An Out-of-Core Task-based Middleware for Data-Intensive Scientific Computing

Erik Saule, Hasan Metin Aktulga, Chao Yang, Esmond G. Ng and Ümit V. Çatalyürek

1 Introduction

Petascale scientific computing, next-generation telescopes, high-throughput experiments, data-oriented business technologies and the Internet have been driving a rapid growth in data acquisition and generation. Analysis of large-scale datasets is likely to bring new breakthroughs in the academic and industrial world. These analyses typically require the use of large computer systems, such as those that can be found in data centers or high performance computing (HPC) facilities.

While the computing power of large computer systems that can enable timely and scalable data analysis has been increasing steadily for decades, their memory capacities have not been able to keep pace [1], see Fig. 1. As we move towards the future, this gap is anticipated to widen even further. The main reason for this trend is that it is not possible to meet the storage capacity and power consumption requirements of future machines using the DRAM technology. Non-volatile memory (NVM) solutions, on the other hand, feature much higher storage densities and lower

E. Saule (✉)
Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC 28223, USA
e-mail: esaule@uncc.edu

H. M. Aktulga · C. Yang · E. G. Ng
Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
e-mail: hmaktulga@lbl.gov

C. Yang
e-mail: cyang@lbl.gov

E. G. Ng
e-mail: egng@lbl.gov

Ü. V. Çatalyürek
Department of Biomedical Informatics, The Ohio State University, Columbus, OH 43210, USA
e-mail: umit@bmi.osu.edu

S. U. Khan, A. Y. Zomaya (eds.), *Handbook on Data Centers,*
DOI 10.1007/978-1-4939-2092-1_22

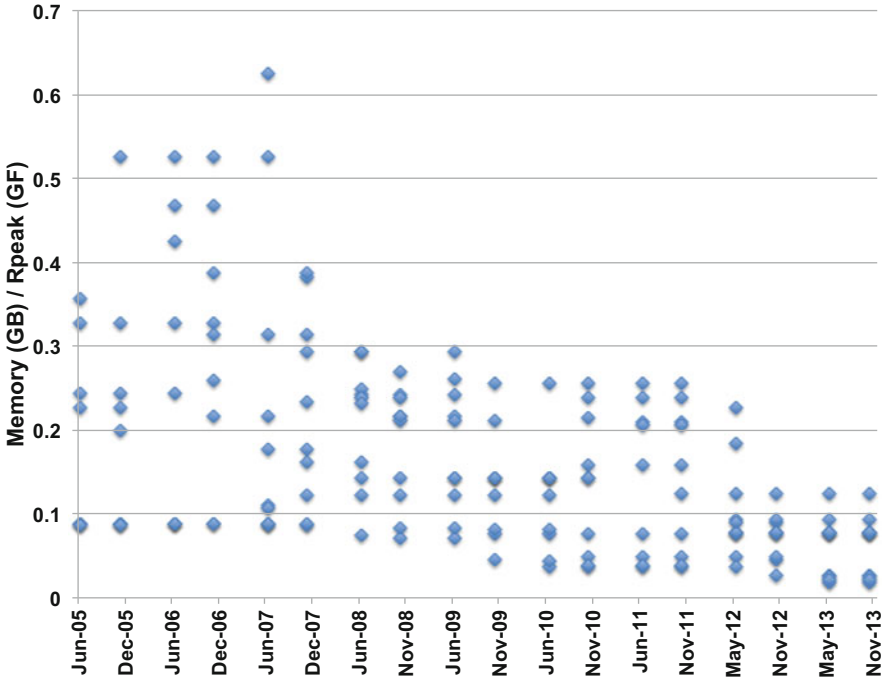

Fig. 1 Memory in gigabytes per gigaflop of computing power for the leading 10 supercomputers on the TOP500 list over the years

power requirements compared to DRAM. Therefore NVM technology will be one of the key enablers for future high end computing architectures.

In datacenters, NVM storages are experiencing a fast adoption rate due to the high bandwidth and low latency advantages that they provide over the traditional disk-based storage systems in the management and analysis of large datasets. Several NVM storage solutions from companies such as Fusion-IO, OCZ Technologies, HP and Seagate already exist in the marketplace. Initially, these NVM storages were used as mere disk replacements and they were connected to the compute resources through low performance interfaces such as SCSI or SATA. Nowadays, we increasingly see high performance NVM storages being connected through the PCI Express bus. As the technology improves, it is anticipated that NVM storages will take their place as a new layer in the memory hierarchy for datacenter systems [2].

