Efficient Input and Output for Scientific Simulations

S. Kuo, M. Winslett, Y. Cho, J. Lee and Y. Chen
Computer Science Department
University of Illinois
{s-kuo,winslett}@cs.uiuc.edu

Abstract

Large simulations which run for hundreds of hours on parallel computers often periodically generate snapshots of states, which are later post-processed to visualize the simulated physical phenomenon. For many applications, fast I/O during post-processing, which is dependent on an efficient organization of data on disk, is as important as minimizing computation-time I/O. In this paper we propose optimizations to support efficient parallel I/O for scientific simulations and subsequent visualizations. We present an ordering mechanism to linearize data on disk, a performance model to help to choose a proper stripe unit size, and a scheduling algorithm to minimize communication contention. Our experiments on an IBM SP show that the combination of these strategies provides a 20-25% performance boost.

1 Introduction

Progress in developing high-performance processors and fast-speed networks has made large-scale scientific simulations possible, such as the simulations of nuclear devices that nations are developing in lieu of environmentally unfriendly empirical tests. Large simulations often run for tens to hundreds of hours on parallel computers and generate snapshots of states periodically. The output datasets are usually multi-dimensional and the volume is large, requiring special care in transferring them out of processors' memory in order to minimize the overall execution time. After data are off-loaded, they are analyzed and post-processed to visualize the evolving state of the simulated physical phenomenon. The latest trend is for the results of the analyses to be fed back immediately to the still-running simulation program to change simulation parameters (computational steering or interactive, integrated visualization). Therefore, fast I/O during post-processing, which is dependent on an efficient organization of data on disk, is as important as minimizing the computation-time I/O. Meeting both criteria poses a significant challenge in data management for time-dependent simulations, exceeding the expertise of domain scientists. As a result, most scientific simulations are I/O bottlenecked.

To overcome the I/O bottleneck problem, the Panda group at the University of Illinois (http://drl.cs.uiuc.edu/panda/) has developed a parallel I/O library to facilitate storage management for scientists. Our goal is to provide a simple high-level interface to users to efficiently access the underlying I/O subsystem across all platforms. Panda 2.1 includes very efficient algorithms for moving data to and from a given layout on disk, and our attention has recently turned to the issue of automatically choosing a good layout. In previous work, we proposed the "natural chunking" [23] and the "partitioned natural chunking" [15] data layouts for parallel checkpoint operations, which occur frequently in long-running simulations to provide fault tolerance. These layouts meet the special needs of checkpoint/restart operations and have been demonstrated to deliver high performance. In [5], we proposed the use of a simulated annealing approach to search for good data layouts given detailed knowledge of future I/O requests. This paper also addresses the issue of choosing optimal layouts on disk for common I/O operations and focuses on designing physical schemas for time-dependent simulation output data that will subsequently be visualized. The retrievals that occur during visualization are typically in the form of multidimensional range queries and therefore have different characteristics from checkpoint/restart operations, which read and write entire arrays. Also, the subspaces to be retrieved are typically determined at run-time and so Panda cannot obtain the advance knowledge it needs to search for good layouts. Therefore, a new approach is needed to choose layouts on disk for simulation output data.

Several database systems have been developed to manage multidimensional data for fast post-processing retrievals and visualizations [1, 3, 2, 9, 22]. These systems often deal with spatial data and are mostly designed for geographic systems (GIS) or remote sensing databases. Their storage support, such as R-trees, has been specifically designed to this end. Also, in these systems, datasets are often bulk loaded into the storage devices and the bulk load time is not a very important performance issue, compared to retrieval times. For scientific simulations, however, the underlying datatypes are arrays or grids, which are not as complicated as spatial data. Also, the I/O operation to output a snapshot of simulation results should be executed as efficiently as possible, as it is interleaved with computations (not nearly enough memory is available for the snapshot operation to be non-blocking). Therefore, existing database systems cannot be used directly to efficiently move data in and out of scientific simulation programs, and other specially

This research is funded by the U.S. Department of Energy through the University of California under Subcontract number B341494. We gratefully acknowledge use of the advanced computing resources at Argonne's Center for Computational Science and Technology.

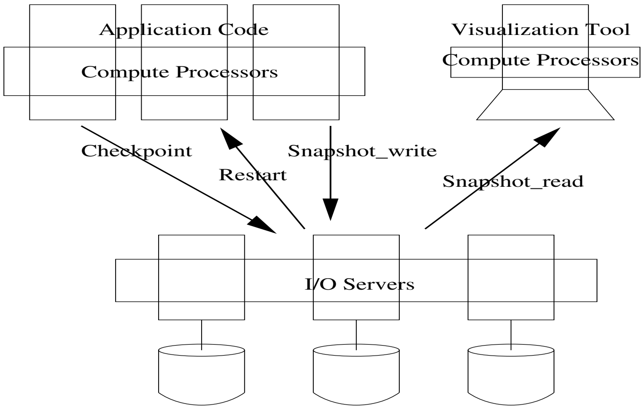

Figure 1: System architecture of Panda

designed I/O libraries, such as Panda, are needed for this purpose.

