Parallel Access of Out-Of-Core Dense Extendible Arrays

Ekow J. Otoo [#1] and Doron Rotem [#2]

[#] *Lawrence Berkeley National Laboratory*
1 Cyclotron Road, MS: 50B-3238
University of California
Berkeley, California 94720

[1] ekw@data.lbl.gov
[2] d_rotem@lbl.gov

***Abstract*— Datasets used in scientific and engineering applications are often modeled as dense multi-dimensional arrays. For very large datasets, the corresponding array models are typically stored out-of-core as array files. The array elements are mapped onto linear consecutive locations that correspond to the linear ordering of the multi-dimensional indices. Two conventional mappings used are the row-major order and the column-major order of multi-dimensional arrays. Such conventional mappings of dense array files highly limit the performance of applications and the extendibility of the dataset. Firstly, an array file that is organized in say row-major order causes applications that subsequently access the data in column-major order, to have abysmal performance. Secondly, any subsequent expansion of the array file is limited to only one dimension. Expansions of such out-of-core conventional arrays along arbitrary dimensions, require storage reorganization that can be very expensive. We present a solution for storing out-of-core dense extendible arrays that resolve the two limitations. The method uses a mapping function $\mathcal{F}_*()$, together with information maintained in *axial vectors*, to compute the linear address of an extendible array element when passed its k-dimensional index. We also give the inverse function, $\mathcal{F}_*^{-1}()$ for deriving the k-dimensional index when given the linear address. We show how the mapping function, in combination with MPI-IO and a parallel file system, allows for the growth of the extendible array without reorganization and no significant performance degradation of applications accessing elements in any desired order. We give methods for reading and writing sub-arrays into and out of parallel applications that run on a cluster of workstations, The *axial-vectors* are replicated and maintained in each node that accesses sub-array elements.**

I. Introduction

Multi-dimensional arrays constitute the fundamental data structures used in scientific computing. They include 1-dimensional structures, sometimes termed vectors; 2-dimensional arrays referred to also as matrices and arbitrary k-dimensional arrays of elements. An element is typically an elementary data type of integer, real or complex. Arrays of arbitrary size and dimensionality are used in a high performance scientific computing codes such as molecular dynamics, finite-element methods, climate modeling, scientific simulations, Astronomy, Astrophysics etc. The extensive use of algebraic libraries, e.g.., LaPACK, ScaLAPACK BLACS, ATLAS [1], Global Array (GA) Toolkit [2], attest to the array/matrix data model used in scientific computing. Persistent storage of these arrays are in the form of array files where the conceptual model of a multi-dimensional array is still maintained but the elements are mapped onto consecutive linear locations of the file. An array can be either sparse or dense thereby requiring different formats in the storage. We concern ourselves to only dense arrays in this paper.

In the last several years the various scientific domains have developed various file formats suitable for their respective applications. These include NetCDF [3], FITS [4] and HDF [5]. These data formats are self-describing and provide extensive specific language application programmers interface (API's), for accessing array elements from application programs. The data sets either consumed or generated by these scientific applications are from observational instruments, scientific instruments or large scale simulations. They are generally very large and can grow incrementally to to become of the order of terabytes. Processing of these datasets is done by applications that run as parallel or distributed programs on a cluster of workstations or massively parallel machines that involve hundreds to thousands of processors. More significantly, recent advances in hardware and storage capacity support the incremental growth of array datasets over time. The use of high performance computing, realized by low cost computing clusters, now provide the required parallel processing capabilities for managing these datasets. Not only should processing of array data be parallelized, but the array allocation and access of these array files (i.e., out-of-core arrays) should be extendible.

An array denoted as $A[\mathbb{N}_0][\mathbb{N}_1]\ldots[\mathbb{N}_{k-1}]$ is characterized by the dimensionality or rank k and the bounds of its dimensions, $\{\mathbb{N}_0, \mathbb{N}_1, \ldots, \mathbb{N}_{k-1}\}$. Each element is referenced by a k-dimensional index of the form $A\langle i_0, i_1 \ldots, i_{k-1}\rangle$, and is assigned to one of the $\mathbb{M} = \prod_{i=0}^{k-1} \mathbb{N}_i$, element locations. In the allocation of the array elements in a file, a *computed-access* mapping function $\mathcal{F} : (i_0, i_1 \ldots, i_{k-1}) \rightarrow j, 0 \leq j < \mathbb{M}$, maps each k-dimensional index to one of the $\mathbb{M}$ locations. We say an array realization is *weakly extendible* if any bound $\mathbb{N}_i$, can be incremented by appending newly allocated elements to the file without modifying the mapping function or reallocating already stored elements. It is *strongly extendible*, if the dimensionality or rank k, can be extended as well without modifying $\mathcal{F}()$. Our use of the term *extendible*

array assumes *weak extendibility* through out the rest of this paper. We use the notation $\mathcal{F}()$ when referring to conventional array mapping that allows extendibility in one dimension only and use the notation $\mathcal{F}_*()$ when referring to a mapping function that allows extendibility on all dimensions.

Array files, such as NetCDF [6] and HDF5 [5] have parallel counterparts called parallel netCDF and parallel HDF5 respectively. Other known file formats that can be processed via cluster computing and parallel processing are the Disk Resident Array [7] and Panda [8]. HDF5 file format allows for array file extendibility but this is limited only to the non-parallel version of the library. HDF5 achieves extendibility through array *chunking* with the *chunks* indexed by a B-Tree indexing method. Chunking is done by partitioning the index range $\mathbb{N}_i$ of each dimension i into I_i regular intervals of length c_r^i so that $\sum_{I_i-1} c_r^i < \mathbb{N}_i \leq \sum_{I_i} c_r^i$. A chunk is a k-dimensional sub-array of elements whose shape is characterized by $[c_r^0, c_r^1, \ldots, c_r^{k-1}]$ and its chunk size is given by $B_j = \prod_{i=0}^{k-1} c_j^i$ elements. A chunk is the unit of access of data between memory and file storage.

Except for HDF5, these array files allow for extendibility only in one dimension. The limitation is due to the fact that the mapping function used is generally one of the conventional array mapping functions often referred to as row-major ordering (i.e., C-language Order) or column-major ordering (i.e., FORTRAN language order). Such array mappings exhibit some restriction with respect to achievable performance. For example an allocation that uses row-major ordering performs poorly if an application subsequently desires the array in column-major order.