NVM storages are already incorporated in today's HPC architectures that are designed to tackle challenging data-intensive problems. For example, the Gordon computer at the San Diego Supercomputing Center (SDSC) houses 300 TBs of flash memory storage in addition to 64 TBs of DRAM space. Trinity (Los Alamos) and NERSC-8 systems, which are planned for operation in 2015, will use flash memory based storages at a much greater scale. Their total flash memory storage capacity is expected to be on the order of 5–10 PBs, which corresponds to about 2–3 times the

total DRAM capacity on those systems. An important use case for the flash memory storage on the Trinity and NERSC-8 systems will be to provide a fast workspace for data-intensive applications. As we move towards the exascale era, NVM storages which are currently seen as fast disk alternatives only will be introduced as a new layer into the memory hierarchy in HPC systems as well. Non-Volatile Random Access Memory (NVRAM) is a main component of exascale computer architecture designs by AMD and IBM as part of their efforts in the DOE's Fast Forward program, [3, 4].

The drastic changes in system architecture will require rethinking systems software as well. Specifically, with improvements in hardware performance, software efficiency will become the next bottleneck. Scalable and efficient analytics on large computer systems require advanced parallel programming skills. However, most computational scientists and data scientist are not parallel programming experts. Besides the need for carefully organizing communication and computations in large scale applications, the need to manage data stored on NVM devices emerges in current architectures designed for data-intensive computing. This adds considerable complexity in code design and development.

Our vision is to increase the programmer productivity while still ensuring good performance and scalability by enabling the separation of computation and data movement. In our approach, the programmer can focus on the computational operations that he/she wants to apply to the sets of data and delegates the chore of data movement to the task-based data-flow middleware, DOoC (**D**istributed **O**ut-**o**f-**C**ore), that we describe in this chapter. DOoC is a runtime environment that determines and executes optimal data movement policies for systems with deep memory/storage hierarchies. Conceptually, in DOoC the entire computation is represented as a Directed Acyclic Graph (DAG), where an operation on a dataset corresponds to a vertex, the input data for the computational task is represented as an incoming edge to that vertex and the resulting data is represented as an outgoing edge of the vertex. Our runtime environment carefully considers the characteristics of the underlying memory/storage subsystem and the needs of the data-intensive applications that it supports to enable efficient execution of large-scale computations. The overall goal of our work is to provide an easy-to-use high-level application interface for data-intensive workloads, while providing efficient and scalable execution by orchestrating pipelined execution of computation, communication and I/O.

We have designed and implemented DOoC to be a generic middleware that can be used in a wide spectrum of applications in fields as diverse as graph mining, bioinformatics and scientific computing. A customizable frontend allows the application developer to interact with the DOoC framework through a simple programming interface. In this chapter, after giving an overview of the DOoC framework, we introduce the Linear Algebra Frontend (LAF) which is developed to enable the implementation of iterative numerical methods using DOoC. We present a case study on the implementation of a block eigensolver for the solution of large-scale eigenvalue problems arising in nuclear structure computations. We give detailed performance and scalability analysis for the resulting distributed out-of-core eigensolver on an experimental testbed equipped with NVM storages. We conclude our chapter with a discussion on the future work planned.

2 Related Work

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 [5], the complete task graph is generated before scheduling. However, in our system the task graph is generated dynamically on-the-fly. Two other middlewares are similar to our effort: StarPU [6] and PaRSEC [7]. They both have been recently used for sparse linear algebra [8].

StarPU [6] is a task-based middleware like DOoC. It has been used for both dense and sparse linear algebra. It is designed to take advantage of multicore systems with accelerators and has been ported to support multiple architectures such as CUDA devices, OpenCL devices, the IBM Cell processor and multicore CPUs. StarPU has no support for out-of-core processing. It also allows multiple copies of a data item to exist on multiple devices as long as they are identical copies. Once a modification is made on one copy, the other existing copies must be deallocated. Two recent developments in StarPU are the composability of StarPU applications [9] and the support for distributed memory computing using MPI [10].

PaRSEC (previously known as DAGuE [7]) has originally been designed for in-core, dense linear algebra computations. Recently, it has been used to perform sparse linear algebra operations [8]. It supports both accelerators and distributed memory computing. The highlight of PaRSEC is the use of Parametrized Task Graph [11] to store the task graph in a compact form to reduce the scheduling overheads and synchronizations [12].

Out-of-core algorithms for sparse numerical linear algebra applications involving large matrices have been an attractive research topic, especially back in the 90's. Toledo gives an excellent survey of such algorithms [13]. More recently, out-of-core direct solvers on a single node have been investigated for symmetric [14, 15] or asymmetric matrices [16, 17]. A parallel (but still single node) out-of-core multifrontal method has recently been developed [18] and recently improved to reduce the amount of I/O transfers [19]. Distributed out-of-core computations were considered to compute the steady state of Markov chains using Jacobi or Conjugate Gradient algorithms [20]. Also approximations to compute the Page Rank of a graph accessed from the disk has recently been proposed [21].

