An Out-Of-Core Dataflow Middleware to Reduce the Cost of Large Scale Iterative Solvers

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
†Dept. of Electrical and Computer Engineering, 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 and Astronomy, Iowa State University Ames, IA 50011, USA

***Abstract*—The emergence of high performance computing (HPC) platforms equipped with solid state drives (SSD) presents an opportunity to dramatically increase the efficiency of out-of-core numerical linear algebra computations. In this paper, we explore the advantages and challenges associated with performing sparse matrix vector multiplications (SpMV) on a small SSD testbed. Such an endeavor requires programming abstractions that ease implementation, while enabling an efficient usage of the resources in the testbed. For this purpose, we adopt a task-based out-of-core programming model on top of a dataflow middleware based on the filter stream programming model. We compare the performance of the resulting out-of-core iterated SpMV procedure running on the SSD testbed to the performance of an in-core implementation on a multi-core cluster for solving large-scale eigenvalue problems. Preliminary experiments indicate that the out-of-core implementation on the SSD testbed can compete with an in-core implementation in terms of the total CPU-hour cost. We conclude with some architectural design suggestions that can enable numerical linear algebra computations in general to be carried out with high efficiency on SSD-equipped platforms.**

I. Introduction

Out-of-core algorithms for efficiently solving large systems of linear equations or computing eigenvalues of large matrices have been an attractive research topic, especially back in the 90's. Toledo gives an excellent survey of such algorithms [1]. More recently, out-of-core direct solvers on a single node have been investigated for symmetric [2], [3] or asymmetric matrices [4], [5]. Distributed out-of-core computations was considered to compute the steady state of Markov chains using Jacobi or Conjugate Gradient algorithms [6]. However, during the last decade there has been little to no interest in parallel 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 (see Fig. 1). As we move away from registers to cache, to DRAM and finally to hard-disk drive (HDD), we see a steady increase of roughly 3 orders of magnitude in storage capacity between layers. Similarly, data access latencies increase at the rate of an order of magnitude between layers until we hit the "latency gap" between the DRAM and HDD. Typically, access latency of DRAM is about 100 CPU cycles, whereas this latency can grow up to 10,000 cycles or more when we want to access

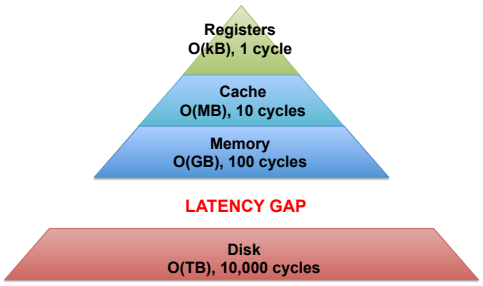


Fig. 1. The memory heirarchy. Storage capacities and access latency times of various layers of the memory hierarchy. **UVC: will we add BW to figure?**

data stored on disk. Likewise, while the bandwidth between DRAM and CPU is on the order of tens of GBs, the peak bandwidth from HDD through the modern SATA interface is on the order of hundreds of MBs only.

A common solution to the low bandwidths and high latencies associated with disk-based storage systems has been the usage of the distributed memory on today's high performance computing (HPC) platforms as storage space, and access data on other nodes through the interconnection network. The challenge of keeping track of which data is stored where and how to access it puts a major burden on the application programmer. While partitioned global address space (PGAS) languages aim to reduce this burden [7], [8], in data-intensive applications communication and synchronization overheads due to the use of a large number of processors may be prohibitive.

At this point, 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 (IOPS) and sustained read/write bandwidths compared to HDDs. For example, the recently released OCZ Z-Drive R4 CloudServ Series PCIe-based flash storage card can achieve up to 1.4 M IOPS and 6 GB/s sustained read/write bandwidth. SSDs also consume less energy, have excellent mean time between failure rates, and they can withstand extreme shock, vibration and temperature ranges [9]. These factors make SSDs attractive for a wide range of purposes, such as mobile

computing (*i.e.*, cell phones), business solutions (*i.e.*, database applications), cloud computing (*i.e.*, large data centers), military industry (*i.e.*, mission-critical applications), and HPC (*i.e.*, data-intensive applications).

HPC platforms equipped with SSD storage (such as Gordon at SDSC[1]) can help alleviate the communication and synchronization overheads in large-scale data-intensive applications.In most applications, the entire data-set does not have to be simultaneously resident on the DRAM. Consequently, the computation can be carried out on a relatively smaller number of processors, or even less, due to possible reduction in some duplicated data. This, in turn, could help reduce significant overheads due to communication and synchronization on large clusters.

In this paper, we investigate the advantages and challenges associated with performing large-scale sparse matrix vector multiplications (SpMV) on a relatively small testbed equipped with flash-memory based SSD cards. SpMV is the key ingredient of several algorithms for solving large systems of linear equations and large-scale eigenvalue problems. Traditionally, numerical linear algebra computations are not viewed as data-intensive applications. However, in certain cases (such as the *ab initio* nuclear structure calculations that we describe in Section II), extremely large sparse matrices (with billions of columns and trillions of non-zero elements) may arise and sparse matrix computations are notorious for their demand on memory system. For efficient utilization of the high-performance storage system, we adopt a dataflow middleware, named DataCutter [10], and develop a system called DOoC (short for Distributed Out-of-Core) for executing DAGs of tasks in an out-of-core fashion (see Section III). In Section V, we study the the performance of DOoC for iterated SpMV computations. We show that DOoC is able to exploit nearly the peak performance of the SSD-equipped testbed. The results are also compared to those of an in-core implementation on a multi-core cluster for solving large-scale eigenvalue problems. In Section VI, we try to identify the source of bottlenecks affecting the performance of our out-of-core SpMV implementation. Finally, we conclude with some architectural design suggestions that can enable numerical linear algebra computations in general to be carried out efficiently on SSD-equipped platforms.

II. Eigenvalue Problem in the Configuration Interaction Model

In this section, we describe the characteristics of the matrices associated with nuclear structure calculations, and how severe communication overheads arise in large-scale in-core calculations.