We present, in this paper, a new method for allocating arrays in a parallel file system and accessing them from a parallel MPI application programs that run on either a cluster of workstations or massively parallel systems such as the IBM SP2. We term this the *DRX-MP* library which stands for *Disk Resident Extendible Array* library for multi-processing. The suite of library functions allow for reading and partitioning of large disk resident array (called the principal array) into sub-arrays and then distributing these onto the processes of a parallel program. Any arbitrary dimension of the out-of-core array can be extended by appending new array elements to the file without reorganizing already allocated array elements. The memory resident sub-arrays of the processes are allocated in a conventional array order using either row-major or column-major order. Such an allocation is consistent with the processing model of the Global-Array [2], [9], [10] toolkit. We use the phrase *principal array* to refer to the totality of extendible array elements partitioned into sub-arrays and managed by the processes to avoid any confusion with the term *Global-Array* that refers the GA-Library. *DRX-MP* is intended to be an alternative library to the disk resident array (DRA) [11], [7] which is used for the out-of-core storage of Global-Arrays.

Like HDF5, *DRX-MP* has a serial processing counterpart library called simply as *DRX* and accesses an extendible array file that is stored in any POSIX-compliant Unix file system. *DRX* has the added feature that the memory arrays can be maintained as either conventional arrays or *memory resident extendible arrays* with I/O caching using the BerkeleyDB Mpool sub-system [12]. This paper focuses on *DRX-MP* and considers its details with respect to storing dense extendible arrays.

The elements of the arrays in *DRX-MP* are stored by chunks where each chunk is of some fixed block size. An array chunk $A[I_0, I_1, \ldots, I_{k-1}]$ has a k-dimensional index $\langle I_0, I_1, \ldots, I_{k-1} \rangle$ that is mapped onto linear chunk address locations q_* using a mapping function $\mathcal{F}_*()$. Given q_*, a corresponding inverse function $\mathcal{F}_*^{-1}()$ computes the k-dimensional index of a chunk. The array expands by adjoining hyper-slabs of array chunks that we call *segments of array chunks*. Figure 1 illustrates the case of a 2-dimensional array that has been expanded by adjoining chunks of array segments. Detailed explanation of the array growth is given in the next section. To simplify our explanations, we will always assume that a single process runs on each node of a cluster or a parallel computing system.

In conventional k-dimensional arrays, efficient computation of the mapping function is carried out with the aid of vector that holds k multiplying coefficients for the respective indices. We do the same by storing vectors of the multiplying coefficients of the adjoined array chunks each time the array expands. Each dimension has one vector. The stored vectors of multiplicative coefficients capture the history of the expansions and are organized as the meta-data information of the extendible array. By replicating the meta-data information over the nodes and storing the distribution information on each node, the address of any element of the principal array can be computed and each node can determine whether the element is local or remote. Efficient collective sub-arrays I/O is done from the respective processes of a parallel program by combining irregular distributed array access methods of MPI-2 [13] with the mapping function presented in this paper. Memory to memory exchange of array elements are carried out either with MPI-2 remote memory addressing (RMA) features or with the portable aggregate remote memory copy interface (ARMCI) library [14], [15].

DRX-MP and *DRX* are not file systems. Rather each is a library of functions for managing and accessing extendible multidimensional arrays stored in a file system. *DRX-MP* mimics the I/O interface of DRA so that it is consistent with Global Array shared memory computational model over both cluster and massively parallel computing systems.

The main contributions of this paper are that we present a method for storing an extendible array in a parallel file systems such that the array can be extended along any dimension without reorganizing the already allocated array elements. We show how the array can be read and distributed as sub-arrays of the respective processes of a parallel program. The sub-arrays of the respective processes can also be written int a single parallel extendible array file. Such I/O's can be done both independently and also as a collective I/O. Further, we show how one can access these extendible arrays and specify that the sub-arrays in memory to be in conventional array order;

i.e., either in row-major or in column-major order.

The rest of the paper is organized as follows. Section II gives an overview of the allocation scheme of the array elements by chunks. We describe also how the array is partitioned and distributed onto nodes of a cluster of workstations for processing. The scheme is illustrated with a 2-dimensional extendible array that is accessed using *BLOCK* by *BLOCK* array distribution scheme. In Section III, we def ne the detailed mapping function for computing the linear address of a chunk when given its k-dimensional index and also give the algorithm for computing the inverse function. The extendible array f le is described in Section IV where we describe the chunking technique and the associated meta-data information maintained. We also discuss how arrays are read, distributed and transposed to be in the desired ordering in memory and We give some examples of the programming interface functions that is used in combination with MPI application program. We conclude with section V where we also give directions for future work.

II. Overview Of Element Allocation and Access of DRX-MP

A. Basic Concepts

The basic concepts of the allocation scheme of a dense extendible array, both in-core and out out-of-core, is illustrated in Figure 1. Consider the 2-dimensional *principal array* of Figure 1, that is denoted by A[10][12] and stored in the f le F. The array is stored in chunks each of shape 2×3. We denote a chunk of an array A by $A_{[I_0,I_1]}$ where I_0 denotes the chunk index of the f rst dimension and I_1 denotes the chunk index of the second dimension. In the illustration of Figure 1, the emboldened labels denote the linear addresses of the chunks in the f le. The chunk $A_{[4,2]}$ is assigned to the linear address location 18 in the f le. Hence the mapping function computes $\mathcal{F}_*(4,2) = 18$. The elements within a chunk are assigned according to the conventional row-major ordering of an array. Once we access the chunk that an element belongs, computing the actual location of an element within the chunk is trivial.

The array expands along any dimension by allocating a segment of array chunks. Which dimension and when an array is expanded is determined by the application program. The array of Figure 1 grew from an initial allocation of chunk 0. It was then expanded by extending dimension 1 with chunk 1. This was followed with the extension of dimension 0 by allocating the segment consisting of chunks 2 and 3. The same dimension was then extended by appending chunks 4 and 5. Each expansion allocates chunks to retain the rectilinear shape of the array. Observe that the maximum index of a dimension does not necessarily fall exactly on a segment boundary. In our illustration, the maximum index value of dimension 1 is 9. The array bound of dimension 1 is $\mathbb{N}_1 = 10$.

Partitioning and distributing the array chunks onto processes is always along chunk boundaries. One instance of a distribution of the array onto 4 processes is illustrated by the f gure. We assume each a process is run on a separate node. The entire array f le is partitioned into disjoint rectilinear regions where each region is composed of a set of adjacent connected chunks referred to as a *zone*. Each process is then assigned a zone of the array where it becomes the primary owner. A *zone* is comprised of a set a chunks that form a rectilinear k-dimensional sub-array. When an array zone is allocated in memory, it is mapped onto locations using the conventional C-order or FORTRAN order.

