Parallel and Distributed Access of Dense Multidimensional Extendible Array Files (Extended Abstract)

Ekow J. Otoo
Lawrence Berkeley National Laboratory,
1 Cyclotron Road,
University of California
Berkeley, CA 94720

Abstract

Datasets in large scale scientific data management, are often modeled as k-dimensional arrays. Elements of these arrays, when kept in files on secondary storage (or disk), are allocated in a sequence of consecutive locations according to either a row-major or column-major ordering of an array mapping function. The allocation of an array in a file can be perceived as an out-of-core image of the in-core linear storage allocation. Such storage allocations limit the degree of extendibility of the arrays, both in core and out-of-core, to one dimension only. We address the problem of allocating storage for the elements of dense multidimensional extendible arrays so that the index range of any dimension can be arbitrarily extended without reorganizing previously allocated elements. The array can be partitioned, distributed and maintained as sub-arrays in the memories of the respective computational nodes of a parallel or distributed program. We present a mapping function $\mathcal{F}_*()$ and its inverse $\mathcal{F}_*^{-1}()$, for computing the linear address of an array element when given its k-dimensional index. Such a function is realized with the help of a *Leaf-Linked Red-Black Tree (LLRB-Tree)* data structure that stores, in a compact manner, the history of the expansions of the dimensions. Access to the LLRB-Tree forms part of the address computation and is maintained as part of the metadata of the corresponding array file. Parallel and distributed accesses of the multidimensional array elements are achieved by maintaining consistent copies of the *LLRB-Tree* on each processor. The technique introduced applies to extendible arrays in main memory just as much as for extendible array files that are resident on disk.

Key Words and Phrases: Multidimensional extendible array, Computed array index, Out-of-core array, parallel I/O.

1 Introduction

In large scale scientific applications, datasets are modeled generally as matrices or multidimensional arrays. A matrix is simply a 2-dimensional rectangular array of elements (or entries) laid out in rows and columns. High performance computations on these datasets run as parallel or distribute programs on a cluster of workstations or massively parallel machines that involve hundreds to thousands of processors. Each node, during the computation, accesses and manipulates sub-arrays (or array chunks) of one or more very large global arrays. The global array can be perceived as an array file from which sub-arrays are read from and written into from the individual nodes of the parallel program. The global array consists of an ordered collection of sub-arrays, each of which is maintained independently by a compute node of a parallel program, but together form a large multidimensional array spanning the entire processors. Such multidimensional array models of scientific datasets are typical in large scale computing performed by various Scientific and Engineering Simulations, Climate Modeling, High Energy and Nuclear Physics, Astrophysics, Computational Fluid Dynamics, Scientific Visualization, etc.

Special file formats for storing multidimensional array files that support sub-array accesses in high performance computing have been extensively studied and developed based on this model of a dataset. These include parallel NetCDF [4], HDF5 [3] and disk resident array (DRA) [7]. DRA is the persistent counterpart of the distributed main memory array structure called Global Array [8]. Except for HDF5, these array files allow for extendibility only in one dimension. We say an array, or an array file, is extendible if the index range of any dimension can be extended by appending to the storage of previously allocated elements, new elements that are addressed from the newly added index range. The address calculation is done without relocating elements of the previously allocated elements and without modifying the addressing function.