The key problem to be solved in nuclear structure calculations is the nuclear many-body Schrödinger's equation $H\psi = E\psi$, where ψ is a many-body wavefunction and H is a nuclear many-body Hamiltonian. In the Configuration Interaction approach (CI), 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, the Schrödinger's equation becomes a finite-dimensional eigenvalue problem, where we seek the lowest eigenvalues (energies) and their associated eigenvectors (wave functions). The total number of many-body states or the dimension of $\hat{H}$ in our adopted harmonic oscillator (HO) basis, which we denote by D, is controlled by the number of particles A, and the truncation parameter $N_{\max}$, which is the maximum number of HO quanta above the minimum for that nucleus. Higher $N_{\max}$ values yield more accurate results for the same nucleus, but at the expense of an exponential growth in the dimensions of $\hat{H}$. The sparsity of $\hat{H}$ is determined by the interaction potential used. We use a 2-body interaction potential for the calculations presented in this paper, which means an entry $\hat{H}_{ij}$ of the Hamiltonian will be non-zero only when the number of different single-particle states corresponding to row i and column j of $\hat{H}$ is at most 2.

MFDn (Many Fermion Dynamics for nuclear structure) code [11], [12] developed by Vary *et al.* is currently one of the most advanced codes used in *ab initio* nuclear structure calculations. Due to the large dimension and the sparsity of $\hat{H}$, in MFDn the Lanczos algorithm is preferred. Applying a k-step Lanczos procedure to the matrix $\hat{H}$, where $k < D$, and a random initial starting vector x yields an orthogonal set of Lanczos vectors spanning the k+1 dimensional Krylov subspace of $x, \hat{H}x, \hat{H}^2x, \ldots, \hat{H}^kx$. Projecting $\hat{H}$ into this basis space allows us to obtain approximations to the desired eigenvalues of $\hat{H}$ by solving a much smaller problem. The appeal of the Lanczos method is in its computational simplicity. The computational cost of the method is dominated by the associated sparse matrix vector multiplications (SpMV) and (to a smaller extent) orthonormalization of Lanczos vectors.

Table I gives problem characteristics for some calculations on the atomic nucleus is ^{10}B (5 protons, 5 neutrons) with different truncation parameters $N_{\max}$ and M_j (total magnetic quantum number) values. Test cases were selected such that each calculation is performed on the minimum number of processors that matches the memory needs of the calculation. All computations are carried out on the Hopper supercomputer, which is a Cray XE6 machine at the National Energy Research Scientific Computing Center[2] (NERSC). Each compute node on Hopper contains 2 twelve-core AMD "MagnyCours" processors (24 cores per node) with 32 GBs of memory (slightly more than 1 GB of memory per core). Hopper uses the Cray "Gemini" interconnect for internode communication. The entire Hopper machine has over 150,000 cores.

In Table II, we present the performance details of running MFDn on Hopper. As the dimension and the number of non-zero elements of $\hat{H}$ increase, it becomes necessary to use tens

[1] http://www.sdsc.edu/supercomputing/gordon/

[2] http://www.nersc.gov

TABLE I

MATRIX DIMENSIONS D AND NUMBER OF NON-ZERO MATRIX ELEMENTS nnz OF THE HAMILTONIAN $\hat{H}$ ASSOCIATED WITH NUCLEAR STRUCTURE CALCULATIONS OF ^{10}B USING DIFFERENT PARAMETER PAIRS ($N_{\max}$,M_j) AND THE NUMBER OF PROCESSORS n_{p} REQUIRED TO RUN EACH EXPERIMENTS. ALSO SHOWN ARE THE AVERAGE SIZES OF THE LOCAL LANCZOS VECTORS v_{local} AND LOCAL $\hat{H}$ MATRICES $\hat{H}_{local}$.

Test Name	($N_{\max}$, M_j)	$D(\hat{H})$	$nnz(\hat{H})$	n_{p}	avg. size of v_{local}	avg. size of $\hat{H}_{local}$
test$_{276}$	(7,0)	4.66×10^7	2.81×10^{10}	276	8.8 MB	880 MB
test$_{1128}$	(8,1)	1.60×10^8	1.24×10^{11}	1,128	13.6 MB	880 MB
test$_{4560}$	(9,2)	4.82×10^8	4.62×10^{11}	4,560	20.4 MB	800 MB
test$_{18336}$	(10,3)	1.30×10^9	1.51×10^{12}	18,336	27.2 MB	750 MB

TABLE II

PERFORMANCE OF 99 LANCZOS ITERATIONS DURING THE NUCLEAR STRUCTURE CALCULATIONS OF ^{10}B ON HOPPER USING MFDN (VERSION13, BETA02), THE CURRENT RELEASE VERSION, SINGLE-THREADED RUNS.

Stats	test$_{276}$	test$_{1128}$	test$_{4560}$	test$_{18336}$
t_{total} (sec)	244	543	759	1870
$t_{\mathrm{comm}}/t_{\mathrm{total}}$ as %	34%	60%	67%	86%
CPU cost per iter. (hours)	0.19	1.72	9.70	96.2

of thousands of cores so that the entire $\hat{H}$ matrix can fit into the distributed memory. This brings significant communication and synchronization overheads associated with the distribution and summing up of Lanczos vectors during both SpMV and orthogonalization phases, resulting in an inefficient utilization of resources. At 18,336 cores, communication overhead during Lanczos iterations is prohibitively high (86% of the total execution time). This situation makes efficient calculations of heavier nuclei out of reach, such as ^{14}Carbon with $N_{\max}$=10 and M_j=0, where the amount of memory required to store the $\hat{H}$ matrix together with the eigenvectors is estimated to take up the entire 200 TBs of memory available on Hopper.

III. DOOC: A DISTRIBUTED DATA STORAGE AND SCHEDULER WITH OUT-OF-CORE CAPABILITIES

We propose storing the Hamiltonian matrix on non-volatile storages instead of storing it on the distributed memory of nodes. Non-volatile storages are typically much larger than the DRAM memory allowing to reduce the number of nodes required to store the matrix. We expect that by using PCIe-based high performance flash memory SSDs, we can obtain significantly lower data access latencies compared to traditional non-volatile storage devices such as HDDs. Also the high sustained read/write bandwidths of individual SSDs can potentially provide enough aggregated I/O bandwidths using an array of SSDs to reach valuable tradeoffs between execution time and total CPU-hours used for a large-scale computation.