Must I/O functions that read sub-array elements from disk into an array region in memory utilize nested loops that scan the index ranges that cover the sub-array in memory. The effect is that the linear ordering in memory direct accesses to disk that are random. Since the chunk layout on disk are sequential and are in increasing order of the linear addresses, independent I/O of sub-array regions are done as sequential scan of the chunks on disk. The inverse mapping function $\mathcal{F}_*^{-1}()$ is then used to compute the k-dimensional index of the elements read. Once the k-dimensional index is known the element can be assigned to the desired location in memory.

Processes control zones of array elements. For example the zone of array elements of process $P2$ is comprised of chunks $9, 10, 16$ and 17. Each processor has the meta-data information of the entire principal array and can compute the range of the chunk indices that def ne the zones of every other process. To access an element from any process, the process f rst determines which zone the element lies and consequently which process rank owns the zone. The element can then be accessed either as a local array element or as a remote array element. The remote memory access methods and the MPI-2 windowing features can now be applied for processing the array as if each process has access to the entire principal array. This model of programming is exactly the shared memory programming model of the Global-Array toolkit [9].

The functionalities of *DRX-MP* subsumes those of the Disk Residents Array (DRA) [11]. *DRX-MP*, has the added capabilities that:

- the principal array can grow indef nitely out-of-core.
- the required layout order of the sub-arrays in memory (either C-order or FORTRAN-order), can be specif ed when the f le is read, and do not require out-of-core array transpositions when used in different application programs that require different ordering of the array elements.
- accessing array elements is by a computed access method which is equivalent to a hashing scheme.
- An element can be accessed either directly from the f le or via a remote memory access of participating and cooperating processes.

B. Related Work

DRX-MP can be perceived as an an alternative to DRA [11], [7] with the added capability that the array is extendible. DRA is the persistent storage counterpart of the memory resident Global-Array and since over the past several years, Global-Array has developed a considerable library of processing functionalities and interfaces to a number of mathematical and scientif c computing libraries, we leverage the GA capabilities

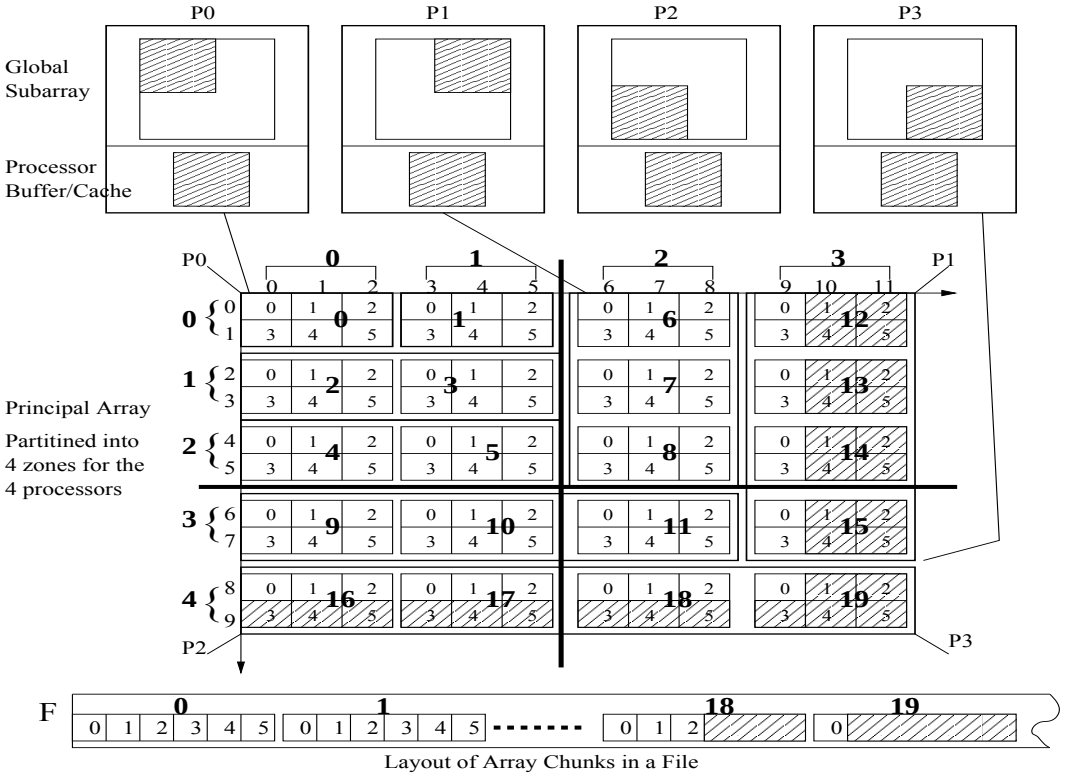


Fig. 1. An Allocation Scheme of 2-D Extendible Array File both In-Core and Out-Of Core

by providing the equivalent functionalities in *DRX-MP* as in the DRA library.

Other related work include a number of scientific file formats. Three of the common scientific file formats are NetCDF [3], HDF5 [5] and Panda [16], [8]. NetCDF is a standard library interface to data access functions for storing and retrieving array data. Its basic format is a file header followed by an organized data section. The file header contains the meta-data for dimensions, attributes, and variables. The data part consists of fixed size data that contain the data for variables that don't have an extendible dimension, followed by data record of variables that have an expandable dimension. Only one dimension is extendible. Parallel NetCDF (or *pNetCDF*) [6] is a parallel interface for NetCDF.

HDF5 is another file formatting scheme for multi-dimensional arrays. It stores multi-dimensional arrays by chunking and allows for array extendability by manging the chunks with a B-tree index. It is both a general purpose library and a file format for storing scientific data. A parallel version of HDF5 has parallel I/O though MPI-IO.