In the remainder of the paper, we begin by giving an overview of the Panda I/O library. Section 3 describes the proposed data organizations for efficient snapshot output/visualization operations. Results of the experiments conducted on an IBM SP2 are shown in Section 4. Related work is discussed in Section 5 and Section 6 concludes this paper.

2 Panda Basics

Panda is a DBMS-style I/O library developed to facilitate storage management for scientists, targeted for high-performance scientific applications running on parallel platforms having a single-program multiple-data (SPMD) structure. Currently, it supports I/O for two kinds of scientific data structures - arrays and grid hierarchies. An array is the most common data structure for high-performance scientific applications and a grid hierarchy is a special data arrangement which is usually used by adaptive mesh refinement (AMR)-style applications [20]. With AMR, the application starts with a coarse grid with a minimum acceptable resolution covering the entire problem space and dynamically places grids of higher resolutions onto interesting areas during the computation until all solution points are represented with an acceptable resolution. For either data structure, data are distributed across the processors using HPF-style or general block data distributions [8]. HPF-style BLOCK, CYCLIC and $*$ distribution directives are used to describe a regular decomposition of the problem space to processors; the generalized block distribution allows data to be irregularly distributed across processors, in order to balance the computational load among processors. Generally, a grid in a grid hierarchy is implemented as an array with the size and shape needed to be provide a particular resolution. Thus in Panda, a grid hierarchy is handled as a set of arrays with different resolutions. For ease of discussion in this paper, we describe only the array data structure, with grid hierarchies handled similarly.

Figure 1 shows an environment where Panda is running. As shown in the figure, scientific applications and visualization tools are running on Panda clients, which communicate with Panda servers when they want to input or output data.

Panda provides users with a set of high-level I/O interfaces, including `snapshot_read`, `snapshot_write`, `checkpoint` and `restart`. With these APIs, applications can read and write entire arrays as well as arbitrary rectangular regions of arrays directly. For each I/O operation, users describe the arrays to be read/written - ranks, sizes, memory locations, the distributions of the arrays among compute processors and (optionally) the desired arrangement of array data on I/O servers. If users do not specify the desired organization of data on disk, Panda automatically chooses a layout suitable for the specific I/O operation. So far, we have designed two natural chunking layouts [15, 23] for checkpoint operations and use an optimizer to search for optimal layouts for a given anticipated sequence of I/O requests [5]. In this paper, we are adding layout selection functionality in Panda for snapshot_write/snapshot_read operations.

In addition to providing high-level interfaces and optimal physical organizations, an important factor in Panda's good performance is a novel I/O strategy called *server-directed I/O* [23]. The server-directed I/O strategy uses knowledge of the current distribution of an array in memory and the desired distribution on disk in order to coordinate the arrival of data at I/O servers, so that the I/O servers perform only sequential access to data files during I/O. More precisely, for collective writes, the I/O servers gather data from compute processors in the order that the data will be written to disk, so that unnecessary seeks are avoided. Rearrangement of data between the desired in-memory and on-disk distributions is built into the scatter/gather processes and executed on-the-fly. By using the commodity file system available on the I/O servers to perform actual file I/O, Panda's logically sequential access to a data file translates into physically sequential access, the cheapest possible form of I/O and the best utilization of available disk bandwidth.

3 Data Organizations for Snapshot Outputs

Applications solving time-dependent problems output data at selected intervals over time (snapshot data), which will later be post-processed or visualized by visualization tools. In this paper, we assume that future access patterns to visualize the data are not known *a priori* during a snapshot output (array write) operation. Therefore, intuitively, an optimal layout for an array snapshot is to decompose the array into mesh-like I/O chunks (tiles), called a *chunking layout*, then distribute I/O chunks to I/O servers with the goal that every future retrieval can be balanced among the disks and use maximum parallelism. However, simply using a chunking layout does not guarantee high performance in inputting/outputting data if the following issues are not considered: declustering - how do we distribute the I/O chunks among the I/O servers; ordering - how should we linearize the I/O chunks on each disk; chunking - what is the optimal I/O chunk size; and scheduling - in what order should the I/O servers gather/scatter I/O chunks from/to compute processors? In the remainder of this section, we discuss the way we address these issues to achieve high performance I/O for both snapshot_write and snapshot_read operations. In describing the approaches we have designed to address these problems, we assume that each I/O server is connected to a different disk.

