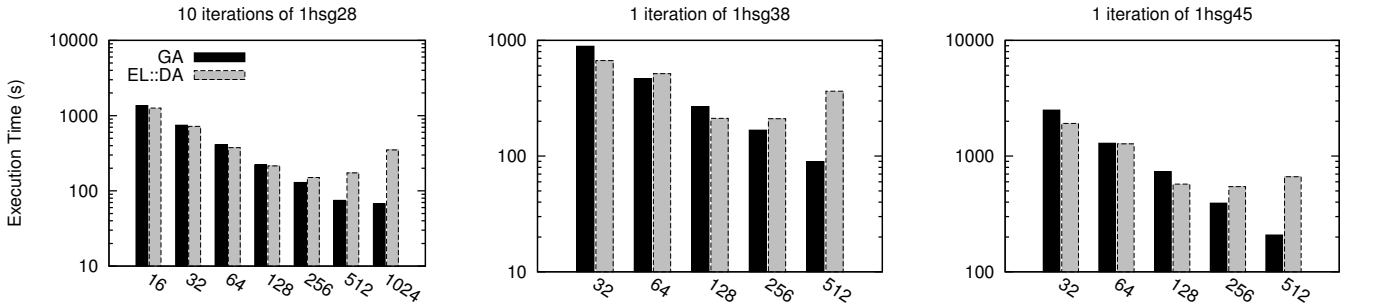

Fig. 9. GTFock execution on Cori (X-axis: Number of processes). Two processes per node are used by Casper for asynchronous progress.

TABLE I
TEST MOLECULES

Molecule	No. of Atoms	No. of Shells	No. of Basis Functions
1hsg_28	122	549	1159
1hsg_38	387	1701	3555
1hsg_45	554	2427	5065

larity of GA and EL::DA APIs, the EL::DA port of GTFock was straightforward, and essentially a drop-in replacement for GA functions.

Table I lists the molecules that we used in our experiments, and some of their properties that have a direct correlation with the input data size. All the molecules that we used as input are provided with the GTFock package as part of the cc-pVDZ basis set. Figure 9 shows the total execution time of only a single iteration for the 1hsg_38 and 1hsg_45 molecules, whereas the execution time of 10 iterations is reported for 1hsg_28. In Figure 9, we observe that for all inputs, EL::DA shows consistently around 20% improvement over GA up to 128 processes. However, with increasing number of processes, the volume of remote communication increases significantly in the Fock matrix building stage, which negatively affects the overall scalability of EL::DA. This is because in EL::DA, local accesses to a global array (via `NGA_Access`/`NGA_Release`) entail extra `MPI_Get` calls to bring elements distributed across the processes (in Elemental cell-cyclic fashion) into a local buffer of the current process. Therefore, `NGA_Access` is not a *local* operation for EL::DA, which in case of GA is a simple pointer assignment to a contiguous buffer (local portion of a global array), due to its data layout. Unfortunately, the design of GTFock relies on frequent local accesses to global arrays, which affects the performance of EL::DA over 256 processes. To maintain the integrity of the solution, EL::DA has to issue extra remote operations. Upon profiling GTFock with EL::DA on 512 processes, we found that more than 40% of the total execution time is spent on two functions that initialize and update the local portion of the global arrays (through `NGA_Access-NGA_Release`) before and after the Fock matrix computation, in every SCF iteration. On the other hand, in case of GTFock with GA, those functions merely contribute with around 1% to the entire execution time. If remote accesses to distributed arrays dominate an application (i.e., more `NGA_Acc`/`Put`/`Get` and less `NGA_Access`), then EL::DA will be more scalable and efficient than GA, as we demonstrated in the matrix multiplication microbenchmark. Parallel HF computations benefit (in terms of scalability) from an irregular data layout, where each process holds nonuniform data points. Elemental is a general purpose library and does not support the special layout that HF needs, while GA supports it because it was designed to support NWChem.