Suppose $A[\mathbb{N}_0][\mathbb{N}_1]\ldots[\mathbb{N}_{k-1}]$, is a k-dimensional array where $\mathbb{N}_j, 0 \leq j < k-1$, denote the bounds on the index ranges of the respective dimensions. An element denoted by $A\langle i_0, i_1 \ldots, i_{k-1}\rangle$, is referenced by a k-dimensional index $\langle i_0, i_1 \ldots, i_{k-1}\rangle$. Let $\mathcal{L} = \{\ell_0, \ell_1 \ldots, \ell_{\mathbb{M}-1}\}$, be a sequence of consecutive storage locations where $\mathbb{M} = \prod_{j=0}^{k-1} \mathbb{N}_j$. An allocation (or mapping) function $\mathcal{F}()$, maps the k-dimensional indices one-to-one, onto the sequence of consecutive indices $\{0, 1, \ldots, \mathbb{M}-1\}$, i.e., $\mathcal{F} : \mathbb{N}_0 \times \mathbb{N}_1 \times \cdots \times \mathbb{N}_{k-1} \rightarrow \{0, 1, \ldots, \mathbb{M}-1\}$. Given an address $\ell_j \in \mathcal{L}$, the inverse mapping function $\mathcal{F}^{-1}$, computes the k-dimensional index $\langle i_0, i_1 \ldots, i_{k-1}\rangle$ that corresponds to ℓ_j. Inverse mapping functions are often needed to compute the addresses of the neighbors of an element given its linear storage location.

All modern programming languages provide native support for multidimensional arrays. The storage mapping function $\mathcal{F}()$ is normally defined so that elements are allocated either in row-major or column major order. We refer to arrays whose elements are allocated in this manner as conventional arrays. Conventional arrays limit their growth, at run-time, to only one dimension. The consequence is that, since the dimensions cannot be extended at run time, programs are forced to declare at compile time, an array size that is sufficiently large for running most applications. Unfortunately, most of these applications utilize memory inefficiently since the maximum memory is always allocated but most of it is rarely used. The problem is easily avoided if an array of some small size is initially allocated and then incrementally expanded at run-time to accommodate just the right size of the array without abandoning the efficient address calculation. Such mapping functions apply also to the storage of array files.

There are numerous reasons why some applications in the scientific data processing need the notion extendible multidimensional array files. For example, in satellite imagery and remote sensing, the datasets are captured along defined swathes, defined spectral bands, time, etc. Some data processing application subsequently require incremental tiling of adjacent scenes and progressive inclusion of selected bands. The HDF5 and/or HDF-EOS data formats used in storing most of these remote sensing and satellite images are essentially multidimensional array files where the dimensions of the datasets are defined by the horizontal and elevations spatial coordinates, time, bands, etc. Progressive inclusion of adjacent tiles, and selected bands amounts to extending an initial allocated array along the respective spatial dimensions and spectral bands.

Consider the maintenance of very large array files in scientific applications. We desire extendibility of such files by allowing any index range to grow so that new array elements can simply be appended to the files without reorganizing the entire file. We present, in this paper, a mapping function $\mathcal{F}_*()$ and its inverse $\mathcal{F}_*^{-1}()$ for addressing elements of an array that is allowed to grow. Sub-arrays can be distributed over the nodes of either a cluster of workstations or a massively parallel machine. 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 for the adjoined block of array elements, each time the array expands. The computation of the linear address, corresponding to a k-dimensional index, uses these stored vectors. The entire array can be perceived as being formed from blocks of array elements termed *hyperslabs* or *segments* of the array. Figure 1a illustrates how the *hyperslabs* are arranged to form the entire array.

The stored vectors of multiplicative coefficients capture the history of the expansions and are organized as the *sentinel* nodes of a *Leaf-Linked Red-Black Tree (LLRB-Tree)*. There is one *LLRB-Tree* for each dimension. The *LLRB-Trees* allow us to compute the mapping function and its inverse function in times comparable to those of conventional arrays. By replicating these *LLRB-Trees* over the nodes that access the sub-arrays, and maintaining copies of the *LLRB-Trees* consistent, we can address elements of the array file fron any node. Efficient collective sub-arrays I/O is done from the respective nodes of a parallel program by combining irregular distributed array access methods of MPI-2 [16] with the mapping functions 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 [6, 9]. The technique presented in this paper is most beneficial to array library developers, e.g., the Global Array (GA) library [8], the disk resident array (DRA) [7], compiler developers, and application programmers wishing to maintains distributed arrays.