To demonstrate the feasibility of this approach, in this work, we have designed and developed DOoC, a distributed-memory task-based runtime system with data-dependency and out-of-core capabilities. The overall architecture of our middleware is shown on Figure 2. We build our system on top of DataCutter [10], which is an easy to program dataflow middleware that naturally supports heterogeneous systems. Two major components of our system are *distributed storage layer*, and *hierarchical data-aware task scheduler*. Below, we briefly discuss DataCutter, and the two major components of DOoC.

A. DataCutter

DataCutter [10] sits on top of the operating systems of the involved computational nodes and provides the dataflow interface abstraction to the application through the use of MPI. DataCutter implements computations as a set of components, referred to as *filters*, that exchange data through logical *streams*. A stream denotes a uni-directional data flow from some filters (*i.e.*, the producers) to others (*i.e.*, the consumers). Data flows along these streams in untyped data-buffers in order to minimize various system overheads. A layout is a filter ontology which describes the set of application tasks, streams, and the connections required for the computation. In the implementation of the filter-stream programming model, the key job left to application developers is writing the filter functions and determining the filter and stream layout. Even on a heterogeneous cluster, DataCutter can hide all architecture-specific details and provide a single data-structure interface. A filter can be replicable, if it is stateless, allowing simple data-parallelism that can be managed by the runtime system. Similarly, the runtime system can easily provide task- and pipelined-parallelism by appropriately scheduling computations, such as either running two independent tasks concurrently (task-parallelism), or running two dependent tasks on different data items (pipe-lined parallelism).

B. Distributed data storage layer

A distributed-memory data storage layer allows any computational task (*i.e.*, filter) to access data stored on any node. It supports prefetching, automatic memory management and out-of-core operations. It provides functionality similar to Global Array [13], which provides a Partitioned Global Address Space semantic to a physically distributed array and can be used in an out-of-core mode using Disk Resident Arrays. However, Global Array needs complicated coherency protocols, locks and synchronizations to resolve concurrent write accesses. Our technique relies on *immutable arrays* which alleviates the need for a complex communication protocol.

In our current prototype, the storage subsystem exposes the data to the filters as one dimensional arrays. A filter can *request* the access to an *interval* of an array either using *read* permission or *write* permission. 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

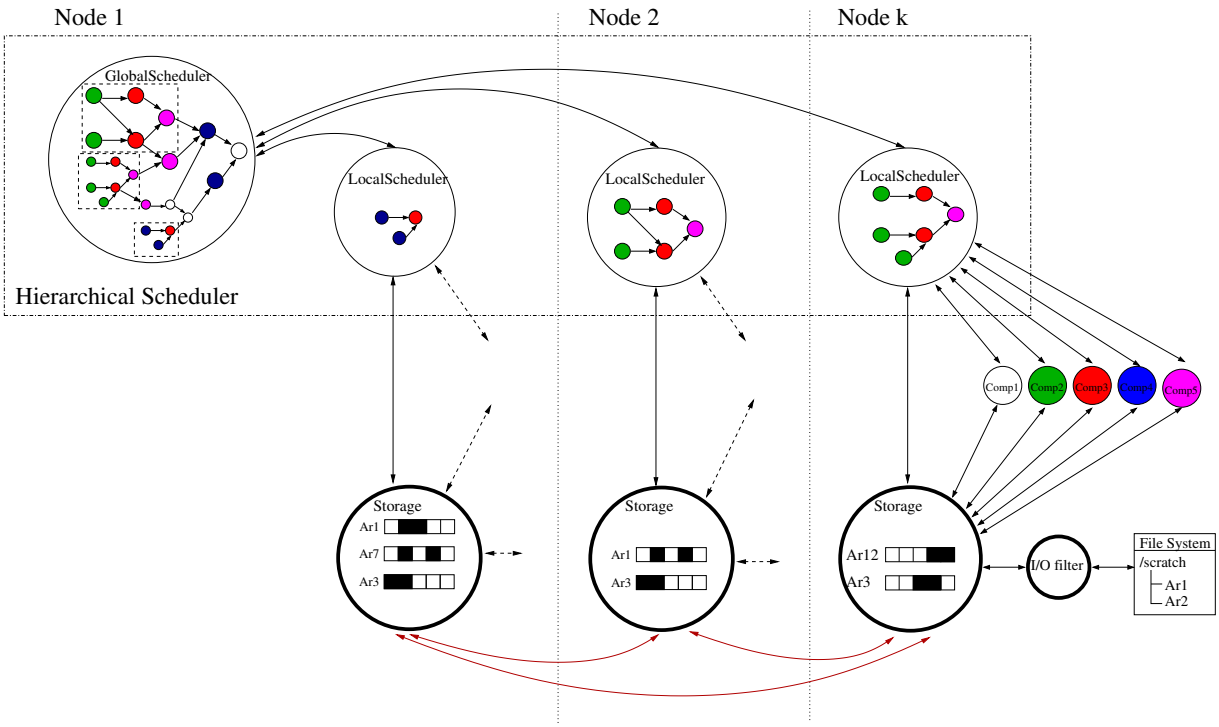


Fig. 2. DOoC architecture. The storage system is distributed on each node with complete peer-to-peer connections between them. Each local storage filter uses an I/O filter to interact with the file system. The hierarchical scheduler is composed of a local scheduler on each node, controlled by a global scheduler located on the first node.

the need for distributed memory coherency protocols (which are major concerns in similar systems with mutable objects such as Global Array [13]). Once a filter no longer needs an interval it should *release* it. For read operations, the storage subsystem guarantees that the data are available until the interval is released. For write operations, the data become available for being read by other filters only after the interval is released.

Arrays can be of arbitrary size, but they are structured in blocks. If one needs to access data that span across multiple blocks, it is required to use one interval per block. The data within a block are stored contiguously in memory ensuring that once obtained it can be accessed as fast as possible. Of course, one can easily build an abstraction that allows to access memory independently of the block it is stored in, trading performance for semantic simplicity.

The storage subsystem provides interface for prefetching some intervals, create some new arrays, delete existing arrays and obtain a map of which part of the arrays are currently available in the storage subsystem. Notice that the implementation in DataCutter is achieved by making the storage subsystem a specific filter and all filters that need to interact with the storage have a bidirectional link to it. This allows all the interactions with the storage layer to be asynchronous.