Given M disks, a GDM based declustering method assigns the D-dimensional chunk with index $[c_0, \ldots, c_{D-1}]$ in the mesh of chunks to disk $(c_0 \times p_0 + \cdots + c_{D-1} \times p_{D-1}) \ mod \ M$, where $(p_0, \ldots, p_{D-1})$ are called the GDM parameters. In [16], we proposed a method to pick GDM parameters for datasets of any dimension, called GDM_Cartesian. We demonstrated that GDM_Cartesian significantly outperforms DM [10], FX [14], Hilbert-curve [11], Vector-based [4], HalfM and Fibonacci [21] on average, for the set of all possible range queries. Thus, for the experiments described in this paper, we used GDM_Cartesian to decluster chunks. Due to space constraints, we refer the reader to [16] for a full description of the GDM_Cartesian approach.

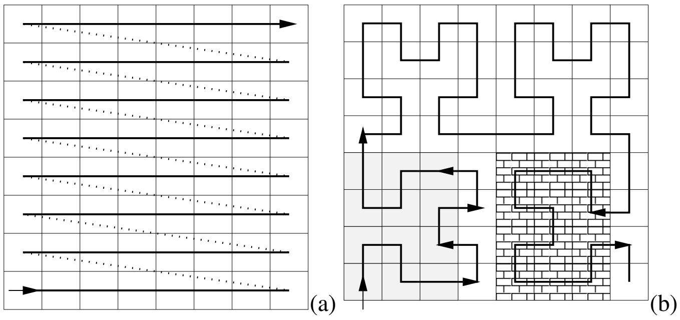

Figure 2: Two linearization methods for 2D arrays: (a) axis-traversal. (b) Hilbert-curve mapping. The two rectangles filled with patterns are for the discussion in Section 3.2.

3.1 Ordering strategy

In addition to a good declustering strategy which can distribute chunks to multiple disks so that every future retrieval in a snapshot_read operation will be balanced among the disks, it is also important to minimize the time to fetch multiple chunks from a single disk, which is affected by the linear order of the chunks on disk. When visualizing part of a multidimensional array, only some of the chunks residing on a disk have to be fetched. Therefore, it is desirable that these chunks which are to be fetched together be close to each other on disk to minimize disk seek overhead[1].

The most common way to linearize chunks is by traversing the chunks according to the axis order, e.g. starting from the z-axis, then the y-axis and finally the x-axis for a 3D mesh of array chunks. Figure 2-(a) shows the result of applying the axis-traversal strategy to 2D array chunks. In [22], Sarawagi et al proposed to change the order of axes in traversal, according to the access pattern of future retrievals. For example, if visualization users will view slices in the z-dimension frequently, it is desirable to traverse the 3D array chunks in the order $y \to x \to z$. As we assume that future access patterns are not known in advance in our work, the idea proposed in [22] is not applicable and we need an ordering approach which is good for all possible future retrievals.

In [13], Jagadish proposed the use of the Hilbert-curve to linearize a set of multidimensional chunks. He defined a mapping function $f : N^D \to N$ as $f : Hilbert\_curve([c_0, \ldots, c_{D-1}]) = a$, which maps a D-dimensional chunk with index $[c_0, \ldots, c_{D-1}]$ to a 1D value a. Chunks are then ordered according to their mapped value. Figure 2-(b) shows the result of applying the Hilbert-curve mapping strategy to linearize a set of 2D array chunks. It is shown in [13] that the Hilbert-curve ordering strategy outperforms other mapping methods, such as the Z-curve [19], for all possible range retrievals. Also, the cost of transforming a chunk index to a value using the Hilbert-curve mapping function is low - on the order of a few dozen machine cycles. For these two reasons, we chose to implement the Hilbert-curve ordering strategy as an alternative ordering strategy in Panda to linearize chunks on each disk.

3.2 Choosing a proper I/O chunk size

In this work, we assume that an I/O chunk is the unit of transfer between servers' memory and disks. That is, each server allocates a contiguous sequence of bytes, equal to the size of an I/O chunk, in memory as an intermediate buffer between disk I/O and communications with compute processors. In a write operation, each I/O server gathers data from compute processors to fill the buffer and empties the buffer using a single write request to the I/O server's underlying file system. In a read operation, the servers read data from disk at top speed to the buffer, then scatter data out to the compute processors that need it.

However, other units can be used in transferring data between servers' memory and disks. For example, in the original design of Panda, a "subchunk", with size equal to a pre-defined disk buffer size, instead of an entire I/O chunk, is the unit of transfer. If the size of an I/O chunk is larger than the disk buffer size, the I/O chunk is further decomposed into a set of subchunks such that the size of each subchunk is equal to the size of the disk buffer. In the Panda experiments described in this paper, we set the disk buffer size to be the same as the size of an I/O chunk to avoid subchunking, as illustrated in the following example (Figure 3). Assuming the size of the array is 16 MB and the disk buffer size is 1 MB, Figure 3-(a) shows the case where the array is decomposed using a 4×4 mesh such that the size of each I/O chunk is equal to the disk buffer size. Figure 3-(b) shows the case where the array is decomposed using a 2×2 mesh, i.e. the size of each I/O chunk is larger than the disk buffer size. During the I/O operation, each I/O chunk is further decomposed into a set of subchunks of size equal to the disk buffer size. Thus, the pieces of array data transferred each time (a subchunk in this case) can no longer be described by a simple mesh, as the subchunks within a chunk may have different sizes and may have boundaries that do not coincide with row and column boundaries. And the shape of a subchunk is no longer mesh-like which we expect would give better performance for range queries of arbitrary shapes. So we avoid subchunking in this work.