We give some details of the mapping function for dense multidimensional extendible arrays in the next section. Section 3, describes how a global array is distributed as sub-arrays and accessed from the respective nodes of a parallel program. We give a summary in section 4 where we also discuss some direction for future work.

2 The Mapping Function for Extendible Arrays

The allocation and access functions for extendible array files are essential the same for extendible arrays in memory. We address first, the approach for realization extendible arrays in memory and adopt these functions for the extendible array files.

2.1 Mapping Function for Conventional Arrays

Consider the k-dimensional array $A[\mathbb{N}_0][\mathbb{N}_1]\dots[\mathbb{N}_{k-1}]$, where each $\mathbb{N}_j, 0 \le j < k-1$, denotes the bound on the index range of dimension j. Suppose the elements of this array are allocated in the linear consecutive storage locations $\mathcal{L} = \{\ell_0, \ell_1 \dots, \ell_{\mathbb{M}-1}\}$, in row-major order according to the mapping function $\mathcal{F}()$. For the rest of the paper, we will always assume row-major ordering and for simplicity we will also assume that an array element occupies one unit of storage. An element $A\langle i_0, i_1, \dots i_{k-1}\rangle$ is assigned to location ℓ_q, where q is computed by the mapping function given as

$$q = \mathcal{F}(\langle i_0, i_1, \dots i_{k-1}\rangle) = i_0 \prod_{j=1}^{k-1} \mathbb{N}_j + i_1 \prod_{j=2}^{k-1} \mathbb{N}_j + \cdots + i_{k-1} \tag{1}$$

with $A\langle 0, 0, \dots 0\rangle$ assigned to ℓ_0, i.e., $\mathcal{F}(\langle i_0, 0, \dots, 0\rangle) = 0$ Alternatively, we can express the computation of q as

$$\begin{gathered} q = \mathcal{F}(\langle i_0, i_1, \dots i_{k-1}\rangle) = i_0 * C_0 + i_1 * C_1 + \cdots + i_{k-1} * C_{k-1} \\ \text{where} \quad C_j = \prod_{r=j+1}^{k-1} \mathbb{N}_r, \quad 0 \le j < k-1 \text{ and } \quad C_{k-1} = 1. \end{gathered} \tag{2}$$

with the understanding that an empty product evaluates to 1.

In a programming language such as C, since the bounds of the arrays are known at compile time, the coefficients $C_0, C_1, \dots, C_{k-1}$ are computed and stored during the code generation phase. Given any k-dimensional index, the computation of the corresponding linear address using equation 2, takes $O(k)$ elementary operations of additions and multiplications and is therefore $O(k)$. When storing elements of any array file, the same mapping function is used. The array mapping function for allocating and accessing elements can be viewed as a *hashing function.* The limitation imposed by $\mathcal{F}$ is that the array can only be extended along one dimension; namely dimension 0 since the evaluation of the function does not involve the bound $\mathbb{N}_0$.

We can still exploit the constraint imposed by $\mathcal{F}$ by shuffling the order of the indices whenever the array is extended along any dimension. We allocate the elements such that, within the newly adjoined hyperslab of elements, the bound of the dimension being extended is not required, i.e., the dimension being extended is made to vary the least in the row-major ordering.