VI. CONCLUDING REMARKS

We presented a case study in designing a one-sided interface within a high-performance linear algebra and optimization framework. Our work started with improving the existing interface for updating distributed matrices in Elemental (*AxpyInterface*), and we justified the need for a new API (*RMAInterface*) for applications that require asynchronous

one-sided operations on distributed arrays. We built a Distributed Arrays interface (EL::DA) using the *RMAInterface* and Elemental `DistMatrix` to enhance the productivity of developers requiring optimized one-sided operations and a high-performance linear algebra framework. Integrating such an interface into Elemental opens up interesting possibilities in directly accessing a rich set of scientific algorithms, which is otherwise not possible from a standalone API such as GA. Overall, we demonstrated that our proposed *RMAInterface* is an effective programming model for asynchronous distributed matrix update delivering competitive performance results compared with those of existing MPI point-to-point APIs and GA. We saw that the general-purpose cyclic distribution used by Elemental is not optimal for HF computations, at least as implemented in GTFock, but Elemental and *RMAInterface* are designed to be general-purpose, not specifically to support a particular quantum chemistry method. Our future work plans include enhancing *RMAInterface* to support the different data structures (e.g., graphs) and distributions offered by Elemental.

ACKNOWLEDGEMENTS

We thank Jack Poulson, the original author of Elemental, for his support, Min Si for Casper related queries, and, Gail Pieper for reviewing the paper. We used resources of the NERSC facility, supported by U.S. DOE SC under Contract No. DE-AC02-05CH11231. We gratefully acknowledge the computing resources provided on Blues, a HPC cluster operated by the LCRC, ANL. We used resources of ALCF, which is a DOE SC User Facility supported under Contract DE-AC02-06CH11357. Assefaw Gebremedhin and Sayan Ghosh were supported in part by NSF CAREER award IIS-1553528.

REFERENCES

[1] S. Balay, W. Gropp, L. C. McInnes, and B. F. Smith. Petsc, the portable, extensible toolkit for scientific computation. *Argonne National Laboratory*, 2:17, 1998.

[2] D. Bonachea and J. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441. IEEE, 2011.

[4] S. Byna, W. Gropp, X-H. Sun, and R. Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *International Conference on Cluster Computing (CLUSTER)*, pages 412–419. IEEE, 2003.

[5] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers — design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

[6] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the global arrays PGAS model using MPI one-sided communication. In *26th International Parallel & Distributed Processing Symposium (IPDPS)*, pages 739–750. IEEE, 2012.

[7] W. Gropp, T. Hoefler, R. Thakur, and J. L. Träff. Performance expectations and guidelines for MPI derived datatypes. In *Recent Advances in the Message Passing Interface*, pages 150–159. Springer, 2011.

[8] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing openshmem using mpi-3 one-sided communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 44–58. Springer, 2014.

[9] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices*, 45(5):159–168, 2010.

[10] X. Liu, A. Patel, and E. Chow. A new scalable parallel algorithm for Fock matrix construction. In *28th International Parallel and Distributed Processing Symposium*, pages 902–914. IEEE, 2014.

[11] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H. Bungartz, and H. Lederer. The elpa library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21):213201, 2014.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[14] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[15] R. Reussner, J. L. Träff, and G. Hunzelmann. A benchmark for MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 10–17. Springer, 2000.

[16] M. Si, A. J. Peña, J. Hammond, P. Balaji, and Y. Ishikawa. Scaling NWChem with efficient and portable asynchronous communication in MPI RMA. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 811–816. IEEE, 2015.

[17] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An asynchronous progress model for MPI RMA on many-core architectures. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 665–676. IEEE, 2015.

[18] M. ten Bruggencate and D. Roweth. DMAPP - An API for one-sided program models on baker systems. In *Cray User Group Conference*, 2010.

[19] J. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 678–678, 1999.

[20] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, et al. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[21] R. A. Van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. MIT Press, 1997.