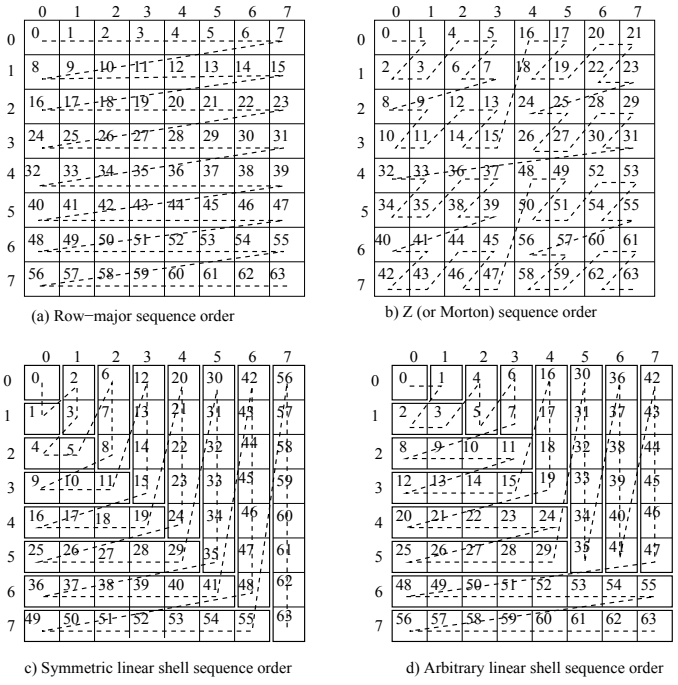
Panda is a library for input and output of multidimensional arrays for cluster and sequential platforms. Panda's array allocation is done using chunking. It strips the files in chunk sizes across I/O server nodes of a parallel file system. Panda supports HPF-style BLOCK and BLOCK CYCLIC(k) data distribution across multiple compute nodes on which Panda clients run. The clients cooperate with the server nodes to perform collective I/O. Unlike HDF5, Panda's chunk layout allows for extendability of the array in one dimension only.

This paper extends some of the earlier methods for realizing memory resident extendible arrays. Extensive detailed discussions of various techniques can be found in [17], [18], [19], [20], [21], [22].

III. Computing the Linear Chunk Addresses

A. Some Allocation Schemes for Arrays

The mapping function for addressing the regular sized array chunks on disk is very much similar to that for direct allocation of array elements on either disk or linear consecutive locations in memory. Consider the allocation of chunks as the cells of Figure 2. Some possible allocation schemes for an array are shown in Figures 2a - 2d. The labels in the cells indicate the linear addresses to which the chunks are assigned relative to the first chunk that is assigned to location 0. Figure 2a shows the conventional row-major order. Since this allows extendibility in one dimensional, this is not a mapping function of choice. Figure 2b illustrates another possible mapping with the standard Z-order (or Morton sequence order). It is one of a number of space-filling curves [23] whose mapping functions are well defined. An allocation scheme based on the Z-order mapping function is constrained to have exponential growth since the array can grow by doubling its size and only in a cyclic order of its dimensions. A linear expansion of an array is possible with the symmetric linear shell sequence order of Figure 2c. A mapping function is well defined but restricts expansions to be in a cyclic order otherwise chunk locations



(a) Row−major sequence order

b) Z (or Morton) sequence order

c) Symmetric linear shell sequence order

d) Arbitrary linear shell sequence order

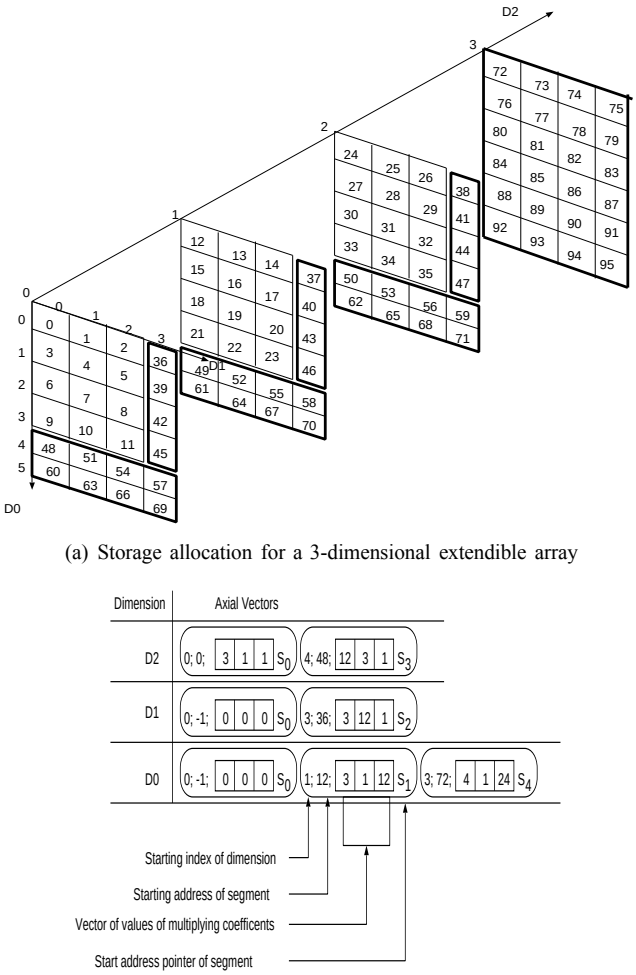
Fig. 2. Some Element Allocation Schemes for Arrays

may be assigned but unused. A much desired allocation scheme is that shown in Figure 2b. Any dimension can be extended in an arbitrary manner, The *axial-vector* technique uses k one dimensional vector of records to store information that allows us to compute the linear address of any chunk.

B. The Axial-Vector Approach

Suppose we now consider an allocation of the chunks of a 3-dimensional extendible array shown in Figure 3. Figure 3a depicts shows the cells as fixed size chunks. The computation done is for the linear address location of a chunk. Figure 3b shows the three Axial-Vectors of the respective dimensions and the records contained in each vector. We explain the fields of these records subsequently. Let $A[\mathbb{N}_0^*][\mathbb{N}_1^*][\mathbb{N}_2^*]$, denote the array in units of chunks, where $\mathbb{N}_j^*$ represents the fact that the bound has the propensity to grow. In this paper we address only the problem of allowing extendibility in the array bounds but not its rank. The labels shown in the cells represent the linear addresses of the respective chunks, as a displacement from the location of the first chunk.

Consider an initially array that is allocated as $A[4][3][1]$, where the dimensions D_0, D_1 and D_2 have the respective instantaneous bounds of $\mathbb{N}_0^* = 4, \mathbb{N}_1^* = 3$ and $\mathbb{N}_2^* = 1$. Suppose the array is then extended along dimension D_2 by two chunk indices; one immediately followed by another. The sequence of the two consecutive extensions along the same dimension, although does occur at two different instances, this is considered as an *uninterrupted extension* of the dimension. Repeated extensions of the same dimension, with no intervening extension of a different dimension, is referred to as an



(a) Storage allocation for a 3-dimensional extendible array

(b) The corresponding 3 distinct Axial-Vectors

Fig. 3. Illustration of a storage allocation of a 3-D extendible array

interrupted extension and is handled by only one expansion record entry in the axial-vector.

