An Out-of-core Eigensolver on SSD-equipped Clusters

Zheng Zhou*‡, Erik Saule*, Hasan Metin Aktulga§, Chao Yang§, Esmond G. Ng§,
Pieter Maris¶, James P. Vary¶ and Ümit V. Çatalyürek*†
*Dept. of Biomedical Informatics, The Ohio State University, Columbus, OH 43210, USA
‡Wuhan University, P. R. China
§Computational Research Division, Lawrence Berkeley National Lab, Berkeley, CA 94720, USA
¶Department of Physics, Iowa State University, Ames, IA 50011, USA
†Dept. of Electrical and Computer Engineering, The Ohio State University, Columbus, OH 43210, USA

***Abstract*—Obtaining highly accurate predictions on properties of light atomic nuclei using the Configuration Interaction (CI) approach requires computing few extremal eigenpairs of a large many-body nuclear Hamiltonian matrix, $\hat{H}$. A forefront challenge in CI calculations is the massive size of $\hat{H}$ and its eigenvectors. The emergence of clusters equipped with non-volatile NAND-flash memory based solid state drives (SSD) presents unique opportunities. In this paper, we present the implementation details of an out-of-core eigensolver using a novel distributed out-of-core linear algebra framework, called DOoC+LAF. The framework provides an easy-to-use high-level application interface for linear algebra operations while providing efficient execution by orchestrating pipelined execution of computation, communication and I/O. We demonstrate the effectiveness of our out-of-core eigensolver implemented using DOoC+LAF by reporting performance results on large-scale eigenvalue problems arising in nuclear structure calculations.**

I. Introduction

The solution of the quantum many-body problem transcends several areas of physics and chemistry. Nuclear physics faces the multiple hurdles of a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The configuration interaction (CI) method allows computing the many-body wavefunctions associated with the discrete energy levels of nuclei with high accuracy [1], [2].

Typically, one is only interested in a limited number of low energy states, which can be computed by partially diagonalizing the nuclear many-body Hamiltonian, $\hat{H}$. In MFDn (Many Fermion Dynamics for nuclear structure), the Hamiltonian $\hat{H}$ is constructed in a many-body basis space based on the harmonic oscillator single-particle wavefunctions [3], [4]. Since $\hat{H}$ is a large sparse matrix, a parallel iterative eigensolver is preferred. These calculations require using large clusters.

A forefront challenge in CI calculations is the massive size of $\hat{H}$ and its eigenvectors. The construction of $\hat{H}$ is typically two orders of magnitude more expensive than a single iteration of the eigensolver. Therefore in MFDn, once $\hat{H}$ is constructed, it is stored throughout the computations, rather than reconstructing it at each iteration. Consequently, calculations that are possible using this approach are limited by the amount of storage available.

While disk-based storage systems provide abundant storage space at a low cost, their high latency and low bandwidth would make data access a severe bottleneck in large computations. On the other hand, the emergence of clusters equipped with non-volatile NAND-flash memory based solid state drives (SSD) presents unique opportunities. Since SSDs have no moving parts, they can achieve much higher I/O operations per second and sustained read/write bandwidths compared to magnetic disks. For example, the recently released OCZ Z-Drive R4 CloudServ Series PCIe-based flash storage cards can achieve up to 1.4 million IOPS and 6 GB/s sustained read/write bandwidth. Moreover, these high-performance storage cards can provide terabytes of storage space, *e.g.* the OCZ Z-Drive R4 can provide up to 16 TB of storage on a single card. The use of SSDs is being investigated for challenging problems such as graph traversal [5].

In this paper, we explore the advantages and challenges associated with using an SSD-equipped cluster for a data-intensive scientific application like MFDn. We restrict our attention to the part that requires accessing large amounts of data in MFDn, the eigenvalue solver. Since the convergence to the desired eigenpairs typically take hundreds of iterations, solution of the eigenvalue problem is also the most time-consuming part overall. Section II gives a description of the eigenvalue problem in the nuclear CI approach, and the iterative procedure that we adopt to solve this problem. Section III discusses the requirements for an efficient parallel out-of-core eigensolver. Since the $\hat{H}$ matrix is stored on the storage system, matrix blocks need to be brought into the local memory of compute nodes for processing. For this purpose, we use a novel distributed out-of-core linear algebra framework, called DOoC+LAF. This framework, presented in Section IV, provides an easy-to-use high-level application interface for linear algebra operations, while providing efficient execution by orchestrating pipelined execution of computation, communication and I/O. Section V presents the

This work was supported in part by U.S. Department of Energy Grant DE-FC02-09ER41582 (SciDAC/UNEDF), DE-FG02-87ER40371, and DE-FC02-06ER2775, and by the US NSF grants 0643969, 0904809 and 0904802, and 0904782. Computational resources were provided by the National Energy Research Supercomputer Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy. The authors would like to thank Shane Canon for his help regarding the SSD-testbed.

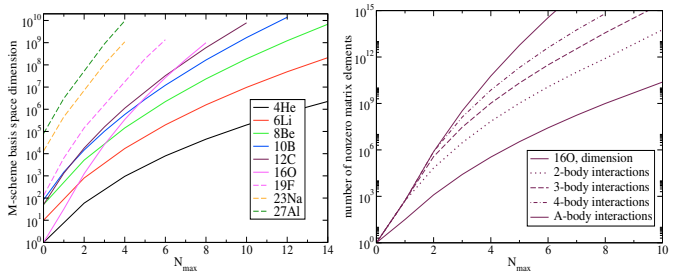


Fig. 1. The dimension and the number of non-zero matrix elements of the various nuclear Hamiltonian matrices.

implementation details of our out-of-core eigensolver using the DOoC+LAF Application Programmer's Interface (API). Finally, we demonstrate the effectiveness of our out-of-core eigensolver by reporting performance results on large-scale eigenvalue problems arising in nuclear structure calculations. We conclude by discussing how the ideas presented can be extended to scientific computing applications in general.