This would complicate future range retrieval, so we avoid subchunking in this work.

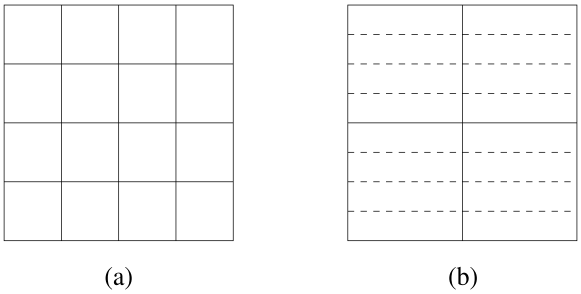

Figure 3: Illustration of different shapes of I/O chunks.

Given that an I/O chunk is the unit of transfer, the next question is: how big should chunks be, i.e., how much data should the I/O servers read/write before scattering/gathering it? For I/O operations that read/write an entire array, the requested data can be read/written in sequential order at each I/O server. Thus, there are no overheads caused by disk seeks or accessing unwanted data. Therefore, provided that appropriate write-behind and prefetching policies are supported by the underlying file system to speed up the write and read performance respectively, the only factors in choosing an appropriate I/O chunk size are the disk I/O and message passing performance - the I/O chunk size should be chosen such that the time for a disk activity to fill or empty out a buffer is no less than the time to scatter/gather data of the same size, in order to overlap the communication with the disk I/O activity. For example, in a read operation, while I/O servers are scattering data in the current buffer to compute processors, the file system should be prefetching the next sequence of data to the buffer.

[1] In fact, the actual number of disk seeks is dependent on the physical allocations of the disk pages, which might be different from what we would expect given the logical order in which I/O chunks are stored. Here, we assume that the logically sequential access to a data file translates to physically sequential access, which is the case in the commodity file systems we have used, when writing out very large files.

However, for I/O operations which only read/write part of the array, there are more factors to be considered in choosing a proper I/O chunk size. For these I/O operations, requested data can no longer be read/written in sequence and the disk arm has to be moved interleavedly with file I/O in order to read/write only those pieces of the array which are requested. Also, as an I/O chunk is the unit of disk transfer, which is read in/written out using a single read/write call, unwanted data will be accessed when only parts of the buffer data are requested. Intuitively, using a larger chunk size reduces the number of disk seeks, but increases the amount of unwanted data accessed. Using a smaller chunk size has the opposite effect. The appropriate chunk size is dependent on the base and shape of the subarray to be read/written in order to balance the number of disk seeks and the amount of unwanted data accessed.

Since multiple factors, with complex interactions between them, affect the determination of an appropriate chunk size, it is hard for us to pick the right size by hand or by simply solving an equation. A cost model can help with the analysis to determine a good I/O chunk size and we built a performance model for this purpose.

3.2.1 The performance model

In previous work, we developed a detailed performance model for Panda which takes the description of an I/O operation (to read/write a whole array or to read/write a subarray with a given base and shape) as inputs, and outputs the estimated time for Panda to execute that particular I/O operation [6]. To apply the existing model to choose an appropriate chunk size which is good for all possible future snapshot_read requests, the simplest way is to determine the set of all potential I/O chunk sizes, then for each chunk size, enumerate all combinations of bases and shapes as input parameters, average all output values and finally choose the I/O chunk size which has the lowest average predicted time. As one can expect, enumerating all combinations of bases and shapes would take a long time, e.g. to visualize a $256 \times 256 \times 256$ subarray out of a $512 \times 512 \times 512$ array, there are over 10 million possible bases to place the subarray and the total number of tests will increase exponentially when the shape of the subarray is changed as well. Therefore, it is not feasible to directly use the model and enumerate all combinations. However, we do not want to bias the selection of an appropriate chunk size by considering only a few sample subarray read requests. A revised model is needed. In the remainder of this section, we explain the relevant part of the preexisting Panda performance model and describe the new modules we plug in to control the cost of choosing an I/O chunk size.