Since the labels in the cells denote the chunk's linear addresses the chunk $A_{[2,1,0]}$ is assigned to address 7 and chunk $A_{[3,1,2]}$ is assigned to address 34. Let the array be subsequently extended along the D_1 dimension by one index, then along the D_0 dimension by 2 indices and then along the D_2 dimension by 1.

Suppose that we now have a k-dimensional extendible array $A[\mathbb{N}_0^*][\mathbb{N}_1^*]\ldots[\mathbb{N}_{k-1}^*]$, for which dimension l is extended by λ_l, so that the index range increases from $\mathbb{N}_l^*$ to $\mathbb{N}_l^* + \lambda_l$. The strategy is to allocate a *segment* of chunks such that addresses within the *segment* are computed as displacements from the location of the first chunk of the *segment*. Let the first chunk of a segment of dimension l be denoted by $A_{[0,0,\ldots,\mathbb{N}_l^*,\ldots,0]}$. Address calculation is computed in row-major order as before, except that now dimension l is the least varying dimension in the allocation scheme while all other dimensions retain their relative order. Let us denote the location of $A_{[0,0,\ldots,\mathbb{N}_l^*,\ldots,0]}$ as $\ell_{\mathbb{M}_l^*}$ where $\mathbb{M}_l^* = \prod_{r=0}^{k-1}(\mathbb{N}_r^*)$. Then the desired mapping

function $\mathcal{F}_*()$ that computes the address q^* of a new chunk $A_{[I_0,I_1,\ldots I_{k-1}]}$ during the allocation is given by:

$$q^* = \mathcal{F}_*(I_0, I_1, \ldots I_{k-1}) = \mathbb{M}_l^* + (I_l - \mathbb{N}_l^*)C_l^* + \sum_{\substack{j=0 \\ j\neq l}}^{k-1} I_j C_j^*$$

$$\text{where} \quad C_l^* = \prod_{\substack{j=0 \\ j\neq l}}^{k-1} \mathbb{N}_j^* \quad \text{and} \quad C_j^* = \prod_{\substack{r=j+1 \\ r\neq l}}^{k-1} \mathbb{N}_r^* \tag{1}$$

We need to retain for dimension l the values of $\mathbb{M}_l^*$ - the location of the first element of the segment, $\mathbb{N}_l^*$ - the first index of the adjoined segment, and $C_r^*, 0 \leq r < k$ - the multiplying coefficients, in some data structure so that these can be easily retrieved to compute an chunk's address within the adjoined segment. These pieces of information is what is kept in the records of the axial-vectors. The *axial-vectors* denoted by $\Gamma_j[\mathcal{E}_j], 0 \leq j < k$, and shown in Figure 3b, are used to retain the required information. $\mathcal{E}_j$ is the number of stored records for axial-vector Γ_j. Note that the number of records in each axial-vector is always less than or equal to the number of chunk indices of the corresponding dimension. It is exactly the number of uninterrupted expansions along the dimension. In the example of Figure 3b, $\mathcal{E}_0 = 2, \mathcal{E}_1 = 2$, and $\mathcal{E}_2 = 3$.

The information of each expansion record of a dimension is a record comprised of four fields. For dimension l, the i^{th} entry denoted by $\Gamma_l\langle i\rangle$ consists of $\Gamma_l\langle i\rangle.\mathbb{N}_l^*$; $\Gamma_l\langle i\rangle.\mathbb{M}_l^*$; $\Gamma_l\langle i\rangle.C[k]$ - the stored multiplying coefficients for computing the displacement values within the segment; and $\Gamma_l\langle i\rangle.S_{i_l}$ - the displacement from the beginning of the file where the segment begins. Note however that for computing record addresses of array files, this last field is not required, since new records are always allocated by appending to the existing array file. i Given a k-dimensional chunk index $\langle I_0, I_1, \ldots, I_{k-1}\rangle$, the main idea in correctly computing the linear address is in determining which of the records $\Gamma_0\langle z_0\rangle, \Gamma_1\langle z_1\rangle \ldots \Gamma_{k-1}\langle z_{k-1}\rangle$, has the first maximum starting address of its segments. The index z_j is given by a modified binary search algorithm that always gives the highest index of the axial-vector where the expansion record has a maximum starting address of the segment less than or equal to I_j.

For example, suppose we desire the linear address of the chunk $A_{[4,2,2]}$, we first note that $z_0 = 1, z_1 = 0$, and $z_2 = 1$. We then determine that

$$\begin{aligned} \mathbb{M}_l^* &= \max(\Gamma_0\langle 1\rangle.\mathbb{M}_0^*, \Gamma_1\langle 0\rangle.\mathbb{M}_1^*, \Gamma_2\langle 1\rangle.\mathbb{M}_2^*) \\ &= \max(48, -1, 12); \end{aligned} \tag{2}$$

from which we deduce that $\mathbb{M}_l^* = 48, l = 0$, and $\mathbb{N}_l^* = \mathbb{N}_0^* = 4$. The computation $\mathcal{F}_*(\langle 4, 2, 2\rangle) = 48 + 12 \times (4 - 4) + 3 \times 2 + 1 \times 2 = 48 + 0 + 6 + 2 = 56$. The value 56 is the linear address relative to the starting address of 0.

The above calculations is equally applicable if the extendible array is realized in memory. In [22] we discuss the equivalent memory resident extendible array allocation function and formalize the characteristics of the extendible array realization functions. The essential algorithm to compute the chunk linear address is given below.