II. Eigenvalue Problem in the Configuration Interaction Approach

The eigenvalue problem arises in nuclear structure calculations because the nuclear wave functions Ψ are solutions of the many-body Schrödinger's equation:

$$H\psi = E\psi \quad (1)$$

$$H = \sum_{i<j} \frac{(p_i - p_j)^2}{2mA} + \sum_{i<j} V_{ij} + \sum_{i<j<k} V_{ijk} + \ldots \quad (2)$$

In the CI approach, both the wave functions ψ and the Hamiltonian H are expanded in a finite basis of Slater determinant of single-particle states (anti-symmetrized product of single-particle states). Each element of this basis is referred to as a many-body basis state. The representation of H under this basis expansion is a sparse symmetric matrix $\hat{H}$. Thus, in CI calculations, Schrödinger's equation becomes a finite-dimensional eigenvalue problem, where we seek the lowest eigenvalues (energies) and their associated eigenvectors (wave functions). Many-body basis state i corresponds to the ith row and column of the Hamiltonian matrix. The total number of many-body states or the dimension of $\hat{H}$ in our adopted harmonic oscillator (HO) basis, which we denote by n, is controlled by the number of particles A, the truncation parameter $N_{\max}$, and the maximum number of HO quanta above the minimum for a given nucleus (see Figure 1). Higher $N_{\max}$ values yield more accurate results for the same nucleus, but at the expense of an exponential growth in the dimension of $\hat{H}$. The sparsity of $\hat{H}$ is determined by the interaction potential used. This paper uses a 2-body interaction potential.

In order to find the lowest nev number of eigenvalues and eigenvectors of $\hat{H}$, we use the locally optimal block preconditioned conjugate gradient (LOBPCG) algorithm [6]. As mentioned above, in this paper we are focused on the efficient execution of a single LOBPCG iteration in our out-of-core approach, rather than how fast the LOBPCG algorithm converges for a given nuclear structure calculation. Therefore, for simplicity of presentation, we take the preconditioning matrix M to be the identity matrix. Algorithm 1 gives the pseudocode for a simplified version of the LOBPCG algorithm, assuming $M = I$.

Algorithm 1: Pseudocode of the LOBPCG algorithm for the eigenproblem of the form $\hat{H}\Psi = E\Psi$. The preconditioning matrix M is assumed to be the identity matrix.

Input: $\hat{H}$, matrix of dimensions $n \times n$
Input: Ψ_0, a block of vectors of dimensions $n \times nev$
Output: Ψ and E such that $\|\hat{H}\Psi - \Psi E\|_F$ is small.
Orthonormalize the columns of Ψ_0
$P_0 \leftarrow 0$
for *i = 0, 1, …, until convergence* **do**
  $E_i \leftarrow {\Psi_i}^T \Psi_i / {\Psi_i}^T \hat{H} \Psi_i$
  $R_i \leftarrow \Psi_i - E_i \hat{H} \Psi_i$
  Use Rayleigh-Ritz method on the span$\{\Psi_i, R_i, P_i\}$
  $\Psi_{i+1} \leftarrow \underset{Y \in span\{\Psi_i, R_i, P_i\}}{\operatorname{argmin}} Y^T Y / Y^T \hat{H} Y$
  $P_{i+1} \leftarrow \Psi_i$
  Check convergence

III. An Out-of-core Parallel Eigensolver

Calculations for the solution of the nuclear eigenvalue problem need to be performed in parallel due to the large amount of computations involved. In a large-scale parallel computation, a good load balance among computing nodes and hiding overheads associated with data movement are necessary to perform an efficient execution. Data movement in a parallel computation generally refers to inter-node communication; in an out-of-core implementation the I/O overheads associated with accessing data on the storage system can be significant.

In Algorithm 1, the main computational tasks are the multiplication of a sparse matrix with a dense block of vectors (SpMM), $\hat{H}\Psi$, and the inner products between long dense vector blocks, in particular $Y^T Y$ and $Y^T \hat{H} Y$. In this section, we describe how these tasks are carried out in parallel.

A. Load-balanced Problem Decomposition

The original ordering of the many-body basis states as they are constructed from the single-particle states leads to an uneven distribution of the non-zero matrix elements in $\hat{H}$. Note that in SpMM computations, the load on a compute node is proportional to the number of non-zero matrix elements assigned to that node. Therefore, with an uneven distribution of the non-zeros in $\hat{H}$, achieving a load balanced computation would be a challenging task. An even distribution of non-zeros is accomplished in MFDn by an appropriate matrix reordering through row/column permutation. As a result, an even 1D partitioning of the rows of $\hat{H}$ among compute nodes would yield a load balanced computation. However, as mentioned above, the nuclear Hamiltonian matrix is symmetric. Since the storage space is at premium, we store only half of the $\hat{H}$ matrix. As depicted in the left subfigure of Figure 2(a), in a 1D decomposition of the lower triangular part of $\hat{H}$, the load assigned to the first compute node would be significantly less compared to the load assigned to the last node. Therefore,

for an $n \times n$ matrix $\hat{H}$, we store the first $n/2$ subdiagonals in the lower triangular part, and the superdiagonals $n/2+1$ through n in the upper triangular part. In this case, an even 1D partitioning of the rows of $\hat{H}$ gives a good load balance among the compute nodes, as shown in the right of Figure 2(a).

B. Storage and Data Access

In our out-of-core implementation, once the nuclear Hamiltonian is constructed, it is stored on the permanent storage system throughout the entire calculation. Since the size of a row partition assigned to a compute node is generally too large to fit into that node's memory, each row partition is further divided into several smaller blocks. Each matrix block (except for the end-cases) is a sparse square matrix of dimensions $b \times b$, where $b << n$. The matrix block on the ith block row and the jth block column is labeled as $\hat{H}_{ij}$. The total number of matrix blocks is $(n_b^2+n_b)/2$, where $n_b = \lceil n/b \rceil$. Let $[rs_k, re_k]$ denote the range of the row indices of blocks assigned to the compute node k for $k = 1, 2, \ldots, n_p$, where n_p denotes the total number of nodes. Each node stores this range information for all other nodes for reasons that are described later.

We use the Compressed Sparse Column (CSC) format to store the matrix blocks. Each block is stored in a separate binary file on the permanent storage. The even distribution of the non-zero matrix elements in $\hat{H}$ ensures that all files are of approximately the same size and the computational cost associated with processing a matrix block is also about the same. (Except the diagonal blocks are twice smaller.)