Another related area of work is the field of memory aware scheduling algorithms. Out-of-core computing relies on reusing available data as much as possible and minimizing the amount of data to transfer from the disk to perform the computation. Many works in scheduling are applicable to out-of-core algorithms. [22] studies the problem of scheduling independent tasks and DAGs onto a cluster to minimize both the makespan of the application and the memory consumption of the node with the most used memory. In this model, the assumption is that once memory is used it is never freed. This assumption can model either the cost of a reading from the disk or the space used on the disk by the tasks. This model is extended in [23] in the context of load balancing for file servers where the author investigates the use of replication of data items and their reallocation to better take the change in the load into account.

The previously described model uses memory as an abstract concept. Some other models attach actual piece of data to the computations and focus on assigning the data to a compute node in order to minimize the cost of off-node data accesses [24].

Other related scheduling problems are concerned with the execution of a task graph under memory pressure where data is deallocated once it is no longer used and the goal of the scheduler is to execute the application using the least amount of memory. This problem has historically been solved to schedule the execution of binary arithmetic trees in compilers with unitary space cost to minimize the amount of used registers [25]. Most of the work in the area is concerned with trees since it has been shown that the problem is NP-Complete on DAGs [26]. The problem of scheduling non-binary in-trees with arbitrary cost has been solved in polynomial time [27]. There also have been interest in the case where multiple chains need to be computed and a cache is available to store the result of some tasks removing the need to compute it. Unfortunately, this problem has also been proved NP-Complete, but some polynomial time approximation algorithms have been proposed for it [28].

Most of the work on memory pressured scheduling only consider the problem of minimizing the memory requirement in a sequential setting. But if the problem can not be solved in memory, then it becomes important to try to minimize the amount of I/O performed to compute the final solution. This problem is shown to be NP-Complete and heuristics have been proposed and tested on instances coming from multifrontal methods [29]. Also, the trade-off of memory and execution time of the execution of an in-tree on a parallel machine have recently been investigated in [30].

During the last decade there has been little interest in distributed memory out-of-core numerical linear algebra algorithms. We argue that the main reason has been the poor performance of these algorithms due to the high latency and low bandwidth associated with traditional disk-based storage systems. At this point, the emergence of clusters equipped with non-volatile NAND-flash memory based solid state drives (SSD) presents unique opportunities and this is exactly what we explore in this chapter.

3 An Out-of-Core Task-based Middleware

DOoC (**D**istributed **O**ut-**o**f-**C**ore) is a recently developed generalized middleware for distributed out-of-core computation and data analysis [31]. DOoC runs on top of DataCutter [32], which itself is a distributed, coarse-grain data-flow 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 the separation of the computations from the data movements and provides an efficient runtime system that orchestrates pipelined executions with computation and communication overlapping.

Figure 2 depicts the architectural overview of our proposed framework, which is composed of DOoC and LAF (**L**inear **A**lgebra **F**rontend) [33]. DOoC provides efficient execution of task graphs with given input and output data dependencies. In

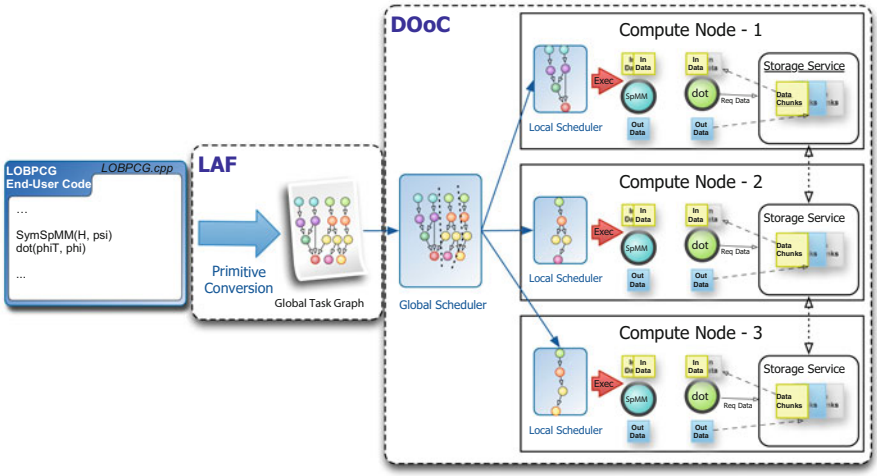

Fig. 2 Schematic overview of our framework Distributed Out-of-Core (DOoC) with Linear Algebra Frontend (LAF)

DOoC, task graphs and task codes need to be generated manually by the application developer. Since our focus in this chapter is on iterative eigensolvers for large-scale sparse matrices, we have designed and developed LAF, which we describe in more detail in Sect. 4. LAF customizes our framework for linear algebra computations by providing a high-level interface to application developers. It acts as a frontend that translates basic linear algebra primitives into global task graphs that can be executed by DOoC.