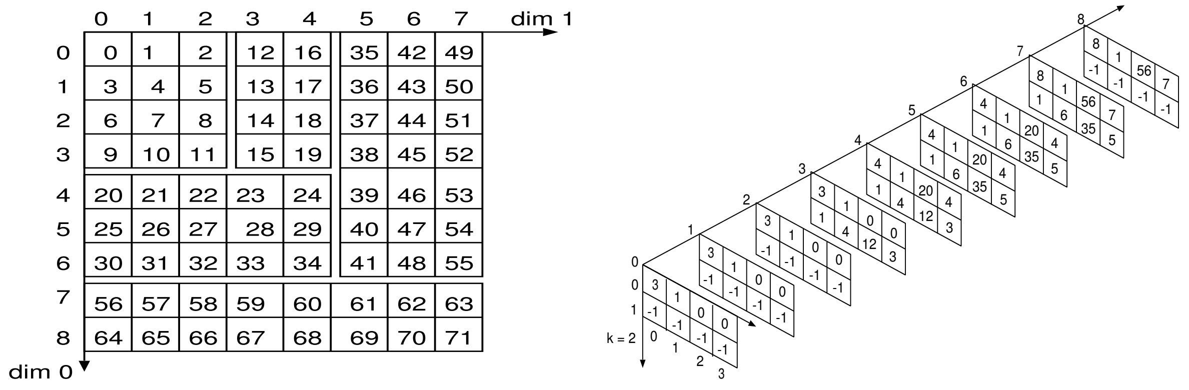
2.2 The Inverse Mapping Function for Conventional Arrays

Given the linear address of an element, we occasional need to determine whether elements, within some specified number of coordinate units along some dimensions exist. Such queries are referred to as neighbour queries with nearest neighbour finding being a special case. If the k-dimensional index of an element is known, finding a specified neighbour is simple and straightforward. We simply generate the k-dimensional index of the array element being sought by adding to or subtracting from the index values of the given element. The linear address is then computed using the mapping function. If we know only the linear address of the given array element, we will need to compute the k-dimensional index address from an inverse mapping function.

The inverse mapping function is simply evaluated by repeated modulus arithmetic using the coefficients $C_{k-2}, C_{k-3}, \ldots, C_0$ in turn. Suppose we are given the linear address q, and we wish to compute the index address $\langle i_0, i_1, \ldots, i_{k-1}\rangle$. The i_{k-1} index is derived as the remainder when q is divided by C_{k-2}, i.e., $i_{k-1} \leftarrow (q \bmod C_{k-2})$ and q is modified as the quotient of q divided by C_{k-2}, i.e., $q \leftarrow q/C_{k-2}$. Each of the coefficients is then reduced by the factor of C_{k-2}. i.e., $C_j \leftarrow C_j/C_{k-2}, 0 < j < k-2$. The process is then repeated to derive i_{k-2} using the modified value of C_{k-3}, and so on.

2.3 Mapping Function for Extendible Arrays

The question of organizing extendible arrays in memory, such that the linear location address can be computed as in the case of static arrays described above, is a long standing one [13, 14, 12]. Some solutions exist for extendible arrays of specific shapes [11]. A solution was proposed that uses an auxiliary array [10] to keep track of the information needed to compute the mapping function. In [15], the content of the auxiliary array was organized as a B-Tree. The use of auxiliary arrays can be expensive in storage depending on the pattern of extensions.



(a) 2-D Extendible Array (b) The Auxiliary Array

Figure 1: Storage allocation of an extendible array by the auxiliary array method

Figure 1a shows a 2-dimensional extendible array that was initially allocated as a 4×3 array, A[4][3]. We denote this in general as $A[\mathbb{N}_0^*][\mathbb{N}_1^*]$, where $\mathbb{N}_0^* = 4$ and $\mathbb{N}_1^* = 3$ are the instantaneous upper bounds. The array was later expanded along dimension 1 by extending the index range from 3 to 5. This means that after the expansion, 2 columns were adjoined to the original array. The labels shown in the array cells represent the linear addresses of the respective cells. For example the element $A\langle 2, 2\rangle$ is assigned to location 8 and $A\langle 2, 4\rangle$ is assigned to location 18. The array subsequently was extended along dimension 0 by increasing the index range from 4 to 7 and then along dimension 1 by increasing its index range from 5 to 8 and so on.