The dense vector blocks Ψ and the result of the SpMM $\hat{H}\Psi$ are also partitioned into subvectors Ψ_i and $\hat{H}\Psi_i$ for $i = 1, 2, \ldots, n_b$, according to the partitioning of $\hat{H}$ into blocks. Since the vectors are dense and the dimension of $\hat{H}$ can be very large, the space required for storing Ψ_i and $\hat{H}\Psi_i$ subvectors can be significant. However, in MFDn only a few lowest eigenvectors are needed, *i.e.*, nev is on the order of 10-15 eigenvectors. Therefore, the subvectors Ψ_i and $\hat{H}\Psi_i$ for $i \in [rs_k, re_k]$ are stored on the local memory of the kth compute node, which is the *host* node for these subvectors.

Figure 2(b) gives an overview of data accesses required for processing a matrix block $\hat{H}_{ij}$ on node p. Since only half of the Hamiltonian is stored, there are two SpMM computations associated with each matrix block: $\hat{H}_{ij} \times \Psi_j = \hat{H}\Psi'_i$, and the transpose operation $\hat{H}_{ij}^T \times \Psi_i = \hat{H}\Psi'_j$. Before the computations start, block $\hat{H}_{ij}$ is brought into the node p's memory from the permanent storage system. The vector block Ψ_i is already hosted locally by node p. The host of the vector block Ψ_j, node k, is found by locating the range where $rs_k \leq j \leq re_k$, for $1 \leq k \leq n_p$. Then Ψ_j is brought into the local memory by communicating with the host node k, assuming $p \neq k$.

Figure 2(c) gives an overview of the SpMM computations performed within a compute node, after all the input data is brought to local memory. Processing a single $\hat{H}_{ij}$ block results in two intermediate output vector blocks that need to be reduced to obtain the global result vectors $\hat{H}\Psi$. While $\hat{H}\Psi'_i$ is reduced locally on node p, $\hat{H}\Psi'_j$ needs to be communicated back to the host node k for reduction. Finally, the matrix block $\hat{H}_{ij}$ is discarded from the volatile memory to free up space for processing the next block.

Once the SpMM computations are completed, the inner products Y^TY and $Y^T\hat{H}Y$ required for the Rayleigh quotient estimations are computed in parallel. First, each compute node performs the inner products of the vector block partitions that it hosts. The actual estimate can be obtained through an all-reduce operation among all compute nodes.

C. Overlapping I/O and Communication with Computation

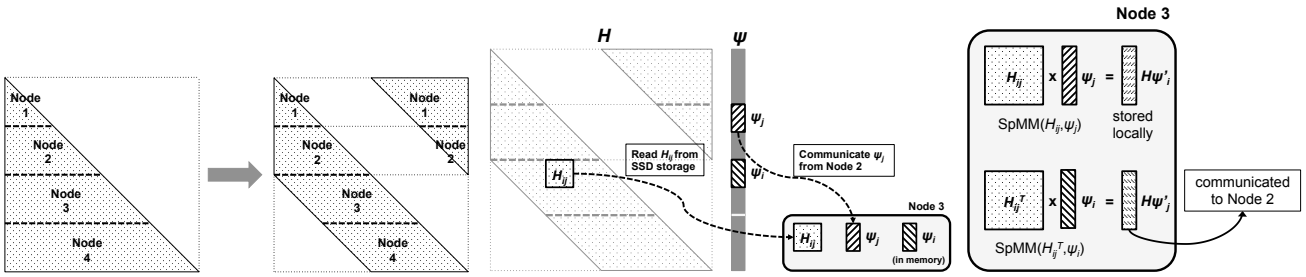
During the SpMM computations, overheads associated with data accesses can severely affect the overall efficiency of the eigenvalue computations. The time required to read a matrix block (which is of size several hundred MBs) from the permanent storage system, and to communicate the dense vector blocks from/to host nodes can be as much as the time required to process a given matrix block, or even more in some cases. Therefore, overlapping the I/O and communication operations with SpMM computations is highly desirable.

The main idea here is to buffer (*i.e.*, prefetch) the upcoming matrix block(s) and the required input vector(s), while still processing the current matrix block. In this way, as soon as the processing the current matrix block is finished, the processing can continue with the next block(s) without idling in between. Furthermore, it turns out that the order in which the matrix blocks are processed can have a significant effect. If the compute node p processes the matrix blocks in a column-major order, *i.e.,* $\hat{H}_{rs_p,j}, \ldots, \hat{H}_{ij}, \hat{H}_{i+1,j}, \ldots, \hat{H}_{re_p,j}$ and then continues with the $j+1$st column, the overall communication overhead can be reduced significantly. This is because all the matrix blocks in the jth column require the same input subvectors Ψ_j, which need to be communicated from the host node k. Also note that the intermediate results $\hat{H}\Psi'_j$ of processing the matrix blocks in the jth column are all destined to the same host node for reduction. Instead of sending all intermediate results one by one and having them reduced on node k, compute node p can perform the reductions on $\hat{H}\Psi'_j$ subvectors locally, and communicate a single message to node k, once it finishes processing the jth column.

IV. A Distributed Out-of-Core Middleware with a Linear Algebra Frontend

In a recent work, we have started development of a generalized middleware for distributed out-of-core computation and data analysis [7], which we call DOoC (**D**istributed **O**ut-**o**f-**C**ore). DOoC runs on top of DataCutter [8], which itself is a distributed, coarse-grain dataflow middleware. We have built our framework on top of DataCutter instead of directly implementing using MPI (or any other low-level library that enables distributed-memory programming), because the programing model of DataCutter naturally enables separation of computation and data movements and provides an efficient runtime system that orchestrates pipelined executions with computation and communication overlapping.

Figure 3 depicts the architectural overview of our proposed framework, which is composed of DOoC and LAF (**L**inear



(a) Load balanced 1D decomposition of the Hamiltonian matrix.

(b) IO and communication operations for processing each matrix block.

(c) SpMM computations within a node.

Fig. 2. Distributed out-of-core SpMM.

Algebra **F**rontend). DOoC provides efficient execution of task graphs with given input and output data dependencies, and LAF acts as a frontend that translates basic linear algebra primitives into global task graphs that can be executed by DOoC. This work extends our previous work in three directions. First, in our earlier work task graphs and task codes were manually generated by the application developer. Here, we develop a customization of the framework for linear algebra computations (*i.e.*, LAF) which provides high-level interface to application developer. Second, we present a complete eigensolver instead of a simplified repeated sparse-matrix vector multiplication kernel. Last but not least, we strengthen the implementation of DOoC by improving its global scheduler, and making the storage layer more robust and proactive.