DOoC 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.

In DataCutter, which serves as the distributed data storage layer for DOoC, the *immutable object* paradigm is adopted. In immutable object paradigm, a given memory location can only be written once and can not be read before being written. This removes race conditions and the need for distributed memory coherency protocols (which are major concerns in similar systems with mutable objects such as Global Array [34]).

Below, we describe each component of the proposed framework in more detail.

3.1 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 Fig. 2). 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. 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 allocates a task onto the node where most of the input data is already located at. 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, too.

The local scheduler obtains regularly (default every 100 ms) from the storage service (which we describe in the next subsection) 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 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 there are tasks that are waiting for input data to be executed, the local scheduler determines the data to be prefetched by using the greedy algorithm presented in Algorithm 1 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. 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 memory space becomes available, prefetching is reinstantiated.

Algorithm 1: Task Ordering Algorithm.

```
AvailData ← storage().getAvailData()
availMem ← storage().getAvailMem()
Tasks ← global().getSchedulableTasks()
outOfMem ← False
PrefetchList ← ∅
while not outOfMem and not Task.empty() do
    for t ∈ Tasks do
        ToFetch ← input_data(t) - AvailData
        cost_t ← Σ_{d∈ToFetch} size_of (d)
    end
    t* = argmin_{t∈Tasks} cost_t
    ToAdd ← input_data(t*) - PrefetchList
    for d ∈ ToAdd do
        if size_of (d) > availMem then
            outOfMem ← True
        end
        else
            PrefetchList ← PrefetchList ∪ {d}
            AvailData ← AvailData ∪ {d}
            availMem ← availMem - size_of(d)
        end
    end
    Tasks ← Tasks − {t*}
end
return PrefetchList
```

3.2 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, it just induces 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 that node's local memory, it may be stored either on the permanent storage system or on the memory of 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 currently scheduled on the cores of 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 policy.

4 Linear Algebra Frontend (LAF)

The Linear Algebra Frontend (LAF) 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 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 1. Although not comprehensive, these operations are sufficient to implement various numerical methods for the solution of linear systems or eigenvalue problems that are widely used in scientific computing. The Conjugate Gradients, LOBPCG, Lanczos, and Page-Rank algorithms are among the examples that can be implemented using the primitives that currently exist in LAF.

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

Table 1 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

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	$a =< x, y >$

with a reduction operation reaches a threshold (default: 5), a local reduction task is dynamically created.

5 A Case Study: Block Iterative Eigensolver Using DOoC+LAF

In this section, we present a case study using our DOoC+LAF framework. We give the implementation details of a block eigensolver for the solution of large-scale eigenvalue problems arising in nuclear structure computations.

5.1 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:

$$H\psi = E\psi \tag{1}$$

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

In the Configuration Interaction (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

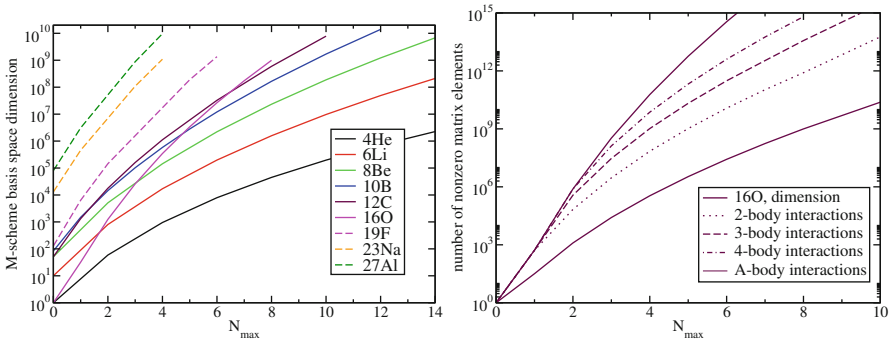

Fig. 3 The dimension and the number of non-zero matrix elements of the various nuclear Hamiltonian matrices

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 Fig. 3). 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 which can be a 2-body, 3-body or even a higher order interaction. The approach described above is implemented in the MFDn (Many Fermion Dynamics nuclei) code, which is a state-of-the-art CI code to study the properties of light nuclei with high precision [35–37]. In MFDn, a round-robin distribution of the many-body basis states to the processors is used to ensure a uniform distribution of the nonzero matrix elements in the $\hat{H}$ matrix. This way load imbalances among processors is reduced significantly [38].

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 [39]. 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 2 gives the pseudocode for a simplified version of the LOBPCG algorithm, assuming $M = I$.