In general an array grows simply by adjoining array *segments.* An array *segment* or *hyperslab* can be viewed as a specialized array *chunk*(a term used in [3]), where all but one of the dimensions of a *chunk* take the maximum current values of the array sizes of their respective dimensions. Suppose that in a k-dimensional extendible array $A[\mathbb{N}^*_0][\mathbb{N}^*_1]\dots[\mathbb{N}^*_{k-1}]$, 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 the adjoined *segment* so that addresses are computed as displacements from the location of the element $A\langle 0, 0, \cdots, \mathbb{N}^*_l, \dots, 0\rangle$, using the row-major order as before, except that now dimension l is the least varying dimension in the allocation scheme. Denote the location of $A\langle 0, 0, \cdots, \mathbb{N}^*_l, \dots, 0\rangle$ as $\ell_{S^0_{\mathbb{N}^*_l}}$ where $S^0_{\mathbb{N}^*_l} = \prod_{r=0}^{k-1}(\mathbb{N}^*_r)$. Then the desired mapping function $\mathcal{F}_*()$, that computes the address q^* of an element $A\langle i_0, i_1, \dots i_{k-1}\rangle$ during the allocation is given by:

$$q^* = \mathcal{F}_*(\langle i_0, i_1, \dots i_{k-1}\rangle) = S^0_{\mathbb{N}^*_l} + (i_l - N^*_l)C^*_l + \sum_{\substack{j=0 \\ j\neq l}}^{k-1} i_j C^*_j$$
$$\text{where} \quad C^*_l = \prod_{\substack{r=0 \\ r\neq l}}^{k-1} \mathbb{N}^*_r \quad \text{and} \quad C^*_j = \prod_{\substack{r=j+1 \\ r\neq l}}^{k-1} \mathbb{N}^*_r \tag{3}$$

We need to maintain appropriately, the values for $S^0_{\mathbb{N}^*_l}$, l, N^*_l and the multiplying coefficients, $C^*_0, C^*_i, \dots, C^*_{k-1}$, for computing an element's address within the adjoined hyperslab. These pieces of information are organized and represented in an auxiliary array. We denote such an array by $\Gamma[\mathbb{M}^*][k][k+2]$, where $\mathbb{M}^* = \max(\mathbb{N}^*_0, \mathbb{N}^*_1, \dots, \mathbb{N}^*_{k-1})$. The auxiliary array Γ, expands naturally only in one dimension; namely the dimension corresponding to the increasing index values. Figure 1b shows the auxiliary array for the extendible array of Figure 1a. We use the value -1 to represent a *null* value. A slightly different presentation of this techniques was first introduced in [10]. Given a k-dimensional index $\langle i_0, i_1, \dots i_{k-1}\rangle$, the key idea to determining the correct address is in locating the segment that has the maximum starting address for all the segment that the indices $i_0, i_1, \dots i_{k-1}$ occur. This is done simply by comparing the values of $\Gamma\langle i_0, 0, k\rangle$, $\Gamma\langle i_1, 1, k\rangle, \dots$, $\Gamma\langle i_{k-1}, k-1, k\rangle$.

The main concern of the auxiliary array approach in implementing extendible arrays is that the storage overhead could be large. The main properties of the realization of extendible arrays using the auxiliary array approach are summarized in the following proposition.

Proposition 2.1. *A k-dimensional extendible array of N elements with arbitrary extendibility in any dimension is realizable by the auxiliary array method with an element access function that is computable in time $O(k)$ using space that is at best $O(k^2N^{1/k})$ and at worst $O(k^2N)$.*

Proof. The computation of the mapping function, once the array segment where the element occurs is known, is the same as in a conventional array. The additional overhead to determine which array segment has the largest starting block address requires k comparisons with a constant time access for each element of the auxiliary array. Hence the computation of the access function is still $O(k)$. The additional storage overhead is minimal when the array expansions cause it to maintain a cubical shape where all the dimensions are of equal size, i.e., $\mathbb{N}^*_0 = \mathbb{N}^*_1 = \dots = \mathbb{N}^*_{k-1} = N^{1/k}$. This gives the best additional storage of $O(k^2N^{1/k})$. The worst case occurs when $\max(\mathbb{N}^*_0, \mathbb{N}^*_1, \dots \mathbb{N}^*_{k-1}) \approx N$ and the result follows. □