Internal protocol: The storage layer is implemented by deploying a storage filter on each compute node of the distributed system. The storage layer is implemented to be as asynchronous as possible. When a request is received, either the storage has all the information to answer it and it replies immediately, or it logs the request and replies back when all the relevant information becomes available. When a data interval which is not contained in the storage is requested, since global mapping (of which data is stored where) is not replicated on each node but instead partitioned, the storage asks the storage filter on a randomly selected compute node for this interval. To avoid asking for an interval multiple times, the storage keeps track of which interval it has requested from other computing nodes.

The storage supports an out-of-core mode of operation. A directory in the filesystem is used by the storage filter as its scratch memory. Upon start of the system, the storage looks for files in that directory and records the name of the arrays as well as their sizes. All reading of the data stored on the filesystem are performed implicitly: when an interval of data, which is not in the memory, is requested it is read from the filesystem. However, the write operations are performed explicitly upon request of a filter. Interactions with the filesystem (both read and write) are performed by a separate I/O filter. The I/O filter is only connected to the storage filter and allows the interactions with the file system to be completely asynchronous. There should be as many I/O filters as is necessary to efficiently use the parallelism contained in the I/O subsystem of the machine (most likely, the number of I/O controllers).

Eventually, the allocation of a block will exceed the amount of memory available in the node (which is a parameter of

the storage subsystem) and will trigger a memory reclaiming procedure. This is simply achieved by reference counting. The accesses to a given block of an array by filters are counted so that the storage can know whether a block is currently in use or not. When reclaiming memory, the storage reclaims blocks that are stored on the disk of any node and which are not currently used according to the Least Recently Used policy. Of course, explicit memory management can also be directly provided by the programmer.

C. A hierarchical data-aware scheduler

DOoC features a hierarchical data-aware task scheduler which efficiently distributes the computations to the computing filters. In our prototype system, the hierarchy is composed of two levels: *global scheduler* and *local scheduler*. At the coarse level, global scheduler allocates tasks to the computing nodes which have the capabilities to process them. At the fine level, local scheduler decomposes the tasks to expose more parallelism when necessary, and reorders the tasks to minimize the cost of memory transfers. One can draw similarities between our approach and other approaches that uses Directed Acyclic Graphs (DAG) to model computational dependencies, such as DAGuE [14] which targets in-core, dense linear algebra computations. Our system is designed for efficient, distributed memory, out-of-core execution of data-intensive applications.

The global scheduler is responsible for distributing the tasks across the nodes. Its input is the list of computations to be executed. Each computation takes some data as an input and outputs some data. Each data is a complete array that is (or will be) stored within the storage layer. The input and output data information is used to derive a DAG of the tasks. The global scheduler is responsible for sending parts of the DAG to compute nodes which will process them. The global scheduler currently uses the following simple, affinity-based, heuristic to fulfill this responsibility. Tasks are sent to the compute nodes which host most of the data required to process them. Exposing the application as a DAG of tasks enables the middleware to perform smart scheduling decisions which yield a more efficient execution. (See Section IV for an example.)

To execute the computations efficiently, each compute node has its own local scheduler which is responsible for ensuring that computations are performed in a correct order and tasks are completed as fast as possible. The local scheduler on each node receives tasks from the global scheduler, and splits them (if possible) to match the parallelism available on the node. All tasks that do not have any unprocessed predecessors are marked as ready. The local scheduler periodically queries the state of the storage to know which data are available in memory and which are not. When a computing filter is free, a task which is ready and whose data input are available in memory is sent to the computing filter. The local scheduler makes sure that there are a given number of ready tasks whose data are in memory by sending sufficient prefetch requests to the storage layer.

One can certainly question the efficiency of the scheduler for an arbitrary application. The theoretical problem the scheduler has to solve is a generic variant of the caching problem. This problem is NP-Hard and it is difficult to find a good approximate solution in the worst case, even if the DAG is only composed of chains [15]. However, the pathological case for most policies do not appear in practice: they typically appear only when the size of the task and their duration follow some maliciously crafted sequence.

IV. A use case scenario: Iterated sparse matrix vector multiplication

We evaluate the quality of our out-of-core dataflow middleware DOoC described above by implementing a distributed sparse matrix vector multiplication (SpMV), $y = Ax$, which is actually a key ingredient of several iterative algorithms in sparse linear algebra (as is the case in MFDn, see Section II). We describe in this section the details of the implementation and the impact of DOoC on performance at an abstract level.

The A matrix is assumed to be too large to fit into the distributed memory of compute nodes. Therefore it is partitioned into sub-matrices of a $K * K$ square grid, such that each sub-matrix is small enough to fit into the local memory available to a compute node along with the necessary input and output vectors. Each sub-matrix is labeled by its coordinates on the grid, *i.e.*, $A_{u,v}$ corresponds to the sub-matrix on the uth row and vth column of the grid for $0 \leq u, v \leq K-1$. Each sub-matrix is stored in a separate file in binary Compressed Row Storage (CRS) format. The iterated SpMV process is seeded with an initial x vector which is partitioned accordingly to the row partitioning of the A matrix, that is to say into K parts. The initial vector x^0 is stored as sub-vectors x^0_u for $0 \leq u \leq K-1$ in a distributed fashion.

The SpMV is executed by first generating intermediate results: $x^i_{u,v} = A_{u,v} * x^{i-1}_u$. Then the output vector is generated by summing intermediate results: $x^i_u = \sum_v x^i_{u,v}$. Depending on the degree of parallelism available on a node (*i.e.*, the number of threads), local schedulers can decide to execute the *multiply* and *sum* operations assigned to them in parallel by partitioning the output vector. The list of operations for the first two iterations with a 3×3 partitioning of the A matrix is presented in Figure 3: 9 sub-matrix sub-vector multiplications and 6 sub-vector additions are necessary at each iteration. Dependencies between these operations are shown in Figure 4.

Using the DOoC middleware has several benefits. An easy way to program the iterated SpMV procedure using MPI is to perform the computation one iteration at a time, and perform all iterations identically. Assume that computations are performed using 3 nodes and node i stores the sub-matrices $A_{1,i}$, $A_{2,i}$ and $A_{3,i}$ on its file system. If a node can keep only one sub-matrix at a time on its main memory, then a typical MPI programmer would reach the execution plan laid out in Figure 5(a). Such an execution performs 6 matrix load operations (3 per iteration), 6 matrix vector multiply operations (3 per iteration) and 2 vector sum operations (1 per iteration) on each node with little synchronization.