Algorithm 2: 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 $i = 0, 1, \ldots,$ *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
end

5.2 Implementation Using 1D partitioning

Our first implementation of the out-of-core eigensolver is an implementation of the LOBPCG algorithm given in Algorithm 2 using the linear algebra primitives of the DOoC+LAF framework and using a one dimensional partitioning of the matrix. In this scheme, the matrix is cut into p bands of equal size $\frac{n}{p}$, and each band is of length $\frac{n}{2}$. The allocation of the parts of the matrix to each node is depicted in Fig. 4a. 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 a load balanced 1D decomposition of the SpMM operation.

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 Compressed Sparse Column (CSC) format, is less than the size of any vector block Ψ_i, for $1 \leq i \leq n_b$. Then Algorithm 1 orders the set of tasks on node p

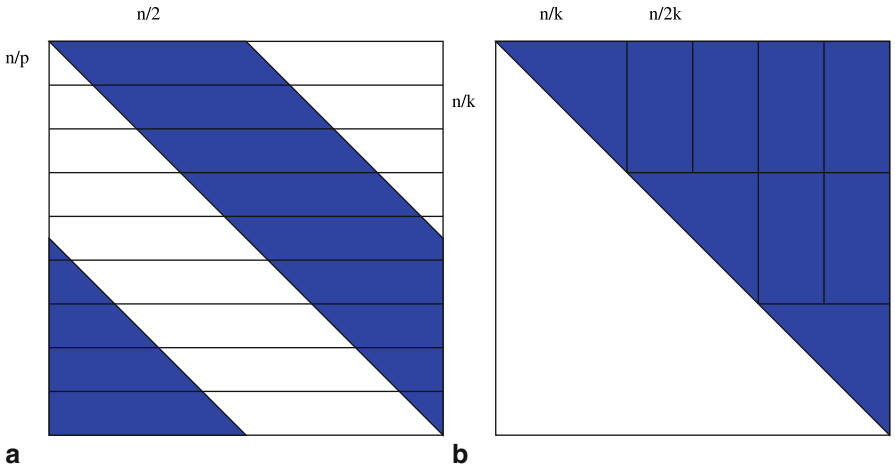

Fig. 4 Different partitioning of the matrix $\hat{H}$ on the processors. Notice that since the matrix is symetric, only half of it needs to be stored. **a** 1D decomposition on p nodes. **b** 2D decomposition on $p = k^2$ nodes

$\{\texttt{SymSpMM}(i, j) \mid rs_p \leq i \leq re_p \wedge 1 \leq j \leq n_b\}$ such that they are executed in a column-major order.

Proof Without loss of generality, let $\texttt{SymSpMM}(rs_p, j)$ 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 1 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. 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 $\texttt{dot}(Y_i, Y_i)$ and $\texttt{dot}(Y_i, \hat{H}Y_i)$, 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.

5.3 Implementation Using a 2D Partitioning

The 1D partitioning scheme, shown in Fig. 4a, requires that each node touches $n(\frac{1}{2} + \frac{1}{p})$ row/column. When the number of nodes p increases, the volume of communication will be proportional to the problem dimension, i.e., with $\frac{n}{2}$. This indicates a potential scalability bottleneck, as the number of nodes and problem dimensions increase together in order to solve larger problems.

One can partition the upper triangle of the matrix in two dimensions (2D) by using horizontal and vertical bands. Because the matrix is symmetric, a classical checkerboard partitioning would make the nodes responsible for the diagonal blocks processing half the non zeros of the other nodes. Therefore, we propose to split the non diagonal blocks in two so as to remove this problem. Such a partitioning is depicted in Fig. 4b and requires a number of nodes which is a square number $p = k^2$. Diagonal nodes touch only $\frac{n}{k}$ row/columns since the rows one touches are the same as the columns it touches. Meanwhile the non diagonal nodes touch $\frac{3n}{2k}$ row/columns. Since $p = k^2$, the communication volume will behave like $\frac{3n}{2\sqrt{p}}$ and is much better than the number of node increases than the 1D decomposition.

Notice also that improving the communication volumes is not the only interest of this 2D decomposition. Indeed when a processor touches a row or a column, not only it will perform communications, but also it needs to store the partial results. So a 2D decomposition will be necessary to allow to scale the computation to larger problems in terms of size of the matrix or number of vectors.

In term of implementation within the DOoC+LAF framework, there is no difference between a 1D decomposition of the work and a 2D decomposition of the work. It is sufficient to place the blocks of the matrix on the computing nodes that will process them. The framework will automatically add the appropriate communications.

6 Experiments

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

Table 2 General information on the testcase

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

Table 3 General information on the vector block sizes

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

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 [40]. 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. The truncation parameter $N_{\max} = 8$ is used. Some key properties of this testcase are summarized in Table 2. Since storage space is at premium for MFDn, matrix blocks are stored in single precision CSC format.