The existing model decomposes Panda's execution into three phases - startup, intermediate and closing. The new model considers only the intermediate stage and does not count the startup (when compute processors inform I/O servers about the details of the I/O request) and shutdown cost of a Panda I/O request, as these costs are essentially constant for all I/O requests. For each I/O request, the model computes the response time for each I/O server at the intermediate stage, and reports the largest value among them as the indication of Panda's execution time. Equation 1 presents the time for I/O server i at the intermediate stage.

$$T(i) = \sum_{1 \le k \le NC(i)} T_{chunk}(i,k), \qquad (1)$$

where $NC(i)$ is the total number of I/O chunks to be transferred by I/O server i, and $T_{chunk}(i,k)$ is the time to process I/O chunk k at server i:

$$T_{chunk}(i,k) = \begin{cases} T_{comp}(i,k) + T_{file\_io}(i,k) + T_{comm}(i,k) \\ \quad - T_{overlap}(i,k), \text{for a prefetched chunk} \\ T_{comp}(i,k) + T_{file\_io}(i,k) + T_{comm}(i,k) \\ \quad + T_{seek}(i,k), \text{otherwise.} \end{cases} \qquad (2)$$

Here, $T_{comp}(i,k)$ is the time for server i to determine which compute processors hold which parts of chunk k; $T_{file\_io}(i,k)$ is the time for server i to read chunk k from disk; and $T_{overlap}$ is the savings in the total response time because of successful prefetching. That is, when chunk k resides contiguously in the file with chunk $k-1$, the underlying file system can prefetch part of chunk k while the server is sending chunk $k-1$ to the appropriate compute processors. Finally, $T_{comm}(i,k)$ is the time to send the pieces of chunk k's data to each relevant compute processor. Thus if a prefetch succeeds, as shown in the first case in Equation 2, the total response time can be computed by summing T_{comp}, T_{comm} and T_{file_io} together and subtracting the prefetching savings. However, if chunk $k-1$ and k lie apart in the file, as presented in the second case in Equation 2, there is no prefetching deduction in the total response time and an extra seek overhead, $T_{seek}(i,k)$, has to be added in order to move the disk arm to the right position to read chunk k[2].

We have built a micro benchmark suite to extract the platform-dependent parameters, such as the file system and message passing system performance. Based on the outputs of the microbenchmark, the two terms T_{file_io} and T_{comm} can be calculated easily (using equations not shown here) for each chunk. The term $T_{overlap}$, which accounts for the overlapping of message passing and file system prefetching, can be computed easily, too, based on the system's prefetching policy, i.e. how much data will be fetched by the file system when it detects a sequential read. The average seek overhead, T_{seek}, can be obtained from the disk specification. T_{comp} can be obtained by choosing some sample requests, feeding them to Panda, and measuring the average time Panda spends in that activity.

The only terms left to be resolved in order to estimate Panda's execution time for an I/O operation are the number of chunks belonging to the first case in Equation 2 and the number of chunks falling into the second case, i.e. how many chunks can be read sequentially and how many chunks will result in disk seeks. As explained earlier, we do not want to enumerate all possible input requests, feed them into the performance model, and average their results. Instead, we devise a new module which estimates Panda execution time for all combinations in a more efficient way by grouping all subarray read requests with the same shape together and estimating their average execution time at once. Also, recall that we are only interested in calculating the largest I/O time among all servers. Instead of computing the average response time of each server and reporting the largest number, we only consider the servers which might possibly have the largest response time, i.e. the server which is responsible for accessing the largest number of chunks and the server which will have the largest disk seek overhead. Variations in T_{comm} (e.g. by taking into consideration the effect of scheduling I/O, which reorders the transfer of data between compute processors and I/O servers to minimize communication contention, as explained in Section 3.3), would also need to be considered on platforms where disk bandwidth is high compared to interconnect bandwidth, e.g. clusters with a slow interconnect. As our target platform is the SP, we need not consider T_{comm} when determining the slowest servers.

[2] The model does not consider the effect of caching as inter-request locality is very low when reading and writing large pieces of very large arrays.

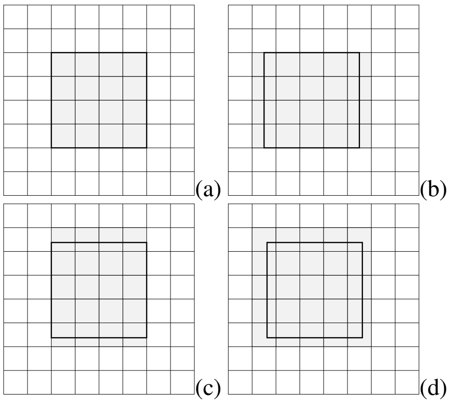

Figure 4: Four cases in reading a subarray from an array. In each of the cases, the thick-lined rectangle represents the subarray and the shaded chunks are the chunks to be fetched. Here, as a chunk is the unit of transfer, the whole chunk is fetched even if only part of it is needed. (a) The subarray is aligned with the chunk boundary in both axes. (b) The subarray is aligned with the boundary in the y-axis. (c) The subarray is aligned with the boundary in the x-axis. (d) None of the axes is aligned.