A. Distributed Out-of-Core (DOoC) Middleware

The Distributed Out-of-Core middleware is composed of two parts: (i) a hierarchical scheduler responsible for ordering and triggering the execution of tasks, and (ii) a storage service responsible for managing the memory as a resource and handling transfers of data, which is either the input for local computational tasks or the output of them. Data transfer in the context of a distributed out-of-core computation involves reading from or writing to the permanent storage system, or communicating with other compute nodes.

1) Global and Local Schedulers: Within the scheduler, the application is represented as a set of tasks. Each task is annotated with the set of data it needs (input data) and the set of data it generates (output data). These annotations are used to generate a partial ordering between the tasks (such as the one presented in Figure 3). An efficient partial ordering is achieved by the use of hash tables, where for each data the mapping of which tasks use it as input and which tasks produce it as output is kept.

Each individual task is sequentially executed on a single computing node. One can draw similarities between our approach and other approaches that use directed acyclic graphs (DAG) to model computational dependencies. In the classical DAG scheduling [9], the complete task graph is generated before scheduling. However, in our system the task graph is generated dynamically on-the-fly, as done in DAGuE [10]. DAGuE is designed for in-core, dense linear algebra computations. Our system is designed for efficient, distributed memory, out-of-core execution of general data-intensive applications.

The tasks are created on the global scheduler. The global scheduler is responsible for assigning these tasks to the local schedulers on compute nodes for processing, as well as tracking the completions of those tasks. It assigns a task to a local scheduler only when all the input data of the task have been generated or will be generated as a result of executing the tasks already assigned to that particular local scheduler. Among all the compute nodes, the global scheduler picks the node where most of the input data is already located in memory for executing a task. This is a heuristic aimed at minimizing the data movement required for starting to process tasks. Alternatively, a task assignment can be forced to a different node by the application programmer.

The local scheduler obtains regularly (default every 100 ms) from the storage service the list of data that is available on the local memory. Based on this information, the local scheduler decides which tasks among those assigned to itself are ready for execution. The scheduler triggers the execution of a ready task as soon as a computation thread becomes idle. There are as many computation threads as the number of cores on a compute node. The output data from executing a task, which will serve as the input data for a subsequent task, resides in the compute node's memory until it is consumed.

Another key responsibility of the local scheduler is to enable the pipelined execution of computation, communication and I/O. It achieves this task by sending prefetching requests to the storage service. The local scheduler first queries the storage service to learn the amount of memory space available for prefetching. As long as there is space available and tasks waiting to be executed, local scheduler determines the data to be prefetched by using the greedy algorithm presented in Algorithm 2 to order tasks. This greedy algorithm orders the tasks in the local scheduler's list based on the amount of additional input data that needs to be brought into the local memory to make each task ready for execution. The task which requires the least amount of additional input data is ordered first, and the prefetching requests for its input data are issued. Those input data are added to the list of available data, and the algorithm continues to determine the next task for prefetching. Please note that, data will be actually available after it has been prefetched by the storage service. Prefetching is paused when there is no more memory space available. The prefetched data is consumed when ready tasks are executed. As soon as enough

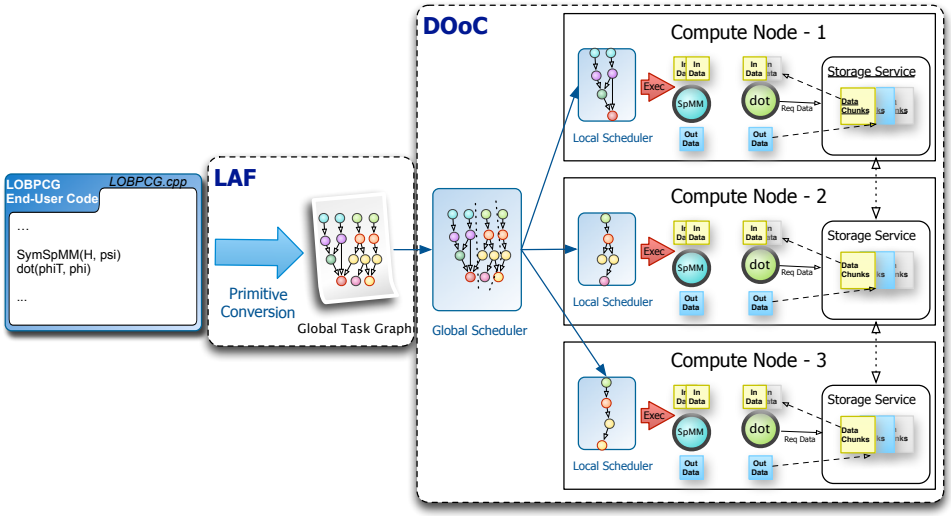


Fig. 3. Schematic overview of our framework Distributed Out-of-Core (DOoC) with Linear Algebra Frontend (LAF).

memory space becomes available, prefetching is reinstantiated.

Algorithm 2: Task Ordering Algorithm.

```
AVAILDATA ← storage().getAvailData()
AVAILMEM ← storage().getAvailMem()
TASKS ← global().getSchedulableTasks()
OUTOFMEM ← False
PREFETCHLIST ← ∅
while not OUTOFMEM & not TASK.empty() do
    for t ∈ TASKS do
        TOFETCH ← input_data(t) - AVAILDATA
        cost_t ← Σ_{d∈ToFetch} size_of (d)
    t* = argmin_{t∈Tasks} cost_t
    TOADD ← input_data(t*) - PREFETCHLIST
    for d ∈ TOADD do
        if size_of (d) > AVAILMEM then
            OUTOFMEM ← True
        else
            PREFETCHLIST ← PREFETCHLIST ∪ {d}
            AVAILDATA ← AVAILDATA ∪ {d}
            AVAILMEM ← AVAILMEM - size_of(d)
    TASKS ← TASKS − {t*}
return PREFETCHLIST
```

2) Storage Service: The storage service is responsible for managing the local memory, managing the data transfer to/from the permanent storage system and handling the communication between compute nodes.

When the storage service starts, it queries the permanent storage system through its file system and makes a list of the data stored there. This information is reported to the global scheduler. In addition, the storage service provides functions to declare new data objects and to destroy ones that are no longer necessary. In DOoC, declaring a new data object does not actually induce memory allocation, just the creation of appropriate meta-data. The memory allocation is done when the newly created data object is accessed for the first time.