6.1 Practical Considerations

The number of eigenpairs to be computed is fixed at $nev = 8$ for our test-case. Table 3 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 5.1 GB for the $N_{\max} = 8$ case. 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 30.6 GB for the $N_{\max} = 8$ case, 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

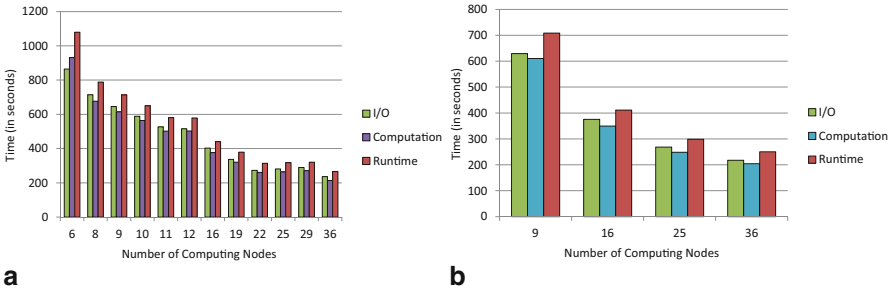

Fig. 5 Runtime of the application and time spent doing computations and I/O. The I/O and Computation mostly overlap. **a** 1D partitioning. **b** 2D partitioning

are on the order of tens of MBs (see the size of Ψ_i in Table 3), 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 $N_{\max} = 8$ computations is 6, respectively.

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, we report the timings from the first 5 iterations of the LOBPCG algorithm for a reliable performance evaluation.

Since all the computing nodes share the same file system, each node will read its data in different directory so as to provide data partitioning.

6.2 *Performance Results for* $N_{\max} = 8$

Figure 5 presents the runtime obtained when executing the application on different number of nodes. The figure also presents the time taken by the computations and by the I/O separately. These times varies on all computation and I/O threads. The figure reports the maximum value of all threads but the average value is fairly close to the maximum.

The first remark is that the difference between the Runtime and the maximum of I/O and computation is fairly small. This indicates that the computations and I/O are fairly well overlapped and that the design of our middleware is sound. The runtime decreases with an increase of the number of computing nodes. Though, the runtime is fairly stable after 20 nodes. This comes from a saturation of the GPFS after 20 computing nodes which draws 16 GB/s, the peak performance for the I/O nodes. This shows that the traditional organization of the cluster with I/O nodes on one side and compute node on the other one is not a scalable setup for the data-intensive clusters.

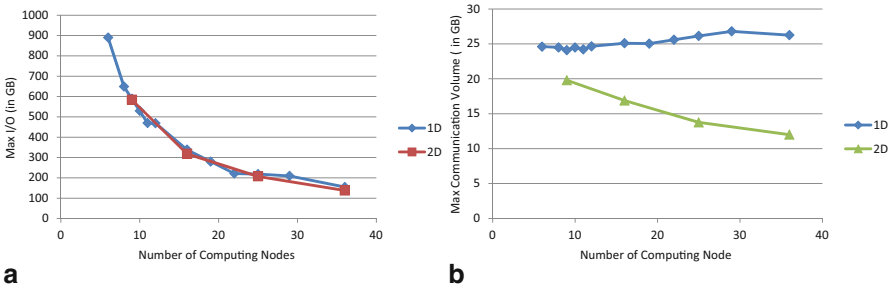

Fig. 6 Comparison of 1D and 2D partitioning. **a** Amount of GPFS I/O. **b** Amount of Inter node communication

Indeed, within a single I/O node, we get a bandwidth from the disks to the memory of about 2 GB/s. Yet to be able to reach such bandwidth from the application more than twice the amount of compute nodes are required.

We can see on Fig. 5 that there is little difference in total runtime between using 1D and 2D decomposition. The only existing difference is entirely explained by the difference of I/O performed by the 1D and 2D decomposition. The differences between 1D and 2D decompositions are presented in Fig. 6. One can see that the amount of GPFS I/O (Fig. 6a) is slightly lower for the 2D decomposition. Indeed, when 2D decomposition is used, less memory is used for storing the intermediate values of the multiplications which leaves more memory available for caching the data from the matrix (as explained in Sect. 5.3). Another interest of the 2D decomposition lies in the amount of communication performed by each node involved in the computation which is depicted in Fig. 6b. With a 1D decomposition, each processor transfers a whole Ψ vector at each iteration. Leading to a communication volume (per node) constant when the number of nodes increases. Meanwhile the communication volution when using a 2D decomposition decreases when the number of node increases. This confirms the analysis of Sect. 5.3 that 2D decomposition is more scalable than a 1D decomposition.