2.4 Leaf-Linked Red-Black-Trees of Array Expansions

One main problem associated with the auxiliary array method for extendible arrays is that the worst case storage overhead can be large. A new approach to capturing the same information stored by the auxiliary array is with a *Leaf-Linked Red-Black Tree (LLRB-Tree).* There is one LLRB-Tree for each dimension and the corresponding root nodes are stored in a vector of length k.

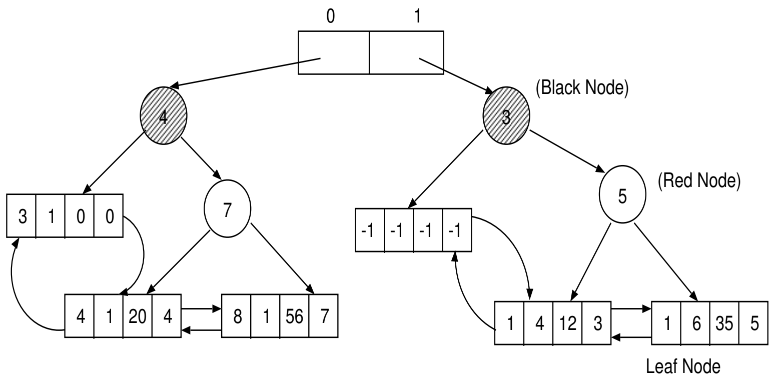
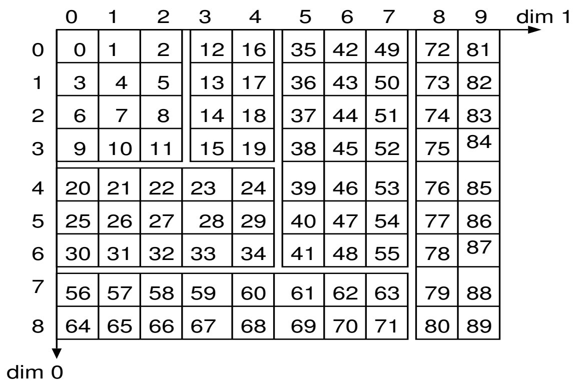


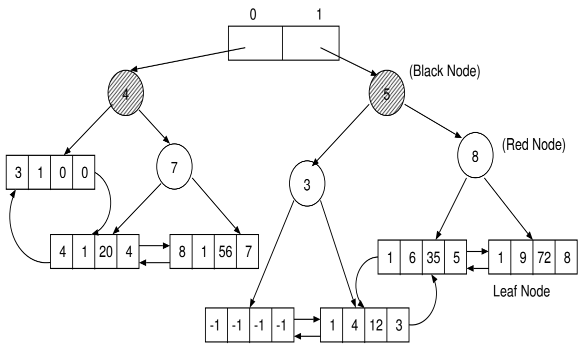
Figure 2: The Leaf-Linked Red-Black Tree Structure of the Auxiliary Array

Figure 2 shows the (LLRB-Tree) for the same extendible array of 1a. Unlike a regular Red-Black tree described in some detail in [2], the *LLRB-Tree* stores complete records only at the leaf nodes. The internal nodes only store separator keys for directing the traversal during a search. For this reason all searches terminate at a leaf node. The leaf nodes (or sentinel nodes, as is sometimes called), are double linked, hence the name *Leaf-Linked Red-Black Tree*. The significance of linking the leaf nodes is to allow for the inverse computation of the mapping function. Suppose we arrive at an internal node during the search of the record of a given key. A left branch is followed if the search key is less than the key at the node; otherwise a right branch is followed. All the usual Red-Black tree properties are preserved through standard left- and right-rotations.