The way DOoC handles an access to a data object differs based on whether it is a read access or a write access. In a read access, if the data object is currently not in the local memory, it is being stored either on the permanent storage system or is being hosted by another node. If the data is stored on permanent storage system, it is simply read from there. Otherwise, it needs to be communicated from the hosting node. The storage service randomly queries other nodes until it locates the one where the data object is stored. Once the data is located, a *hint* is created to speedup the querying process in subsequent accesses to the same data object.

Write access to a data object is only possible if the data object resides in local memory. Notice that because the data objects in DOoC are immutable, they are only written once. Therefore there is no need for a complex coherency protocol. All data access operations are performed asynchronously to be able to process multiple requests simultaneously. However, after a certain number of simultaneous requests (default: 50) within a node, subsequent ones are queued.

A deallocation procedure is triggered when there is no more memory available on a compute node. The input data that are necessary for executing the tasks scheduled for that compute node, as well as any data object that cannot be reobtained are excluded from deallocation. A data object cannot be reobtained, if it was created on the node itself. Such data objects must be kept until they are written to the permanent storage system or they are explicitly deallocated by the application programmer. On the other hand, a data object can be reobtained if it was read from the permanent storage system, or communicated from another node. Such data objects are eligible for deallocation along with remaining data objects that do not fit into any of the categories above. The storage service frees data objects eligible for deallocation according to the Least Recently Used (LRU) policy.

B. Linear Algebra Frontend (LAF)

The Linear Algebra Frontend is a C++ library which works with objects of different data types including dense and sparse matrices, (dense) vectors, and scalars. Objects are persistent, and can be partitioned into chunks and distributed in the system. Each object is identified by a string that gives it a unique name. Each object is considered immutable, similar

Primitives	Operation
Primitives that creates Matrix	
MM, (Sym)SpMM	$C = AB$
addM	$C = A + B$
axpyM	$C = aA + b$
randomM	$C = random()$
Primitives that creates Vector	
MV, (Sym)SpMV	$y = Ax$
addV	$y = x + w$
axpyV	$y = ax + b$
Primitives that creates scalar	
dot	$a =< x, y >$

TABLE I
PRIMITIVES THAT ARE CURRENTLY AVAILABLE IN LAF. A, B AND C ARE MATRICES, y, x AND w ARE VECTORS, AND a AND b ARE SCALARS.

to objects in functional programming. Hence it is generated once and is never overwritten. New objects can be generated from the stored data, and also as a result of computation using provided primitives.

When an object is no longer needed, the associated memory needs to be deallocated within the system. This is triggered upon the destruction of the object in the frontend which can be explicit or automatic when the program exits the scope an object was declared in.

Currently supported primitives are listed in Table I. Although not comprehensive, these operations are sufficient to implement various useful numerical methods including the Conjugate Gradients, LOBPCG, Lanczos, and Page-Rank algorithms.

Some of these primitives (such as dot product, MM and MV) require a reduction phase when the data are partitioned into multiple chunks. The reduction operation can be implemented using a static reduction tree. Since the summation operation required for these reductions are commutative, it does not matter in which order the different chunks are added up. So the reduction is first performed locally on each node and then globally on the destination node to reduce communication overheads. In order to prevent the accumulation of intermediate results on a node (which may be very costly in terms of memory space), local reduction tasks are implemented to listen on scheduling events. When the number of intermediate results associated with a reduction operation reaches a threshold (default: 5), a local reduction task is dynamically created.

V. OUT-OF CORE EIGENSOLVER USING DOOC+LAF

Our out-of-core eigensolver is an implementation of the LOBPCG algorithm given in Algorithm 1 using the linear algebra primitives of the DOoC+LAF framework. The implementation is composed of two main parts: symmetric SpMM computations, followed by two inner products. Each matrix block $\hat{H}_{ij}$ stored on the permanent storage system essentially corresponds to a task, which we denote by `SymSpMM`(i, j). The input data of `SymSpMM`(i, j) are Ψ_i and Ψ_j subvectors. The 1D decomposition of the matrix $\hat{H}$ is ensured by having the compute node p create the subvector blocks $\Psi_{rs_p}, \Psi_{rs_p+1}, \ldots, \Psi_{re_p}$ for the initial guess Ψ using the DOoC+LAF primitive `randomM`. As mentioned above, the global scheduler assigns each task to the compute node which stores the most amount of input data required for that task. Consequently, all tasks `SymSpMM`(i, j), where $rs_p \leq i \leq re_p$ and $1 \leq j \leq n_b$, would be scheduled to the compute node p, essentially resulting in the load balanced 1D decomposition of the SpMM operation as described in Section III-A.

As a result of executing the task `SymSpMM`(i, j) on node p, two intermediate output vector blocks of $\hat{H}\Psi'_i$ and $\hat{H}\Psi'_j$ are produced. $\hat{H}\Psi'_i$ is consumed by a local reduction task denoted by `addV`$(\hat{H}\Psi_i, \hat{H}\Psi'_i)$ on node p. Similarly, $\hat{H}\Psi'_j$ is consumed by the task `addV`$(\hat{H}\Psi_j, \hat{H}\Psi'_j)$. However, note that $\hat{H}\Psi_j$ is stored on node k such that $rs_k \leq j \leq re_k$. Assuming that $k \neq p$, the intermediate result vectors $\hat{H}\Psi'_j$ first need to be communicated to node k for the execution of the task `addV`$(\hat{H}\Psi_j, \hat{H}\Psi'_j)$.

Lemma: Assume that on node p, the difference between the sizes of the smallest and largest matrix blocks (as measured by the space required to store a block in CSC format) is less than the size of any vector block Ψ_i, for $1 \leq i \leq n_b$. Then Algorithm 2 orders the set of tasks on node p $\{\texttt{SymSpMM}(i, j) \,|\, rs_p \leq i \leq re_p \wedge 1 \leq j \leq n_b\}$ such that they are executed in a column-major order.