However, in a context where loading the matrix into the memory is an expensive operation, one wants to reduce the

- $x^1_{0,0} = A_{0,0} * x^0_0$
- $x^1_{0,1} = A_{0,1} * x^0_0$
- $x^1_{0,2} = A_{0,2} * x^0_0$
- $x^1_{1,0} = A_{1,0} * x^0_1$
- $x^1_{1,1} = A_{1,1} * x^0_1$
- $x^1_{1,2} = A_{1,2} * x^0_1$
- $x^1_{2,0} = A_{2,0} * x^0_2$
- $x^1_{2,1} = A_{2,1} * x^0_2$
- $x^1_{2,2} = A_{2,2} * x^0_2$
- $x^1_0 = x^1_{0,0} + x^1_{1,0} + x^1_{2,0}$
- $x^1_1 = x^1_{0,1} + x^1_{1,1} + x^1_{2,1}$
- $x^1_2 = x^1_{0,2} + x^1_{1,2} + x^1_{2,2}$

- $x^2_{0,0} = A_{0,0} * x^1_0$
- $x^2_{0,1} = A_{0,1} * x^1_0$
- $x^2_{0,2} = A_{0,2} * x^1_0$
- $x^2_{1,0} = A_{1,0} * x^1_1$
- $x^2_{1,1} = A_{1,1} * x^1_1$
- $x^2_{1,2} = A_{1,2} * x^1_1$
- $x^2_{2,0} = A_{2,0} * x^1_2$
- $x^2_{2,1} = A_{2,1} * x^1_2$
- $x^2_{2,2} = A_{2,2} * x^1_2$
- $x^2_0 = x^2_{0,0} + x^2_{1,0} + x^2_{2,0}$
- $x^2_1 = x^2_{0,1} + x^2_{1,1} + x^2_{2,1}$
- $x^2_2 = x^2_{0,2} + x^2_{1,2} + x^2_{2,2}$

Fig. 3. Commands emitted for the first two iterations of the sparse matrix vector multiply operation.

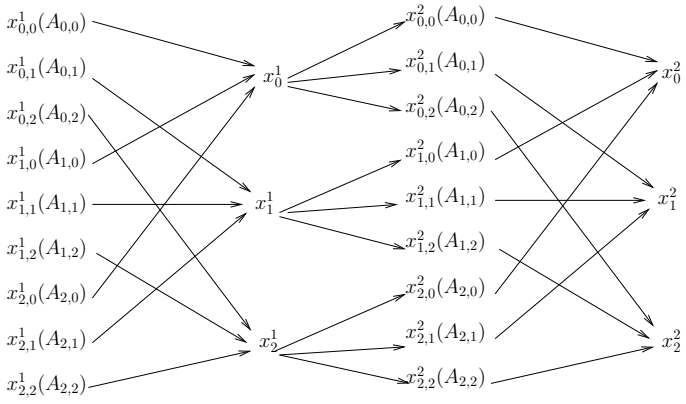


Fig. 4. Dependencies between the operations of Figure 3. Commands are abbreviated using the output vector name. The matrix blocks necessary for an operation are indicated between parentheses.

number of matrix load operations. This can be achieved by a simple reordering of the operations as depicted in Figure 5(b). The first iteration is performed similarly, but the second iteration is performed backwards. In this way, the intermediate result vectors $x^2_{u,2}$ are computed directly after the $x^1_{u,2}$'s. Since the sub-matrices $A_{u,2}$ are already in the memory, the number of matrix loads is reduced by 1 for the second iteration. Notice that the same scheme could be used for subsequent iterations. Swapping the order of traversal for each iteration leads to a cost of 3 matrix loads for the first iteration and 2 matrix loads for each subsequent iteration. This plan is automatically discovered and executed by the DOoC middleware without requiring any effort or input from the application programmer.

In an out-of-core execution, it is desirable to overlap the expensive loads from the file system with useful computations. If there is enough memory to store multiple sub-matrices at once, this overlapping can be achieved by loading the next sub-matrix, while still processing a sub-matrix. As mentioned in Section III, the local scheduler achieves this otherwise tedious task transparently, by querying the storage layer to learn about the amount of memory available, and issuing prefetching requests.

Finally, note that the reduction tasks required to obtain the result vector can be performed as soon as intermediate results become available. One does not have to wait for all SpMV tasks to be finished for reductions. DAG execution model adopted by DOoC makes the interleaving of SpMV and reduction tasks possible, potentially leading to better performance.

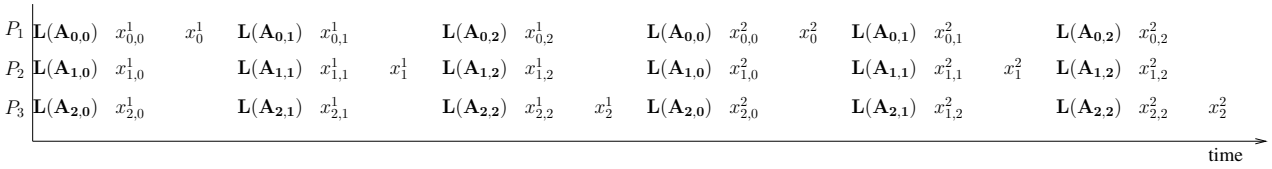
V. Experiments

Experiments are run on an experimental SSD testbed on the Carver cluster at NERSC. The testbed is composed of 50 nodes: 40 computational nodes and 10 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. All 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. DataCutter and the application are compiled with GCC 4.5.2. The InfiniBand interconnect is leveraged through the use of the MVAPICH 1.2 library.

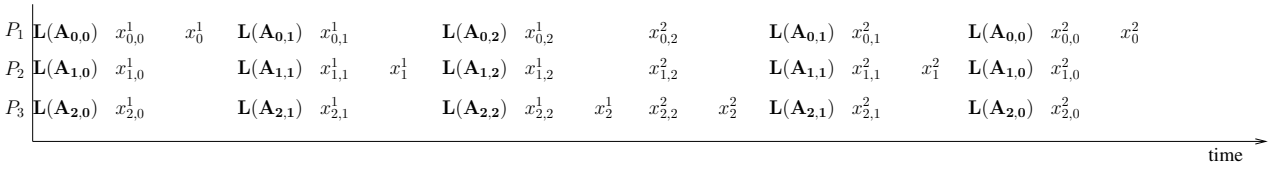
Each I/O node is equipped with two SSD cards, Virident tachIOn 400 GB, connected through the PCI-express bus. Each card can deliver up to 1 GB/s sustained read bandwidth, leading to a peak bandwidth of 2 GB/s per I/O node. I/O nodes are accessed by the compute nodes through a Global Parallel File System (GPFS) [16]. The maximum throughput the storage system can deliver is 20 GB/s. This is the performance we try to achieve in our experiments. Data is streamed from the I/O nodes to the requesting compute nodes using the 4X QDR InfiniBand interconnect.