dim 0 \ dim 1	0	1	2	3	4	5	6	7	8	9
0	0	1	2	12	16	35	42	49	72	81
1	3	4	5	13	17	36	43	50	73	82
2	6	7	8	14	18	37	44	51	74	83
3	9	10	11	15	19	38	45	52	75	84
4	20	21	22	23	24	39	46	53	76	85
5	25	26	27	28	29	40	47	54	77	86
6	30	31	32	33	34	41	48	55	78	87
7	56	57	58	59	60	61	62	63	79	88
8	64	65	66	67	68	69	70	71	80	89

(a) 2-D extendible array extended along dimension 1 by two columns



(b) The LLRB-Tree of the expanded array

Figure 3: Storage allocation of an extendible array using LLRB-Tree

Suppose the extendible array of Figure 1a is extended along dimension 1 by two additional columns given by indices 8 and 9. The allocation of storage is as shown by the labeling of the cells of the adjoined segment of array elements. We need to update the LLRB-Tree of dimension 1. The vector of values constructed is of the form $\langle 1, 9, 72, 8\rangle$ with a search key value 8. The search for the terminal node corresponding to this record terminates on the right link of node 5. Since the record does not exist, we proceed to insert it. To insert this record we replace the terminal node with an internal node colored red having a key value of 8. The left link points to the old record of the form $\langle 1, 6, 35, 5\rangle$. The right link of the internal node points to the new record being inserted. After establishing the links of the terminal nodes to each other the black height of the tree is fixed-up since the newly inserted node is colored red and has a red colored parent. The result of fixing up the black height of the tree generates the new tree shown in Figure 3b. Due to space limitation we leave out formal description of the LLRB-Tree algorithm. We state the main characteristics of the extendible array realization using the *LLRB-Tree* in the following proposition without a formal proof.

Proposition 2.2. *A k-dimensional extendible array is realizable with the* LLRB-Tree *method with an element access computation in time $O(k + k \log(k + \mathcal{E}) - k \log k)$, using $O(k\mathcal{E})$ additional space, where $\mathcal{E} = \sum_{j=0}^{k-1} \mathcal{E}_j$ and $\mathcal{E}_j$ is the number of uninterrupted extensions or leaf records of the* LLRB-Tree *of dimension j.*

Observe that if no array extensions occur, the element access function is computed very much the same as in a conventional array.

3 Allocation of Large Multidimensional Array Files

An array file can be perceived as a persistent copy of the array in main memory. Scientific applications that access such arrays run as parallel programs on a cluster of workstations or on massively parallel machines. For our purposes, we will ignore the details of the organization of the array files. However, we should note that only the contents of leaf nodes of the *LLRB-Trees* need to be stored in order to reconstruct the *LLRB-Trees* in memory. We assume that the elements of the arrays are fixed size data types. The environment being considered stores the array file in a parallel file system such as the parallel virtual file system (PVFS2) [5]. Application programs are MPI programs that use MPI-IO either exclusively or in combination with other libraries such as Global-Arrays [8], HDF5 [3] and parallel NetCDF [4]. Figure 4 shows one typical configuration.

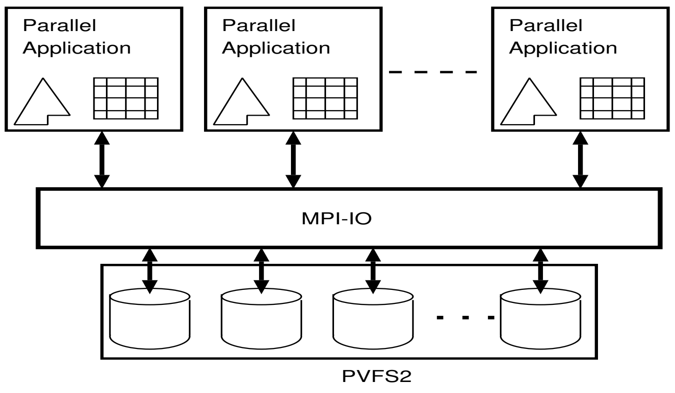


Figure 4: Typical Configuration for Accessing and Processing Very Large Multidimensional array file