Proof: Without loss of generality, let `SymSpMM`(rs_p, j) be the first task executed on node p for some j. Then the subvector Ψ_j is the only input data on the local memory of node p, besides the locally stored subvectors Ψ_i for $rs_p \leq i \leq re_p$. Additional input data required to execute other tasks associated with the matrix blocks in the jth column is the matrix block itself only. However, to execute a task corresponding to a matrix block in a column $c \neq j$, both the matrix block and the subvector Ψ_c would be needed. Hence, the tasks of the jth column would be ordered by Algorithm 2 before the tasks in any other column. This leads to a column-major processing of matrix-blocks. □

As a result, our out-of-core implementation using the DOoC+LAF framework is able to execute the computations related to the solution of the eigenvalue problem in a way that reduces the communication overheads, as discussed in Section III-C. It is a natural result of the task ordering algorithm, and the pipelined execution of computation, communication and I/O operations in the DOoC+LAF framework. Since no explicit effort is required to achieve this, a significant burden on the application programmer is removed.

After the symmetric SpMM computations are completed, two inner products of the form $Y^T Y$ and $Y^T \hat{H} Y$, where $Y = \text{span}\{\Psi, R, P\}$ and $\hat{H}Y = \text{span}\{\hat{H}\Psi, \hat{H}R, \hat{H}P\}$, need to be performed. Vector blocks R and P, and consequently Y, are also partitioned according to the partitioning of Ψ and $\hat{H}\Psi$. Hence these inner products are performed on node k, for $k = 1, 2, \ldots, n_p$, as a set of tasks denoted by `dot`(Y_i, Y_i) and `dot`$(Y_i, \hat{H}Y_i)$, where $rs_k \leq i \leq re_k$. The local inner products are reduced on node 1. Then all computing nodes estimate the Rayleigh quotients. Once the estimates for eigenvalues E and eigenvectors Ψ are obtained, the computation continues with the next iteration.

	$N_{\max}$=7	$N_{\max}$=8
Matrix Dimension (n)	46.6×10^6	159.9×10^6
# Nonzero matrix elements	28.1×10^9	123.6×10^9
Total matrix size	209 GB	920 GB
# Block row/columns (n_b)	43	87
Total number of matrix blocks	946	3828
Average size of a matrix block	226 MB	246 MB

TABLE II
GENERAL INFORMATION ON THE TESTCASES

VI. PERFORMANCE EVALUATION

Experiments are run on an experimental SSD testbed on the Carver cluster at NERSC. The testbed is composed of 48 nodes: 40 computational nodes and 8 I/O nodes. Each node is equipped with two Intel Xeon X5550 processors clocked at 2.67 GHz (4 cores each, hyper-threading is disabled) and 24 GB of DDR3 memory. Each node runs on Red Hat 5.5 with Linux kernel 2.6.18-238.12.1.el5. Nodes are interconnected by 4X QDR InfiniBand technology, providing 32 Gb/s of point-to-point bandwidth for high-performance message passing and I/O. Our codes are compiled with GCC 4.5.2. The InfiniBand interconnect is leveraged through the use of the MVAPICH 1.2 library. Each I/O node is equipped with two SSD cards, Virident tachIOn 400 GB, connected through the PCI-express bus. Each card can deliver up to 1 GB/s sustained read bandwidth, leading to a peak bandwidth of 2 GB/s per I/O node, and 16 GB/s maximum I/O bandwidth from the permanent storage system to the compute nodes. I/O nodes are accessed by the compute nodes through the Global Parallel File System [11]. Data is streamed from the I/O nodes to the compute nodes using the 4X QDR InfiniBand interconnect as well.

Performance evaluation of our out-of-core implementation is done with the nuclear structure computations of the ^{10}B (5 protons, 5 neutrons) nucleus. Two different truncation parameters $N_{\max}$=7 and $N_{\max}$=8 are used, leading to two eigenproblems of different sizes. Some key properties of these testcases are summarized in Table II. Since storage space is at premium for MFDn, matrix blocks are stored in single precision CSC format.

A. Practical Considerations

The number of eigenpairs to be computed is fixed at nev=8 for both test-cases. Table III gives detailed information regarding the sizes of vector blocks involved when nev=8. The size of the entire Ψ vector block, which is also stored in single precision, is 1.5 GB and 5.1 GB for $N_{\max}$=7 and $N_{\max}$=8 cases, respectively. In the LOBPCG algorithm, 6 such vectors (Ψ, R, P from the previous iteration and $\hat{H}\Psi, \hat{H}R, \hat{H}P$ of the current iteration) need to be hosted on the volatile memory available to compute nodes. The total space required for this purpose would be 9 GB and 30.6 GB for $N_{\max}$=7 and $N_{\max}$=8 cases, respectively. On Carver, about 5 GB of the 24 GB memory on a compute node is reserved for the OS kernel, and the network file system (NFS). Since matrix blocks to be read are on the order of hundreds of MBs, and the messages to be communicated are on the order of tens of MBs (see the size of Ψ_i in Table III), significant space is needed for the I/O and MPI buffers. As a result, only 15 GB out of the 24 GB memory on a compute node can be used by our out-of-core eigensolver. We choose to use at most 5 GB of the usable memory for hosting the vector blocks, and the remaining memory for processing the tasks. Therefore the minimum number of nodes required for $N_{\max}$=7 and $N_{\max}$=8 computations are 2 and 6, respectively.

	$N_{\max}$=7	$N_{\max}$=8
Number of eigenpairs (nev)	8	8
Size of a subvector block Ψ_i	34.7 MB	58.8 MB
Total size of the vector block Ψ	1.5 GB	5.1 GB
Total size of all 6 vector blocks	9 GB	30.6 GB

TABLE III
GENERAL INFORMATION ON THE VECTOR BLOCK SIZES

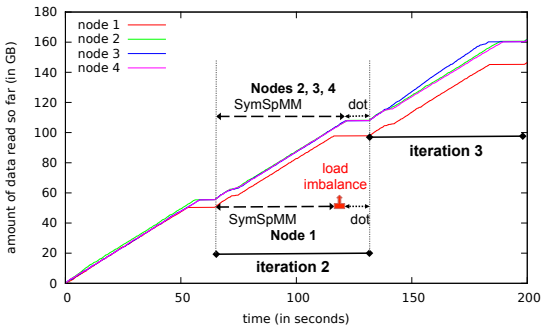


Fig. 4. Execution of the $N_{\max}$=7 case on 4 nodes with annotations.

We create 8 computation threads (one for each core), which collectively work on the tasks assigned to a node. Since there are lots of I/O and communication operations involved in our out-of-core eigensolver, per iteration timings may fluctuate during execution. Therefore in this section, we report the timings from the first 5 iterations of the LOBPCG algorithm for a reliable performance evaluation.