Each local storage of our distributed data storage layer "owns" a part of the globally distributed data. Ideally, it would be beneficial for scheduling purposes, if each local storage had a dedicated file system to its own. Unfortunately, in our testbed all compute nodes share the single GPFS. To separate the data of each local storage, each compute node simply uses a different directory in GPFS. However, we should note that since the GPFS is shared, in some experiments this resulted in some noticeable variation in read bandwidth observed by individual compute nodes.

All runs are performed using a perfect square number of nodes so that the overall matrix is easily partitioned into a square grid. During the runs, each compute node is responsible for a block of the matrix with 50 million rows and columns which contains about 12.8 billion non-zero elements in total. Each block of the matrix is further decomposed into 25 sub-matrices, each of which actually constitutes the smallest unit of data transferred from the I/O nodes to compute nodes for processing. The size of a sub-matrix stored in binary CSR format is about 4 GBs. These submatrices have been generated randomly, such that the separation between two consecutive nonzero entries on a row is uniformly distributed in the interval $[1 : 2d]$, where d is a parameter. d is chosen to yield a certain number of total non-zero elements in a sub-matrix. Larger matrices are built by replicating the matrix block generated for a compute node for all nodes in an experiment.



(a) Regular



(b) Back and forth

Fig. 5. Two Gantt charts of a scenario with three nodes. The tasks are denoted by the name of the output vector. Since loading the matrices into the memory from disk is typically an expensive operation, they are denoted in bold.

For experiments reported in this section, we have chosen d such that the total number of non-zeros in a randomly generated matrix, and the number of non-zeros per row/column approximately match those of real MFDn matrices (see test_{1128} of Table II vs. the 9-node experiment of Table III, and also test_{4560} vs. 36-node experiment). The performance of an SpMV computation strongly depends on the structure of the sparse matrix and we are aware of the fact that our randomly generated matrices differ from real MFDn matrices in this respect. However, in an out-of-core computation, the main factor that determines the overall performance will be how fast sub-matrices can be transferred from the file system to the local memory of compute nodes. This transfer rate depends on the size of the sub-matrices rather than their structures. Therefore we believe that the performance results that we obtain from randomly generated matrices in this section will be representative of the behavior of DOoC on out-of-core MFDn computations.

A. Performance Results

In Table III, we present performance results collected from 4 SpMV iterations using a simple task scheduling policy. Here, all compute nodes perform their local SpMV's first. Then partial results are reduced on the first processor of each row, and the next iteration starts. According to Table III, overall performance increases almost linearly from 1 to 9 nodes, but plateaus after 16 nodes at around 3.2 Gflop/s. We extracted the bandwidth obtained by the filesystem I/O components ("read BW" column) from the logs of the application. We also present the fraction of the total runtime of the application which is not spent reading from the file system. Figure 6 (a) presents the runtime of the system relatively to the minimum achievable time assuming I/O is the only bottleneck and the peak bandwidth of 20GB/s is sustained.

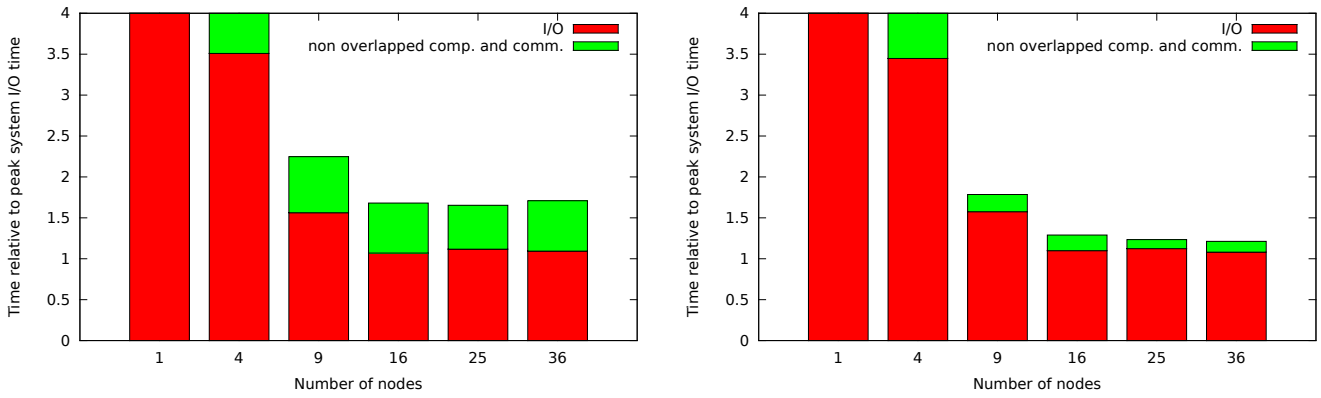
In the 16-node experiment, the I/O components see a bandwidth of 18.6 GB/s. This indicates that our DOoC based out-of-core iterated SpMV implementation attains 93% of the 20 GB/s peak I/O bandwidth of SSD-testbed. However, 36% of the time is not spent in filesystem I/O: the I/O operations represent only 64% of the total runtime. This indicates that I/O operations are handled efficiently in our implementation, but a significant portion of the overall execution is spent in inter-node synchronizations after SpMV.

The performance loss comes mainly from the two global synchronization steps in a single iteration: one after the SpMV, and another one after the reduction. Also, the network is not used efficiently: all the intermediate results $x^t_{i,j}$ are sent to the node that host $A_{i,0}$. In the next experiment, we remove the synchronization after the SpMV iteration, and keep only the synchronization between iterations (because a Lanczos iteration contains a reorthogonalization step which requires a global-synchronization). Also, note that each compute node is responsible from a block of $5*5$ arrangement of sub-matrices. At the end of a local SpMV, each node has 5 intermediate result vectors for a row of sub-matrices. In the simple task scheduling policy used above, all these intermediate vectors were being sent to the node responsible for the reduction. In this experiment, the reduction is instead first performed locally by each node before communicating the results.