We now go over the steps of our approach in more detail. First, we need to compute the average number of chunks to be fetched in reading a subarray of a particular shape. For ease of discussion, we use 2D arrays as examples and the analysis can be easily extended to higher dimensions. In [25], Yu gave a complete analysis of the different ways that a region of a given shape can clip an array, as shown in Figure 4, which shows an array of size $a \times a$, decomposed into 64 I/O chunks using an 8×8 mesh and with a requested subarray of size $b \times b$, with $b = \frac{a}{2}$. In (a), the subarray is precisely aligned with the I/O chunk boundary and only 16 chunks have to be fetched for the read request. In (b) and (c), the subarray is aligned with the I/O chunk boundary along one axis but not the other, and 4 more I/O chunks must be fetched. In case (d), both x and y axes are not aligned and 25 chunks have to be fetched in total. More generally, to read a $b \times b$ subarray out of an $a \times a$ array, which is decomposed into a set of I/O chunks with each chunk of size $l \times l$, the probability of the case occurring and the number of chunks to be fetched in each of the four cases are shown in Table 1. In this analysis, for simplicity, we assume that both the array and the subarray are squares. Also, b is divisible by l. All these assumptions can be easily relaxed.

Based on the above analyses, for 2D arrays, Equation 1, which is originally used to estimate the time for reading a particular subarray, can be extended to estimate the average execution time for server i to read subarrays of a given shape:

$$\begin{aligned} T(i) &= \sum_{j=case\ a,b,c,d} (Probability(j) \\ &\times \sum_{1 \leq k \leq NC(i,Fetch(j))} T_{chunk}(i,k)), \end{aligned} \qquad (3)$$

where $NC(i,n)$ is the average number of chunks fetched by server i to read a subarray of a given shape with n chunks[3]. Clearly, $NC(i,n)$ is dependent on the declustering strategy used.

[3] In the following discussions, we assume that subarrays are aligned with the I/O chunk boundaries. If the subarray is initially not aligned, it is expanded until it aligns in all dimensions (Figure 4), because a chunk is the unit of transfer.

Given M I/O servers, let $NC(S) = \max_{1 \leq k \leq M} NC(k,n)$ be the largest number of chunks fetched by any server in reading a particular subarray S of a given shape with n chunks and let NC_{avg} be the average of $NC(S)$ for reading all subarrays of the given shape with n chunks. As stated before, we are only interested in computing NC_{avg}. Fortunately, GDM based declustering approaches have the following property:

Proposition 1 Let A be a D dimensional array decomposed into a set of chunks, which are declustered to M servers using a GDM-based declustering approach. Let S be a subarray of A to be read. Then the maximum number of chunks fetched by any I/O server, $NC(S)$, is fixed, independent of the base of the subarray S.

Proof Let $NC(S1)_i$ be the number of chunks fetched by I/O server i in reading a subarray $S1$ with base $(0,\ldots,0)$ and shape $(s_0,\ldots,s_{D-1})$. Then $\max(NC(S1)_0,\ldots,NC(S1)_{M-1}) = NC(S1)$. By definition, chunk $[c_0,\ldots,c_{D-1}]$ is assigned to server i where $i = (c_0 \times p_0 + \cdots + c_{D-1} \times p_{D-1})\ mod\ M$. The number of chunks read by each server for a subarray $S2$ with the same shape as $S1$, but with the same base as chunk $[c_0,\ldots,c_{D-1}]$, is $NC(S1)_{M-i},\ldots,NC(S1)_0,\ldots,NC(S1)_{M-1-i}$. Clearly, $NC(S2) = max(NC(S1)_{M-i},\ldots,NC(S1)_0,\ldots,NC(S1)_{M-1-i}) = max(NC(S1)_0,\ldots,NC(S1)_{M-1}) = NC(S1)$.

Based on this property, we only need to consider the subarray with base $(0,\ldots,0)$ in computing NC_{avg}, which can be calculated in a short time.

To complete the computation in Equation 3, we also need to figure out how many of the NC_{avg} chunks are to be read sequentially and how many are not, which is dependent on the ordering strategies. Here, we define a *cluster* as a set of chunks within a subarray which can be read sequentially. That is, the number of clusters within a subarray corresponds to the number of disk seeks in reading the subarray. The model assumes that the Hilbert-curve ordering strategy is used to linearize chunks at each I/O server for its higher performance than the axis-traversal ordering strategy (empirical comparisons are shown in Section 4).

Unlike NC_{avg}, the number of clusters within a subarray of the given shape is dependent on the base of the subarray. For example, as shown in Figure 2, the two rectangles filled with different patterns represent two subarrays with the same shape but different bases. Assuming that there is only one I/O server and the chunks are stored on disk according to a Hilbert-curve order, there are three clusters within the shaded subarray and there is only one cluster within the subarray filled with bricks, as illustrated by the arrows where a pair of an in-arrow and an out-arrow across the subarray boundary represents a cluster. That is, to read the former subarray requires three disk seeks, while reading the latter subarray needs only one seek, assuming that the disk head is initially not positioned at the beginning of either subarray. Therefore, we will compute the average number of clusters in a subarray of a given shape, and find its maximum among all servers.