Function $\mathcal{F}_*$ ($\langle\Gamma_0, \Gamma_1, \ldots, \Gamma_{k-1}\rangle, \langle I_0, I_1, \ldots, I_{k-1}\rangle$)

input : k: number of dimensions
$\langle\Gamma_0, \Gamma_1 \ldots \Gamma_{k-1}\rangle$: a vector of k axial-vectors
$\langle I_0, I_1 \ldots I_{k-1}\rangle$: the k-dimensional index

output : q^* The linear address of the k-dimensional index

begin

Initialize:
$z \leftarrow 0$;
$iz \leftarrow bsearch(\Gamma_z, I_z)$;
$S^0 \leftarrow \Gamma_z\langle iz\rangle.S_{iz}^0$;
for $j \leftarrow 1$ ***to*** *k-1* **do**
  $ij \leftarrow bsearch(\Gamma_j, I_j)$;
  if $S^0 < \Gamma_j\langle ij\rangle.S_{ij}^0$ **then**
    $z \leftarrow j$;
    $iz \leftarrow ij$;
    $S^0 \leftarrow \Gamma_j\langle ij\rangle.S_{ij}^0$;
prodsum $\leftarrow 0$;
for $j \leftarrow 0$ ***to*** $k-1$ **do**
  if $j = z$ **then**
    prodsum $\leftarrow$ prodsum + $(I_j - \Gamma_z\langle iz\rangle.x) * \Gamma_z\langle iz\rangle.C_j$;
  else
    prodsum $\leftarrow$ prodsum + $I_j * \Gamma_z\langle iz\rangle.C_j$;
return *prodsum* + $\Gamma_z\langle iz\rangle.x$;

end

C. The Inverse Mapping Function $\mathcal{F}_*^{-1}$

The basic idea of deriving the k-dimensional index from the linear location address of an array element is easily explained with a conventional array mapping function. In row-major order allocation, an element $A\langle i_0, i_1, \ldots i_{k-1}\rangle$ is assigned to location ℓ_q, where q is computed by the mapping function defined as

$$q = \mathcal{F}(\langle i_0, i_1, \ldots i_{k-1}\rangle) = i_0 * C_0 + i_1 * C_1 + \cdots + i_{k-1} * C_{k-1}$$

$$\text{where} \quad C_j = \prod_{r=j+1}^{k-1} \mathbb{N}_r, 0 \leq j \leq k-1. \tag{3}$$

with $A\langle 0, 0, \ldots 0\rangle$ assigned to location 0.

In most programming languages, since the bounds of the arrays are known at compilation time, the coefficients $C_0, C_1, \ldots, C_{k-1}$ are computed and stored during code generation. Consequently, given any k-dimensional index, the computation of the corresponding linear address using Equation 3, takes time $O(k)$.

Suppose we know the linear address q of an array element, the k-dimensional index $\langle i_0, i_1, \ldots i_{k-1}\rangle$ corresponding to q can be computed by repeated modulus arithmetic with the coeff cients $C_{k-2}, C_{k-3}, \ldots, C_1$ in turn, i.e., $\mathcal{F}^{-1}(q) \rightarrow \langle i_0, i_1, \ldots i_{k-1}\rangle$. The same idea is carried over in computing the inverse mapping function when given the linear chunk address q^*, relative to the address of the chunk address 0. First, we need to determine which record of the axial-vectors, holds the coeff cients that we must apply. This is given by the record whose starting chunk address of its segment is the *maximum lower bound* of q^*. Be performing k independent binary searches of the *axial-vectors*, we can locate this record and consequently the necessary stored coeff cients. The rest of the calculation is similar to computing the inverse of a conventional array mapping function. The complexity of computing this function is $O(k + \log \mathcal{E})$, where $\mathcal{E}$ is the total number of axial records.

IV. Managing Extending Array Files

The current implementation of the *DRX-MP* storage scheme is simply as a pair of f les in regular parallel f le system, such as PVFS2 [24], that is accessed with MPI-IO. If a user requests the creation of a f le named *xyz*, the corresponding pair of f les created in the specif ed directory are *xyz.xmd* and *xyz.xta*. The f le *xyz.xmd* holds the meta-data information while the *xyz.xta* holds native binary f le of the principal array elements. The array elements can be of three basic data types: integer, double and complex. These correspond to the basic data types that can be def ned and accessed via MPI-2 remote memory access operations of MPI_Get(), MPI_Put() and MPI_Accumulate(). From an application's perspective, the extendible array f le is referred to only as $\langle dir\_path\rangle/xyz$ where $\langle dir\_path\rangle$ is the relative or absolute directory path that is a pref x to the f le's name. The implementation of *DRX-MP* is targeted for an eventual interface with the Global-Array toolkit so that it can leverage all the array manipulation and scientif c computing capabilities of the GA-toolkit. The current testbed of *DRX-MP* is a cluster of workstations running PVFS2 and MPICH2. Application programs are MPI programs that use MPI-IO either exclusively or in combination with other libraries such as Global-Arrays [2], HDF5 [5] and parallel NetCDF [6]. The library provides functions for creation, opening, closing, accessing sub-arrays, etc., of the dataset maintained as an array f le.

A. The Meta-Data File

The meta-data f le of the extendible multidimensional storage scheme, maintains a persistent copy of the content of the *axial-vectors* used in the linear address calculation. Other relevant pieces of information that are kept include the number of dimensions of the array, the data type, values of the chunk shape, the instantaneous bounds of the array, the number of chunks in the principal array f le, etc. When a f le is opened, the content of the meta-data f le is replicated in all participating processes. When an application opens a f le, it obtains a handle of a meta-data structure with which subsequent operations on the datasets can be carried out. All subsequent operations on the extendible array f le specify this handle as a one of its parameters. Memory resident arrays are also associated with a meta-data structure pointer irrespective of whether it is an extendible array or a conventional array. It gives a handle for communicating data between the disk resident extendible array and the in memory resident array. The role of pointers to the meta-data structure is similar to the use of a *FILE* handle in C or the use of an MPI_File() object in MPI_IO. Various f elds of the *DRX-MP* meta-data object can be accessed and set via various meta-data functions. We discuss some of the functions for operating on the principal extendible array f le.

B. Parallel Access of Sub-Arrays

First the principle array of DRX-MP and its meta-data f le can be initialized either from a single serial process or from a parallel program. The array is partitioned into chunks and written onto disk with chunks laid out either in row-major order or in the symmetric linear shell order. Subsequent expansion of the arrays can also be done by a serial process that expands the array by extending any abrtitrary dimension. Parallel expansions of the array can be done but by collective writes of processes that controls *zones* of array chunks that can be extended.

Accessing the principal array as a collection of sub-arrays into the distributed memories of a clusters requires using a function call that requires parameters of group communicator, the *DRX-MP* handle, an in-memory ordering of the indices of array, the in-memory base address of the arrays, etc. The manner in which the array is partitioned can be by default load balancing algorithm of *DRX-MP* or controlled by the application's algorithmic requirements.