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. The 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 Fig. 7a. Here we plot the first 3 iterations of the $N_{\max} = 8$ case on 12 nodes with 1D decomposition. 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

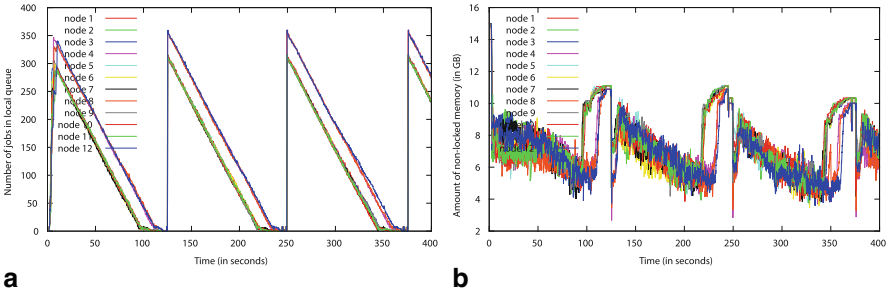

Fig. 7 Amount of free memory available and jobs in the local scheduler during an execution on 12 nodes with 1D decomposition

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.

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.

Figure 7b shows the amount of available memory as the 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.

7 Conclusions

Adaption of NVM-based memory in future HPC architectures and data centers will only increase with time. Efficient use of such, multilevel memory hierarchies will require advanced parallel programming skils. Here, we presented an attempt to relieve such burden from programmer, by providing a domain specific frontend that uses already familiar Basic Linear Algebra Subprograms (BLAS)-like application interface and leverages a capable task-based runtime system that will take care of

efficient orchestration of the execution of use applications. Specifically, we have presented early results of our out-of-core task-based runtime system, (**D**istributed **O**ut-**o**f-**C**ore (DOoC), together with a specialized frontend, Linear Algebra Frontend (LAF), which is developed to enable the implementation of iterative numerical methods using DOoC. Although our out-of-core runtime system generic and could work with any storage system, existance of high-bandwith, low-latency storage system that are based on non-volatile memory makes it feasible to execute larger problems that will not fit into physical RAM memory of the compute nodes. Our results shows that LAF+DOoC pushes the hardware limitations of the underlying testbed we have carried our experiments, while providing an extremely easy application interface.

We argue that in the future systems by co-locating SSD storages with computation [41], one can further optimize the out-of-core execution further. Our task-based runtime system DOoC is well positioned to take advantage of such hardware changes without requiring the rewrite of application program.

References

1. P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the new normal in computer architecture," *Computing in Science Engineering*, vol. PP, no. 99, pp. 1–1, 2013.
2. P. Ranganathan and J. Chang, "(Re)designing data-centric data centers," *Micro, IEEE*, vol. 32, no. 1, pp. 66–70, 2012.
3. E. Barragy, B. Brantley, S. Gurumurthi, M. Ignatowski, N. Jayasena, A. Lee, G. Loh, S. Manne, M. O'Connor, P. Popescu, S. Reinhardt, and M. Schulte, "Amd's fastforward extreme-scale computing processor and memory research," in *US DOE Exascale Research Conference*, Arlington, VA, USA, Oct. 2012.
4. R. Nair, J. Moreno, and D. Joseph, "Advanced memory concepts for exascale systems," in *US DOE Exascale Research Conference*, Arlington, VA, USA, Oct. 2012.
5. Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999.
6. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.
7. 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, pp. 37–51, 2012.
8. G. Bosilca, M. Faverge, X. Lacoste, I. Yamazaki, and P. Ramet, "Toward a supernodal sparse direct solver over DAG runtimes," in *Proceedings of PMAA'2012*, London, UK, Jun. 2012.
9. A.-E. Hugo, A. Guermouche, R. Namyst, and P.-A. Wacrenier, "Composing multiple StarPU applications over heterogeneous machines: a supervised approach," in *Third International Workshop on Accelerators and Hybrid Exascale Systems*, Boston, États-Unis, May 2013.
10. C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, "StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators," in *EuroMPI 2012*, ser. LNCS, S. B. Jesper Larsson Träff and J. Dongarra, Eds., vol. 7490. Springer, Sep. 2012, poster Session.
11. M. Cosnard and M. Loi, "Automatic task graph genera tion techniques," *Parallel Processing Letters*, vol. 5, no. 4, p. 527–538, 1995.
12. M. Cosnard, E. Jeannot, and T. Yang, "Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors," in *Proc. ICPP*, 1999.