Each node, of a parallel program, accesses and processes sub-arrays independently and may exchange copies of the memory resident arrays with other nodes. An extendible array mapping function is significant for such applications since very large array files can grow by appending new data items to the file without reorganizing the entire file. However, the increase in the index ranges of any dimension must be subsequently reflected in the LLRB-Tree of the file before the new elements can be accessed. The strategy for accessing sub-arrays, maintains distributed copies, of the same LLRB-Trees associated with the file, in each node.

The protocol for maintaining and manipulating the file is simple. Nodes can perform independent or collective reads and writes of their respective sub-arrays as long as the accesses do not overlap new elements of the files being appended. An external program can add new elements to the array file by requesting an exclusive write lock token on the vector of header nodes of the LLRB-trees. This does not prevent concurrent reading of the LLRB-Trees. When the writer for the new array elements completes, the vector of LLRB-Trees of the metadata is updated. If the writer is one of the parallel processes accessing the file, an appropriate record, corresponding to the new segment of added elements, is sent to every node to update their respective LLRB-Trees held in memory. Our implementation and tests of array files does synchronization of the vector of LLRB-Trees, using one-sided communication.

4 Summary and Future Work

We have shown how, a k-dimensional extendible array can be mapped onto linear sequential storage locations so that the linear addresses can be computed from the k-dimensional indices. Such a mapping function has been achieved by using information maintained in a compact structure, referred to as the LLRB-Tree. The LLRB-Tree is an interesting data structure in itself. Due to space limitation, we have left out formal discussions of its characteristic.

By allowing each process to maintain independent copies of the LLRB-Tree, high performance scientific computations on large array files can proceed while the array file continues to grow. The mapping function developed is primarily intended to be used for implementing libraries for array files such as the Global Array Library. The technique also gives a solution to a long standing problem of whether one can define a storage allocation function for extendible arrays with arbitrary index expansions. Previous solutions presented in [1, 12, 13, 14] have not been very satisfactory. Future work intends to explore how our solution can be incorporated in actual array libraries and how to extend the technique to arrays with varying dimensionalities or ranks.

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-AC03-76SF00098. 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] A. Barrero. Implementation of abstract data types with arrays of unbounded dimensions. *Commun. ACM*, 39(12es):167–174, 1996.

[2] C. Cormen, T. Stein, C. Leiserson, and R. Rivest. *Introduction to Algorithms, 2nd Ed.* MIT Press, Cambridge, Mass., 2001.

[3] Hierachical Data Format (HDF) group. *HDF5 User's Guide.* National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana-Champaign, Illinois, Urbana-Champaign, release 1.6.3. edition, Nov. 2004.

[4] 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.

[5] N. Miller, R. Latham, R. Ross, and P. Carns. Improving cluster performance with PVFS2. *Cluster World*, 2(4), Apr. 2004.

[6] 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*, pages 533–546, 1999.

[7] 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*, pages 196 – 204, 1996.

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

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

[10] E. J. Otoo and T. H. Merrett. A storage scheme for extendible arrays. *Computing*, 31:1–9, 1983.

[11] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 90 – 105, Atlanta, March 1983.

[12] D. M. Ritchie. Variable-size arrays in C. *Journal of C Language Translation*, 2(2):81–86, Sept. 1990.

[13] A. L. Rosenberg. Allocating storage for extendible arrays. *J. ACM*, 21(4):652–670, Oct 1974.

[14] A. L. Rosenberg. Managing storage for extendible arrays. *SIAM J. Comput.*, 4(3):287–306, Sept. 1975.

[15] 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)*, pages 108–117, Stockholm, Sweden, 1996.

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