We illustrative how some of these functions are implemented with MPI and MPI-IO calls with an example of how in Figure 1, we get the 4 processes to read the chunks of arrays in their respective zones. The sub-array chunks of Figure 1 are collectively read into the respective buffers of 4 processors $P0, P1, P2$ and $P3$ with code listing shown below. We utilize the irregular array method for collective I/O [25]. Note that distibution of the chunks and mapping of chunks in memory can be computed dynamically at run time. In the example code, we assign these statically.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ChunkSize 6
#define ChunksPerPoc 5
#define NDims 2
#define BUFSIZE 256
int main( int argc, char *argv[] )
{
    int  globalSize[NDims],
         globalSizeByChunks[NDims],
         chunkShape[NDims];
    int  i, j, myRank, nprocs, noOfChunks,
         ierr, memBufSize, count, ndbls ;
    int  chunkDistrib[] = {6, 6, 4, 4};
```

```
int globalMap[][6] = {{0, 1, 2, 3, 4, 5},
    {6, 7, 8, 12, 13, 14},
    {9, 10, 16, 17, -1, -1},
    {11, 15, 18, 19, -1, -1} } ;
int inMemoryMap[][6] = {{0, 1, 2, 3, 4, 5},
    {0, 2, 4, 1, 3, 5},
    {0, 1, 2, 3, -1, -1},
    {0, 1, 2, 3, -1, -1}} ;
// negative entries are not used
int *map, *inmemmap, *blocklens, mapSize ;
MPI_Datatype chunk, filetype, memtype;
MPI_Comm     comm;
char *filename =
   "/mnt/pvfs2/chunkedArray4.dat" ;
MPI_File     fh;
MPI_Status   status;
MPI_Offset   disp ;
double *memBuf ;
MPI_Init( &argc, &argv );
/*This code for 2 x 2 process decomp. */
ierr = MPI_Comm_size(MPI_COMM_WORLD,
    &nprocs);
if (nprocs != 4) {
    printf( "Size must be 4 \n" );
    MPI_Abort( MPI_COMM_WORLD, ierr );
}
/*Create cart topology of the processes */
// ---- Ignore creating topology ----
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
ierr = MPI_File_open(MPI_COMM_WORLD,
   filename, MPI_MODE_RDONLY,
   MPI_INFO_NULL, &fh);
if (ierr) {
    printf( "open failure %s\n",filename);
    fflush(stdout);
    MPI_Abort( MPI_COMM_WORLD, ierr );
}
/* For each processor rank, we should
 * generate the chunk addresses. For this
 * illustration we assign them statically*/
noOfChunks = chunkDistrib[myRank];
mapSize = (noOfChunks+1) * sizeof(int);
map = (int *) malloc(mapSize);
inmemmap = (int *) malloc(mapSize);
blocklens = (int *) malloc(mapSize);
for (j = 0; j < noOfChunks; j++) {
    map[j] = globalMap[myRank][j] ;
    inmemmap[j] = inMemoryMap[myRank][j] ;
    blocklens[j] = 1 ;
    printf("Rank %d: map[%d] = %d, \
        inmemmap[%d] = %d\n", myRank, j,
        map[j], j, inmemmap[j]);
}
MPI_Type_contiguous(ChunkSize,
        MPI_DOUBLE, &chunk);
MPI_Type_commit(&chunk);
MPI_Type_indexed(noOfChunks, blocklens,
        map, chunk, &filetype);
MPI_Type_commit(&filetype);
MPI_Type_indexed(noOfChunks, blocklens,
        inmemmap, chunk, &memtype);
MPI_Type_commit(&memtype);

disp = 0 ;
/** This is how to set file view **/
MPI_File_set_view(fh, disp, chunk,
    filetype, "native", MPI_INFO_NULL);
    ndbls = noOfChunks * ChunkSize;
    memBufSize = (ndbls+1) * sizeof(double);
    memBuf = (double *) malloc(memBufSize);