Table IV presents the performance results from that experiment. Figure 6 (b) presents the time relative to optimal I/O time. Notice that all the runs on 9 nodes and more are now 17%-28% faster compared to the previous case. Performance degradation on one node is expected, since the local reduction causes one extra copy without bringing any improvements to the network communication. On 36 nodes, 90% of the time is spent in filesystem I/O and the actual I/O bandwidth is measured to be 18.5 GB/s. Only a small part of the actual computations and inter-node communications are not overlapped with filesystem I/O operations. The I/O operations are still performed at 92% of peak bandwidth. In all configuration, DOoC efficiently overlaps computation, communication and filesystem I/O: more than 85% of the time is spent in file system I/O.

B. Comparison to MFDn runs on Hopper

Finally, we compare the performance of our DOoC-based out-of-core iterated SpMV implementation against the in-core



(a) Simple schedule

(b) Interleaved schedule

Fig. 6. Runtime of DOoC on iterated SpMV relative to minimum time required to acquire the data assuming peak 20GB/s

TABLE III
RESULTS ON THE SSD TESTBED FOR SIMPLE SCHEDULING POLICY.

#nodes	matrix dimension	# non-zeros (billions)	matrix size (TBs)	total time (s)	GFlops/s	read bandwidth (GB/s)	non-overlapped time
1	50 M	12.8	0.10	290	0.35	1.5	13%
4	100 M	51.2	0.39	330	1.24	5.7	19%
9	150 M	115	0.88	384	2.40	12.8	30%
16	200 M	205	1.56	509	3.22	18.7	36%
25	250 M	320	2.43	791	3.23	17.9	32%
36	300 M	460	3.50	1172	3.15	18.3	36%

TABLE IV
RESULTS ON THE SSD TESTBED WITH INTRA-ITERATION INTERLEAVING AND AGGREGATION OF RESULT PER NODE.

#nodes	dimension	#nnz	size (TB)	time (s)	GFlops/s	read bandwidth (GB/s)	non-overlapped time	CPU-hour cost per iteration
1	50 M	12.8 T	0.10	293	0.35	1.4	0%	0.16
4	100 M	51.2 T	0.39	335	1.22	5.8	13%	0.74
9	150 M	115 T	0.88	336	2.74	12.7	11%	1.68
16	200 M	205 T	1.56	432	3.79	18.2	14%	3.84
25	250 M	320 T	2.43	644	3.97	17.8	8%	8.95
36	300 M	460 T	3.50	910	4.05	18.5	10%	18.20

MFDn runs on Hopper. The number of processing units used in the out-of-core runs are on the order of tens of cores, while typically thousands of cores are needed for in-core runs. Therefore a comparison in terms of time-to-completion would not be meaningful. Instead, we compare the two implementations in terms of the total CPU-hours burned during an iteration, which is computed by multiplying the number of cores used with the time spent for an iteration. As mentioned above, two cases allow for a fair comparison: 9-node out-of-core run vs. test_{1128}, and 36-node out-of-core run vs. test_{4560}. Notice that our out-of-core code does not implement the full Lanczos algorithm required for MFDn computations. But SpMV computations account for the major part of the computations in a Lanczos iteration. Therefore this comparison allows us to assess the feasibility of running MFDn on SSD-equipped clusters, instead of large-scale clusters like Hopper.

The CPU-hour cost of a single iteration on our experimental SSD-testbed is presented in Table IV. The CPU-hour costs of various MFDn computations were presented in Table II. Figure 7 presents a comparison of the data in these tables as a line graph.

The 9-node run on the SSD-testbed has a cost of 1.68 CPU-hours per iteration. This is comparable to the test_{1128} case on Hopper which costs 1.72 CPU-hours per iteration. However, the 36-node run costs 18.2 CPU-hours per iteration. This is about 2 times worse than the comparable Hopper run test_{4560}, which costs 9.70 CPU-hours per iteration. As mentioned above, after 16 nodes the I/O bandwidth on the SSD testbed plateaus. Since the performance of our out-of-core approach is clearly bounded by the bandwidth per node, this result is not surprising. Therefore we have re-run the same test-case (the matrix size of 3.50 TB) with 9 nodes where we achieve the best I/O bandwidth per node ratio. As we anticipated, the 9-node run took only slightly longer (1318 s vs.

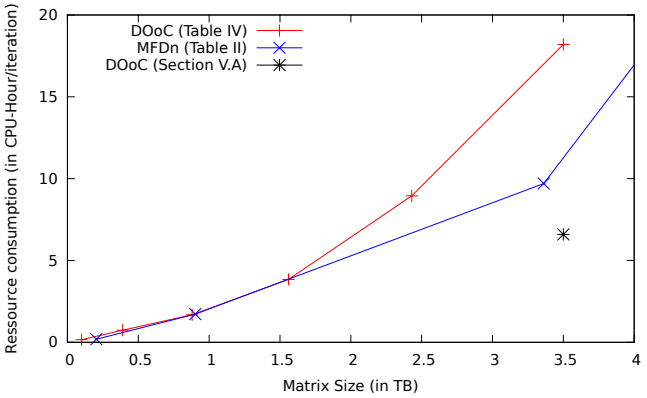


Fig. 7. CPU-hour costs of a single iteration on the SSD-testbed vs. MFDn runs on Hopper.

1172 s) and achieved a sustained I/O bandwidth of 12.5 GB/s. But the clear advantage of the 9-node run using the 3.50 TB matrix case is the cost per iteration, which is only 6.59 CPU-hours. This is significantly (32%) less than the cost of a Lanczos iteration on Hopper for test_{4560}, and it is marked as a star in Figure 7.