In [18], Moon et al presented analyses of the clustering effects of the Hilbert-curve ordering strategy. Given a region of arbitrary shape, they provided closed-form formulas for the average number of clusters within the region. Our work only considers rectangular regions, called subarrays in our case, which is a special case of their study. Also, in their work, they only considered the case where all chunks are stored on a single disk in a Hilbert-curve order. But in our work, chunks are first declustered to multiple servers and the set of chunks which are assigned to the same disk are then linearized using the Hilbert-curve mapping. That is, our work needs to consider the effect of declustering, as illustrated in Figure 5. There are 5 clusters within the subarray if all chunks are assigned to a single server, but if chunks are assigned to multiple servers, there are 3 clusters within the subarray for the server that is assigned the shaded chunks. So, we extended their work to fit our needs.

Case	Probability()	Fetch()
a	$(\frac{a-b}{l}+1)^2/(a-b+1)^2$	$\frac{b}{l} \times \frac{b}{l}$
b	$(\frac{a-b}{l}+1) \times (a-b-\frac{a-b}{l})/(a-b+1)^2$	$(\frac{b}{l}+1) \times \frac{b}{l}$
c	$(\frac{a-b}{l}+1) \times (a-b-\frac{a-b}{l})/(a-b+1)^2$	$(\frac{b}{l}+1) \times \frac{b}{l}$
d	$(a-b-\frac{a-b}{l})^2/(a-b+1)^2$	$(\frac{b}{l}+1) \times (\frac{b}{l}+1)$

Table 1: The probability and the number of chunks to be fetched in each of the four cases.

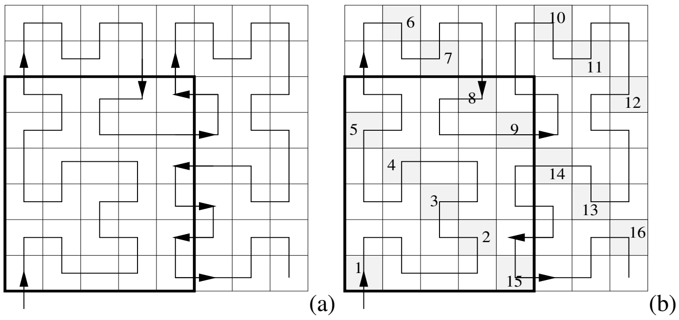

Figure 5: Illustration of the Hilbert-curve ordering. In both figures, the rectangle represents the subarray to be read. (a) One I/O server, all chunks stored on the same disk. (b) Multiple I/O servers, the chunks filled with dots are assigned to the same disk.

For ease of illustration, we first assume that there is only one I/O server, i.e. all chunks are stored on a disk according to the Hilbert-curve traversal. The key idea of the analysis is that each cluster in the subarray generates an entry into the subarray and an exit out of the subarray, as illustrated by the arrows in Figure 5. That is, each cluster in the subarray results in two edges to be cut by the boundary of the subarray. From another viewpoint, for each edge in the Hilbert-curve, we can test if it is cut by the boundary of a subarray of the given shape. Thus, the number of clusters within a subarray is equal to half the total number of edges cut by the boundary of the subarray [18].

Extending the analysis to the case where there is more than one I/O server, we no longer consider the edges in the Hilbert-curve but all contiguous pairs of chunks residing on the same server. We first number the chunks assigned to a server according to their Hilbert-curve mapping value, as shown in Figure 5-(b). Then, for each pair $(i, i+1)$, we count the number of subarrays whose boundary separates chunk i and chunk $i+1$ such that one chunk is inside the subarray, but the other is not. Summing up the number for each pair of chunks and dividing the total by 2 gives the average number of clusters within the given subarray.

3.2.2 Validation of the performance model

We validated the 3D version of the model on an IBM SP2 at the Argonne National Laboratory (ANL). To predict Panda's performance on a particular platform, we need to obtain the system parameters which characterize the platform. We used a microbenchmark for this purpose and Table 2 is a brief summary of the current configuration of the ANL SP2. More details of the installation can be found on line at http://www.mcs.anl.gov/CCST/computing/quad/.