B. Performance Results for $N_{\max}$=7

The ^{10}B nucleus, $N_{\max}$=7 testcase is executed using different number of compute nodes that range from 2 to 22. The amount of data read from the permanent storage system during the execution when running on 4 nodes is shown in Figure 4. It gives important insights regarding the execution of the out-of-core eigensolver. While the amount of data read is increasing, the eigensolver is in the SpMM phase, and during the plateaus it is doing the inner products required for the Rayleigh quotient estimations. As expected, the out-of-core eigensolver spends most of the time doing SpMM computations.

Moreover, Figure 4 reveals that the computational load for SpMM is fairly well-balanced among the 4 compute nodes. While nodes 2, 3 and 4 finish their SpMM computations at about the same time, node 1 finishes its shortly before them. This is because there are 43 rows of blocks for the $N_{\max}$=7 case. Node 1 is responsible for 10 of those rows, while other nodes are responsible for 11 rows. Therefore node 1 has slightly less work to do.

Table IV presents for different number of nodes a break-down of the execution time for the first 5 iterations of the

$N_{\max}$=7 case into 4 parts: $t_{DAG-build}$ is the time spent for building the task graph, $t_{DAG-exec}$ is the time spent for dynamically determining the order of task execution, t_{IO} is the maximum I/O time spent by a compute node for reading matrix blocks from the SSD-based storage system, and t_{comp} is the maximum amount of time spent by a computation thread for SpMM as well as inner product computations. The total execution time is given by t_{total}. To compute the efficiency of an execution, we take the performance on 2 nodes as the base case with an efficiency of 1.0 (note that 2 is the minimum number of nodes required for the $N_{\max}$=7 case). Then for example, the efficiency on 5 nodes is computed by the formula $\frac{2*t_{total}(2)}{5*t_{total}(5)}$. What is missing from this table is how much time is spent for communication operations. The DOoC+LAF framework is still under development and this is a functionality which is currently not available. In order to give an idea about the load on the interconnection network, we report the total number of messages communicated during execution, $n_{messages}$, and the total communication volume V_{comm} in the last 2 rows of Table IV.

As seen in Table IV, the building and traversal of the task graph ($t_{DAG-build} + t_{DAG-exec}$) corresponds to only a few percent of t_{total} despite the scheduler handles about 30,000 tasks ($O(iter \times n_b^2)$). This shows that the overheads incurred by the DOoC+LAF runtime environment is negligible for the $N_{\max}$=7 case. On the other hand, the total execution time is much less than the time spent in I/O and computation ($t_{IO} + t_{comp}$) together, while going from 2 to 5 nodes. This shows that the DOoC+LAF runtime environment overlaps I/O and computation efficiently.

A close examination of t_{comp} and t_{IO} values going from 2 nodes to 11 or 22 nodes reveals that t_{comp} scales almost linearly, while t_{IO} scales super-linearly with increasing node counts. In fact, at 22 nodes I/O operations take only 13 seconds. This is not surprising given that the total amount of usable memory with 22 nodes (330 GB) is enough to store all of the matrix blocks together with all 6 vector blocks in local memory: the matrix is read only once.

However, on 11 and 22 nodes, the fact that t_{total} is greater than $t_{IO} + t_{comp}$ points to the presence of an important overhead. We suspect that this overhead is due to the communications associated with bringing the input vector blocks to the local memory and sending the intermediate output vector blocks back. The amount of communication volume generated by the algorithm with the 1D partitioning of the matrix is in $O(nev \times n \times n_p)$, where n is the matrix dimension and n_p is the number of compute nodes. This is because in a 1D decomposition of the symmetric Hamiltonian matrix, each node needs to access about half of the Ψ vector, and generates output vectors spanning the same fraction of $\hat{H}\Psi$ during the SpMM computations. $n_{messages}$ and V_{comm} values reported in Table IV confirm this analysis.

C. Performance Results for $N_{\max}$=8

The DOoC+LAF runtime environment generates a detailed log file on each compute node for all the steps it takes during the execution of a code. Analysis of these log files can give important insights. One way to analyze how our out-of-core eigensolver performs is to look at the number of jobs in the local scheduler's queue versus execution time plot, as shown in the top subfigure of Figure 5. Here we plot the first 3 iterations of the $N_{\max}$=8 case on 12 nodes. There are 87 rows of matrix blocks in this calculation, therefore 3 nodes (nodes 1, 3 and 4) are responsible for an extra row of matrix blocks compared to other compute nodes. This is reflected as a higher peak at the start of an iteration for those 3 nodes. The rise of the peak corresponds to the building and partitioning of the task graph part. The percentage of this part is again negligible compared to the total time per iteration. The fall of the peak means that the task graph is shrinking, because tasks are being executed. As seen in the plot, the peak falls at a constant slope during the SpMM computations. This means that computation and I/O operations are overlapped efficiently, and the SpMM computations progress smoothly, without idling.

#nodes	**2**	**3**	**4**	**5**	**11**	**22**
$t_{DAG-build}$ (s)	.1	.1	.1	.1	.1	.2
$t_{DAG-exec}$ (s)	4	4	4	4	3	3
t_{IO} (s)	577	394	280	230	73	13
t_{comp} (s)	511	352	264	218	98	54
t_{total} (s)	606	424	325	286	185	191
$efficiency$	1.0	.95	.93	.85	.60	.29
$n_{messages}$	444	648	912	1146	2550	5124
V_{comm} (GB)	13	20	27	33	74	149

TABLE IV
BREAK-DOWN OF THE EXECUTION TIME FOR THE FIRST 5 ITERATIONS OF THE $N_{\max}$=7 CASE USING DIFFERENT NUMBER OF COMPUTE NODES. THE LAST 2 ROWS SHOW THE LOAD ON THE INTERCONNECTION NETWORK.

When using the DOoC+LAF framework, it is important to keep track of the amount of memory available. Because this memory is used to prefetch the data of the upcoming tasks. Here, the available memory is used to buffer the blocks of $\hat{H}$ from the file system and Ψ_i vectors from other nodes. If the available memory is low, the prefetching is no longer possible, the computation are sequentialized and the overlapping of I/Os, computations and communications might not be effective.