VI. Discussion and Future Work

The results obtained in Sect. V-B are very encouraging. Nevertheless, we believe that the I/O node, compute node separation in the experimental SSD-testbed is prone to some bottlenecks. First of all, the compute nodes need to go through the interconnection network to access data stored on the SSD cards. In a data-intensive application which deals with large amounts of data, the interconnection network may easily become the bottleneck. Heavy data transfers from the storage system might encumber the network for other traffic such as the communication between the computational nodes (*i.e.* the communication of the input/output vectors during SpMV). In this study, we were the sole users of the experimental testbed, but in practice there would be several applications running on a cluster at the same time. In such a case, accesses from multiple applications to the same I/O nodes might cause severe contentions. A final potential bottleneck is the GPFS which manages the interaction between the I/O nodes and compute nodes. GPFS does not allow manual tuning of parameters critical to I/O performance (such as striping width and count). Instead it dynamically alters these parameters to optimize the application performance. However, GPFS is not developed with today's emerging non-volatile storage technologies and the growing need for data-intensive applications, in mind. Moreover, when multiple applications with different data access characteristics access the same I/O node, GPFS's dynamic parameter adjustment algorithms may give sub-optimal results.

A. Different Software/Hardware Configurations

As future work, we are planning to try different filesystems for accessing the data stored on the I/O nodes and compare their performances against GPFS.

Also, we would like to investigate the effect of different hardware configurations. For achieving the best performance improvements in computationally heavy tasks, it is the *de facto* standard to connect the GPU accelerators on the compute nodes themselves, rather than hosting them on separate "accelerator nodes". By drawing an analogy from the GPU case, we argue that for achieving the best performance with data-intensive tasks on SSD-equipped platforms, SSD cards should be positioned on the compute nodes themselves, as well. We expect that such a design would solve most of the bottlenecks mentioned above. So in this design, a dataflow middleware such as DOoC presented in this work can seamlessly manage the interaction with the distributed non-volatile storage system, while increasing data locality for better performance.

B. Energy Efficiency

Power consumption of large-scale HPC platforms is another major concern. Leaked current during idle CPU times and the need to power up the entire DRAM constantly is a big contributor to this power consumption. On the other hand, SSDs are non-volatile, meaning that they do not need any power to store data. So, during data-intensive computations on SSD-equipped HPC platforms, power is needed only for the data that resides on the DRAM. Combined with the potential to reduce idle CPU times, SSD-equipped HPC platforms might increase the energy efficiency for certain classes of applications.

In this perspective, we are planning to investigate the SSD-equipped clusters from an energy-efficiency point of view, too. In the current configuration of the SSD-testbed we used, the separation between I/O nodes and compute nodes prevents shutting off unused I/O nodes to save energy. At all times, all I/O nodes must be powered up. Furthermore, all I/O accesses have to go through the InfiniBand network, which means huge amounts of data must be transferred over long distances. This is clearly a very costly operation in terms of energy usage. We think that a study where the energy-efficiency of alternative SSD-testbed configurations are compared against large-scale clusters like Hopper could be very interesting.

The only impediment to wide-range adoption of SSD-equipped HPC platforms could be the cost of high-end SSD cards (which currently costs above $10,000). The growing demand for high-performance non-volatile storage devices in various markets should quickly reduce these costs, as has been the case with most other computer technologies. Also, the increased efficiency of data-intensive tasks (measured in terms of CPU-hours and energy usage) that we expect from using SSD-equipped clusters can help pay for itself in the long-run.

SSD-accelerated supercomputers are being investigated to improve the efficiency of the graph traversal problem. In June 2011, according to the graph 500 benchmark[3], the Leviathan machine at LLNL is a single node machine equipped with a 12TB SSD system from Fusion-IO was used to process a graph of size 2^{36} with the same throughput than Krakenm a

[3]http://www.graph500.org/

machine with 6128 cores, using an in-memory algorithm on a graph of size 2^{34}.

VII. Conclusions

To deal with the increasing complexity of scientific applications, researchers have been using larger number of processors which lead to new challenges for efficient usage of the computing resources. The performance of certain applications such as most computational biology problems, applications analyzing huge flow of information from social media or financial markets, as well as some computational physics and chemistry applications dealing with sparse linear algebra problems is memory-bound. As we move towards the exascale era, where new architectures are anticipated to have less memory and less network bandwidth per core, we believe that it will be a great challenge for such applications to scale-up to the size of exascale machines. At this point, we think that the emergence of high-performance non-volatile storage devices presents a great opportunity. Out-of-core algorithms running on these new devices could decrease the CPU-hour and energy usage costs for such applications. However, developing an application that properly exploits the out-of-core execution capabilities is a difficult task. Therefore we introduced DOoC, a middleware that allows one to easily develop such applications by relying on task-based decomposition and globally accessible arrays. We demonstrated on a sparse linear algebra application that DOoC is able to exploit most of the performance of the system. It does not hinder the performance significantly. It should allow a more efficient use of computing resources by performing out-of-core operations on significantly less nodes that are necessary to run the same application in-core.

However, this work is only the first step toward our goal. We plan to demonstrate the gain in efficiency on a larger cluster equipped with fast local storage. Developing more linear algebra kernels will lower the bar for the application scientists to use our proposed paradigm.

Acknowledgment

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.

References

[1] 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.

[2] J. K. Reid and J. A. Scott, "An out-of-core sparse cholesky solver," *ACM Trans. Math. Softw.*, vol. 36, no. 2, 2009.

[3] 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.

[4] 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.

[5] J. A. Scott, "Scaling and pivoting in an out-of-core sparse direct solver," *ACM Trans. Math. Softw.*, vol. 37, no. 2, 2010.

[6] 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.

[7] A. Krishnamurthy, K. E. Schauser, C. Scheiman, R. Wang, D. Culler, and K. Yelick, "Evaluation of architectural support for global address-based communication in large-scale parallel machines," in *Proceedings of Architecture Support on Programming Languages and Operating Systems*, 1996.

[8] K. Datta, D. Bonachea, and K. Yelick, "Titanium performance and potential: An npb experimental study," in *proc. of LCPC 2005*, 2007, pp. 200–214.

[9] G. Drossel, "Solid-state drives meet military storage security requirements," *Military Embedded Systems*, Feb. 2007.

[10] 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.

[11] 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 the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08, 2008.

[12] 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.

[13] 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.

[14] 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.

[15] V. Rehn-Sonigo, D. Trystram, F. Wagner, H. Xu, and G. Zhang, "Offline scheduling of multi-threaded request streams on a caching server," in *International Parallel and Distributed Processing Symposium*, 2011, pp. 1167–1176.

[16] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 231–244.