13. S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," in *External memory algorithms*, J. M. Abello and J. S. Vitter, Eds. Boston, MA, USA: American Mathematical Society, 1999, pp. 161–179.
14. J. K. Reid and J. A. Scott, "An out-of-core sparse cholesky solver," *ACM Trans. Math. Softw.*, vol. 36, no. 2, 2009.
15. V. Rotkin and S. Toledo, "The design and implementation of a new out-of-core sparse cholesky factorization method," *ACM Trans. Math. Softw.*, vol. 30, no. 1, pp. 19–46, 2004.
16. P. R. Amestoy, I. S. Duff, Y. Robert, F.-H. Rouet, and B. Ucar, "On computing inverse entries of a sparse matrix in an out-of-core environment," CERFACS, Tech. Rep. TR/PA/10/59, 2010.
17. J. A. Scott, "Scaling and pivoting in an out-of-core sparse direct solver," *ACM Trans. Math. Softw.*, vol. 37, no. 2, 2010.
18. E. Agullo, A. Guermouche, and J.-Y. L'Excellent, "A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory," *Parallel Computing, Special Issue on Parallel Matrix Algorithms*, no. 6–8, 2008.
19. E. Agullo, A. Guermouche, and J.-Y. L'Excellent, "Reducing the I/O Volume in Sparse Out-of-core Multifrontal Methods," *SIAM Journal on Scientific Computing*, no. 6, 2010.
20. W. J. Knottenbelt and P. G. Harrison, "Distributed disk-based solution techniques for large markov models," in *Proc. of Numerical Solution of Markov Chains*, 1999.
21. Y.-Y. Chen, Q. Gan, and T. Suel, "Local methods for estimating pagerank values," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, ser. CIKM '04. New York, NY, USA: ACM, 2004, pp. 381–389.
22. E. Saule, P.-F. Dutot, and G. Mounié, "Scheduling With Storage Constraints," in *Proc of IPDPS'08*, Apr. 2008, conference, acceptance rate: 25.6%.
23. S. S. Tse, "Online bicriteria load balancing using object reallocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 379–388, 2009.
24. Ü. V. Çatalyürek, K. Kaya, and B. Uçar, "Integrated data placement and task assignment for scientific workflows in clouds," in *The Fourth International Workshop on Data Intensive Distributed Computing (DIDC 2011), in conjunction with the 20th International Symposium on High Performance Distributed Computing (HPDC 2011)*, Jun 2011.
25. R. Sethi, "Pebble games for studying storage sharing." *Theor. Comput. Sci.*, vol. 19, pp. 69–84, 1982.
26. S. Biswas and S. Kannan, "Minimizing space usage in evaluation of expression trees," in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, P. Thiagarajan, Ed. Springer Berlin Heidelberg, 1995, vol. 1026, pp. 377–390.
27. C.-C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan, "Memory-optimal evaluation of expression trees involving large objects," in *High Performance Computing – HiPC'99*, ser. Lecture Notes in Computer Science, P. Banerjee, V. Prasanna, and B. Sinha, Eds. Springer Berlin Heidelberg, 1999, vol. 1745, pp. 103–110.
28. V. Rehn-Sonigo, D. Trystram, F. Wagner, H. Xu, and G. Zhang, "Offline scheduling of multi-threaded request streams on a caching server," in *IPDPS*, 2011, pp. 1167–1176.
29. M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar, "On optimal tree traversals for sparse matrix factorization," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 556–567.
30. L. Marchal, O. Sinnen, and F. Vivien, "Scheduling tree-shaped task graphs to minimize memory and makespan," INRIA, Rapport de recherche RR-8082, Oct. 2012.
31. Z. Zhou, E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, J. P. Vary, and Ü. V. Çatalyürek, "An out-of-core dataflow middleware to reduce the cost of large scale iterative solvers," in *2012 International Conference on Parallel Processing (ICPP) Workshops, Fifth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, Sep 2012.
32. M. D. Beynon, T. Kurc, Ü. V. Çatalyürek, 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.

33. Z. Zhou, E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, J. P. Vary, and Ü. V. Çatalyürek, "An out-of-core eigensolver on SSD-equipped clusters," in *Proc. of IEEE Cluster*, Sep. 2012.
34. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, pp. 203–231, 2006.
35. P. Maris, H. M. Aktulga, M. A. Caprio, Ü. V. Çatalyürek, E. G. Ng, D. Oryspayev, H. Potter, E. Saule, M. Sosonkina, J. P. Vary *et al.*, "Large-scale ab initio configuration interaction calculations for light nuclei," *Journal of Physics: Conference Series*, vol. 403, no. 1, p. 012019, 2012.
36. P. Maris, H. M. Aktulga, S. Binder, A. Calci, Ü. V. Çatalyürek, J. Langhammer, E. Ng, E. Saule, R. Roth, J. P. Vary, and C. Yang, "No-Core CI calculations for light nuclei with chiral 2- and 3-body forces," *Journal of Physics: Conference Series*, vol. 454, no. 1, p. 012063, 2013.
37. H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of a symmetric iterative eigensolver for multi-core platforms," *Concurrency and Computation: Practice and Experience*, p. in press, 2013.
38. 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 SC08*, 2008.
39. 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.
40. 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.
41. M. Jung, E. H. W. III, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, Ü. V. Çatalyürek, and M. Kandemir, "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *Proc. of Conference on High Performance Computing Networking, Storage and Analysis (SC '13)*, Nov 2013.