The bottom plot in Figure 5 shows the amount of available memory as execution progresses. At the start of an iteration, the local scheduler reserves memory space and issues prefetching requests for the initial batch of matrix blocks. This results in a sharp drop in the amount of memory available. As tasks associated with these matrix blocks are completed, the memory space that becomes available is filled in further with other matrix blocks. Once all the SpMM tasks are finished, we see a sudden jump at the amount of memory space available. This is because the inner product computations do not consume much memory. The slight load imbalance caused due to the higher number of tasks on 3 nodes, is reflected as a phase difference in this plot. Nodes 1, 3 and 4 finish their SpMM computations a little after other compute nodes, and the amount of memory available makes a peak slightly later on these nodes.

Finally, in Table V for different number of nodes, we

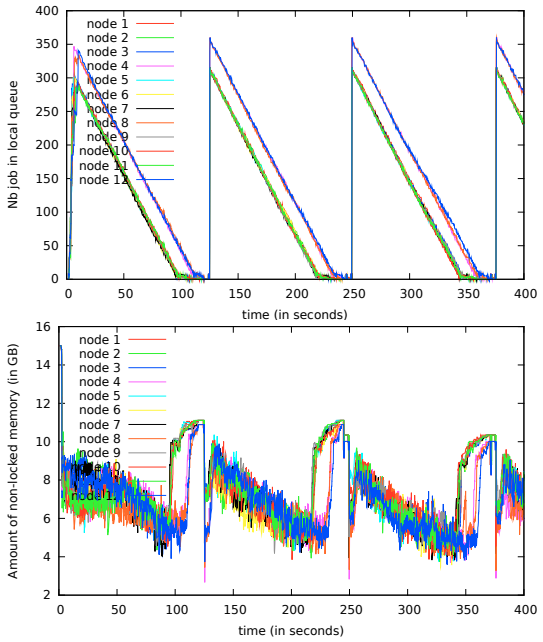


Fig. 5. Amount of free memory available and jobs in the local scheduler during an execution on 12 nodes.

#nodes	6	8	10	12	22	29
$t_{DAG-build}$ (s)	.7	.7	.7	.7	.8	.9
$t_{DAG-exec}$ (s)	53	53	56	56	56	61
t_{IO} (s)	879	631	528	473	242	263
t_{comp} (s)	936	701	580	521	270	277
t_{total} (s)	1136	842	698	641	454	481
$efficiency$	1.0	1.0	.98	.89	.68	.49
$n_{messages}$	2700	3608	4516	5424	9964	13142
V_{comm} (GB)	141	188	235	282	518	683

TABLE V
BREAK-DOWN OF THE EXECUTION TIME FOR THE FIRST 5 ITERATIONS OF THE $N_{\max}$=8 CASE USING DIFFERENT NUMBER OF COMPUTE NODES. THE LAST 2 ROWS SHOW THE LOAD ON THE INTERCONNECTION NETWORK.

present a break-down of the total execution time of $N_{\max}$=8 computations into different parts. Due to the increase in the number of tasks per node, task graph building and traversal take 5% to 10% of the total execution time. However, we still see an efficient overlapping of I/O and computations, leading to a good scaling up to large number of nodes. We achieve .68 parallel efficiency on 22 nodes and .49 parallel efficiency on 29 nodes (performance on 6 nodes is taken as the base case).

VII. Conclusions and Future Work

To deal with the increasing complexity of scientific applications, researchers have been using larger clusters which lead to new challenges for efficient usage of resources. In most data-intensive computations, larger clusters are not only used to solve problems faster, they are used because the problem size makes the use of smaller systems infeasible due to physical memory limitations. However, in large-scale data-intensive scientific applications, communication overheads also increase with the number of nodes. Hence such large clusters are hard to use efficiently for these purposes.

We think that the emergence of high-performance non-volatile storage devices presents a great opportunity. Distributed out-of-core algorithms running on these new devices would enable running large problems on small to moderate size clusters. However, developing an application that properly exploits the out-of-core execution capabilities is a difficult task. Therefore we introduced a middleware (DOoC) that allows one to easily develop such applications by relying on task-based decomposition and distributed immutable objects. Furthermore, we have developed a linear algebra frontend (LAF) that translates basic linear algebra primitives into tasks that can be executed on our DOoC middleware. We demonstrated the effectiveness of our distributed out-of-core framework by implementing an out-of-core eigensolver.

However, this work is only the first step toward our goal. Our initial results are encouraging, but not yet competitive with hand-optimized codes. We plan to improve the performance of our system in different and compatible ways. In order to reduce the communication overheads which hurt the scalability of our eigensolver, we are planning to develop better heuristics which would take into account the communication volume generated. Also the local scheduler can make better decisions by being aware of the source of the data (file system or network) as well as location of the data in memory (NUMA awareness). Supporting more linear algebra primitives will also lower the bar for the application scientists to use our proposed framework.

References

[1] P. Maris, J. P. Vary, P. Navratil, W. E. Ormand, H. Nam, and D. J. Dean, "Origin of the anomalous long lifetime of 14c," *Phys. Rev. Lett.*, vol. 106, no. 202502, 2011.

[2] P. Maris, A. M. Shirokov, and J. P. Vary, "Ab initio nuclear structure simulations: the speculative 14f nucleus," *Phys. rev. C*, vol. 81, no. 021301, 2010.

[3] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le, "Accelerating configuration interaction calculations for nuclear structure," in *Proc. of SuperComputing*, 2008.

[4] P. Maris, M. Sosonkina, J. P. Vary, E. G. Ng, and C. Yang, "Scaling of ab-initio nuclear physics calculations on multicore computer architectures," *Procedia CS*, vol. 1, no. 1, pp. 97–106, 2010.

[5] B. V. Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of NVRAM in data-intensive architectures: an evaluation," in *Proc. of IPDPS*, 2012.

[6] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM Journal on Scientific Computing*, vol. 23, no. 2, pp. 517–541, 2001.

[7] Z. Zhou, E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, J. P. Vary, and U. V. Catalyurek, "An out-of-core dataflow middleware to reduce the cost of large scale iterative solvers," in *Proc. of P2S2*, 2012.

[8] M. D. Beynon, T. Kurc, U. V. Catalyurek, C. Chang, A. Sussman, and J. Saltz, "Distributed processing of very large datasets with DataCutter," *Parallel Computing*, vol. 27, no. 11, pp. 1457–1478, Oct. 2001.

[9] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, 2012.

[11] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. of FAST'02*, 2002, pp. 231–244.