    for (i = 0; i < ndbls; i++) {
        memBuf[i] = -1.0 ;
    }
    MPI_File_read_all(fh, memBuf, 1,
            memtype, &status);
    MPI_Get_count(&status, chunk, &count);
    printf ("Rank %d: Number read = %d\n",
            myRank, count);
    if (myRank == 3) {//Check chunks of rank 3
        for (j = 0; j < ndbls; j++) {
          printf ("Rank %d: %d->val = %f\n",
              myRank, j, memBuf[j]);
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_File_close(&fh);
    free (memBuf); free (map);
    free (inmemmap); free (blocklens) ;
    MPI_Finalize();
    return EXIT_SUCCESS ;
}
```

C. Disk Resident Extendible Array File Library

The library provides a file header *drxmp.h*, that is included in any application wishing to use functions of the library. A pointer to the memory resident header of the meta-data *DRXMDHdrPtr* is defined. Some functions may return error codes that are defined in the context of the extendible array file environment. The meanings of most of the parameters can be inferred from the names and data types used in the prototype definitions. All *DRX-MP* functions must be enclosed by *MPI_Init()* and *MPI_Finalize()* routines. Some examples of the extensive list of functions are given below.

Initialization:

*int DRXMP_Init(DRXMDHdl *drxhdl, int kdim, size_t *initsize, int *chkshape, DRXType dtype, DRX-Comm comm)*;

This is a collective call that gives each process access to their respective meta-data handle. All other parameters are inputs. *kdim* states the rank or number of dimensions of the array, *chkshape* is an array of the chunk shapes, *dtype* specifies the data type of the array elements,

Opening:

*int DRXMP_Open(DRXMDHdl *drxhdl, char *file-name, char *mode)*

This function opens an extendible array file. The file must exists otherwise it returns and error. Failure to open the file returns an error. A successful opening reads the content of the meta-data into the contents of the structure given by the file handle *drxhdl*. Access permission mode is specified in *mode*.

Closing:

int DRXMP_Close(DRXMDHdl drxhdl)

This function closes the disk resident extendible array file whose handle is given by *drxhdl*.

Terminating:

int DRXMP_Terminate()

The function closes all opened extendible arrays and frees the *DRX-MP* allocated structures.

Reading:

*int DRXMP_Read(DRXMDHdl drxhdl, DRXMD-MemHdl memhdl, DRXMPStatus *stat)*

*int DRXMP_Read_all(DRXMDHdl drxhdl, DRXMD-MemHdl memhdl, DRXMPStatus *stat)*

These functions read the content of the extendible array given by the handle *drxhdl*, into the memory resident array whose base address can be extracted from the memory resident array handle *memhdl*. A collective reading version is given by the function *DRXMP_Read_all()*.

The above set only gives some examples of the functionality of *DRX-MP* library. Most of the functions are implemented using MPI_IO functions.

V. Conclusion and Future Work

We have presented some preliminary work on managing out-of-core dense extendible arrays in a parallel file system. Any array name specified is considered as a pair of files; one containing the principal array with suffix ".xta" and the other containing the meta-data information and has suffix ".xmd." It is possible to combine the meta-data file and the principal array file as a single file in which the meta-data information is kept as the header content of the DRXMP file but this is left for future work.

We have shown how the extendible array can be accessed and distributed using collective I/O as sub-arrays over a cluster of a workstations. The suite of functions for storing and reading elements of the array file is referred to as the *DRX-MP* library.

The array elements are stored out-of-core by regular chunks of some specified shape. We have presented the essential mapping function for accessing each array chunk and consequently the array elements. Some interesting features of our method are that:

- Instead of managing the chunks by an index scheme, the chunks can be addressed by a computed access function in a manner similar to hashing.
- There is no need for out-of-core array element transposition since this can be done on the fly as the array elements are read into core.
- The model of partitioning the array for distribution into memory is consistent with the computational model of the global-array toolkit. This allows the library to leverage the memory resident functions of global-arrays in manipulating the array once the array is read into memory.

.

Future work intends to develop the interface functions to work with Global-Array library. Further we intend to explore how the array distribution method can be generalized to ensure relative balanced data distribution and how to distribute the array by BLOCK Cyclic(K) methods. More importantly, we intend to pursue extensive performance testing and comparison with other file formats used in storing array files; namely parallel HDF5, parallel NetCDF and Disk Resident Arrays.

Optimizing the access by reconciling the chunk size with the strip size of the parallel file system for optimal chunk accesses.

Acknowledgment

This work is supported by the Director, Office of Laboratory Policy and Infrastructure Management of the U. S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing (NERSC), which is supported by the Office of Science of the U.S. Department of Energy.

References

[1] N. Repository, "http://www.netlib.org/."

[2] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169 – 189, 1996.

[3] NetCDF (Network Common Data Form) Home Page, "http://my.unidata.ucar.edu/content/software/netcdf/index.html."

[4] D. C. Wells, E. W. Greisen, and R. H. Harten, "FITS: a flexible image transport system," *Astronomy and Astrophysics. Supp. Ser*, vol. 44, pp. 363–370, Jun 1981.

[5] Hierachical Data Format (HDF) group, *HDF5 User's Guide*, National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana-Champaign, Illinois, Urbana-Champaign, Nov. 2004.

[6] J. Li, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, and A. Siegel, "Parallel NetCDF: A high-performance scientific I/O interface," in *Super Computing SC2003*, Pheonix, Arizona, USA, Nov. 15 - 21 2003.

[7] S. Krishnamoorthy, G. Baumgartner, C.-C. Lam, J. Nieplocha, and P. Sadayappan, "Layout transformation support for the disk resident arrays framework," *J. Supercomput.*, vol. 36, no. 2, pp. 153–170, 2006.

[8] P. Brezany, P. Czerwinski, A. Swietanowski, and M. Winslett, "Parallel access to persistent multidimensional arrays from HPF applications using *PANDA*." in *HPCN Europe*, 2000, pp. 323–332.

[9] 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," *Int'l. Journal of High Perf. Comput. Appl.*, vol. 20, no. 2, pp. 203 – 231, 2006, ga_acts.pdf.

[10] PNNL:, "Global Arrays Webpage. http://www.emsl.pnl.gov/docs/global/."

[11] J. Nieplocha and I. Foster, "Disk resident arrays: An array-oriented I/O library for out-of-core computations," in *Proc. IEEE Conf. Frontiers of Massively Parallel Computing Frontiers'96*, 1996, pp. 196 – 204.

[12] Oracle Berkeley DB, "http://www.oracle.com/technology/documentation-/berkeley-db/db/index.html."

[13] R. Thakur, W. Gropp, and E. Lusk, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, Mass: MIT Press, 1999.

[14] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Proc of Workshop on Runtime Syst. for Parallel Prog. (RTSPP'99), IPPS/SPDP, LNCS 1586*, 1999, pp. 533–546.

[15] J. Nieplocha and J. Ju, "ARMCI: A portable aggregate remote memory copy interface," Oct. 2000.

[16] K. E. Seamons and M. Winslett, "Multidimensional array i/o in panda 1.0." *Journal of Supercomputing*, vol. 10, no. 2, pp. 191 – 211, 1996.

[17] A. L. Rosenberg, "Managing storage for extendible arrays." *SIAM J. Comput.*, vol. 4, no. 3, pp. 287–306, Sept. 1975.

[18] T. A. Standish, *Data Structure Techniques*. Reading, Mass.: Addison-Wesley, 1980.

[19] D. Rotem and J. L. Zhao, "Extendible arrays for statistical databases and OLAP applications," in *8th Int'l. Conf. on Sc. and Stat. Database Management (SSDBM '96)*, Stockholm, Sweden, 1996, pp. 108–117.

[20] T. Tsuji, H. Kawahara, T. Hochin, and K. Higuchi, “Sharing extendible arrays in a distributed environment,” in *IICS ’01: Proc. of the Int’l. Workshop on Innovative Internet Comput. Syst.* London, UK: Springer-Verlag, 2001, pp. 41–52.
[21] E. J. Otoo and T. H. Merrett, “A storage scheme for extendible arrays.” *Computing*, vol. 31, pp. 1–9, 1983.
[22] E. J. Otoo and D. Rotem, “A storage scheme for multi-dimensional databases using extendible array f les,” in *Proc. 3rd Workshop on Spatio Temporal Database Management (STDBM’06), in conjunction with VLDB’2006*, Seoul, Korea, Sept. 11 2006.
[23] H. Sagan, *Space-Filling Curves*. New York: Springer-Verlag, 1994.
[24] N. Miller, R. Latham, R. Ross, and P. Carns, “Improving cluster performance with pvfs2,” *Cluster World*, vol. 2, no. 4, Apr. 2004.
[25] A. Ching, C. A., K. Coloma, W.-K. Liao, R. Ross, and W. Gropp, “Noncontiguous I/O accesses through MPI-IO,” in *Proc. 3rd IEEE/ACM Int’l. Symp. on Cluster Comput. and the Grid*. Tokyo, Japan: IEEE Computer Society Press, May 2003, pp. 104–111.