To validate the model, we compare Panda's actual performance with the model's output. We use 16 compute processors and vary the number of I/O servers (4, 8, 12 and 16), the I/O chunk sizes (64 KB to 1 MB), and the sizes and shapes of the arrays and the subarrays. The first set of experiments read 3 different subarray shapes ($256 \times 256 \times 192$, $256 \times 256 \times 256$ and $320 \times 256 \times 256$) out of a $512 \times 512 \times 512$ array of floating points. For each subarray shape, 20 subarrays, with the base randomly generated, are tested. The second set of experiments read the same subarray shapes as in the first experiment, but out of a $512 \times 512 \times 1024$ array. Again, 20 subarrays are randomly generated for each subarray shape. For each configuration, we measured the average Panda performance in reading 20 arrays and use the model described above to predict Panda's performance to read a subarray of the given shape. Table 3 shows the normalized difference between the predictions of the model and Panda's actual performance for each set of experiments, computed by $100 \times \frac{(predicted\ time - actual\ time)}{actual\ time}$. As can be seen from the table, in all cases, the prediction is within 7% of the measured performance result. That is, the model can accurately predict Panda's performance.

General information	
Number of nodes	80
Processor type	Power2 Super Chip
Processor speed	120 MHz
Main memory	256 MB
File system peak performance	
Scratch space	2 GB
Disk partition occupancy	4%
Peak JFS writes (measured)	6.9 MB/sec
Peak JFS reads (measured)	7 MB/sec
Message passing performance	
Interconnect	High-performance switch
Latency	31 microseconds
Peak Bandwidth (measured)	90 MB/sec

Table 2: The current hardware and software configurations of the ANL SP2

3.2.3 Choose an appropriate chunk size using the model

With the help of the model, we can choose an appropriate I/O chunk size for snapshot output by first picking a set of potential I/O chunk sizes (how to pick the set will be described later), using the model to predict Panda's performance for each alternative, and finally choosing the chunk size which has the highest predicted performance. However, the performance model described above can only predict Panda's performance in reading/writing a subarray of a given shape. To precisely predict Panda's performance for a wide variety of access patterns with different subarray bases and shapes, we need to repeatedly use the model to predict Panda's performance for all different shapes and average the predicted results, which is still time consuming even though it is much faster than using the old model and enumerating all combinations with different bases

		512 × 512 × 512 array				512 × 512 × 1024 array			
subarray	chunk size	4 I/O	8 I/O	12 I/O	16 I/O	4 I/O	8 I/O	12 I/O	16 I/O
$256 \times 256 \times 192$	64 KB	2.516	-2.859	-5.684	-3.358	3.381	2.04	-0.209	2.303
	128 KB	2.888	-2.213	-6.205	-4.884	3.707	0.202	-1.812	-1.875
	256 KB	2.411	-2.19	0.074	-3.753	4.737	2.328	2.753	-1.016
	512 KB	2.187	-3.527	-2.38	-4.602	3.678	5.097	0.005	-1.552
	1 MB	2.343	2.411	0.267	-0.888	3.324	1.608	0.536	-1.18
$256 \times 256 \times 256$	64 KB	2.198	-2.077	-3.377	-4.302	2.767	1.183	-1.148	1.592
	128 KB	3.107	-0.826	-4.293	-2.608	1.907	-1.699	-4.801	-3.325
	256 KB	2.257	-3.799	0.477	-3.921	3.692	-0.2	0.71	-0.716
	512 KB	1.208	-2.044	-6.823	-6.101	3.951	2.034	0.132	-1.387
	1 MB	3.711	-2.437	2.119	-0.47	5.436	-1.599	0.943	-0.351
$320 \times 256 \times 256$	64 KB	0.096	-4.421	-6.792	-2.621	2.132	0.061	-2.52	3.171
	128 KB	4.489	1.12	-1.549	1.295	2.042	-0.774	-2.851	-1.054
	256 KB	4.289	1.553	0.601	-0.939	3.015	2.427	-0.779	1.422
	512 KB	3.856	0.462	-0.159	0.009	2.796	3.864	-0.453	0.338
	1 MB	3.513	2.949	0.693	1.515	3.588	2.211	0.183	1.479

Table 3: The normalized difference between the model-predicted and Panda's actual performance.

and shapes. Therefore, in choosing a proper chunk size for snapshot outputs, we do not try all different subarray shapes, but only selected samples. In the current implementation, we tested $\frac{1}{3^D}$ different shapes for a D dimensional array. For each dimension, we tested three possibilities, $\frac{1}{4}$, $\frac{1}{2}$ and $\frac{3}{4}$ of the array size in that dimension.

Then, how do we pick a set of potential I/O chunk sizes for testing? In the experiment, we first consider the smallest chunk size that gives peak performance for writing out large snapshots on the SP2 (1 MB). We then repeatedly reduce the size[4], which normally results in higher predicted performance, until the predicted performance starts to decrease. The I/O chunk size is then set to the size which has the highest predicted performance found so far. More precisely, we choose the largest chunk size whose performance is within a certain factor (3% in our experiments) of the highest performance found so far. The reason to choose a larger, near-optimal chunk size is to balance the read performance against the write performance, as larger chunk sizes will result in higher write performance on the SP2.

3.3 Scheduling I/O

On the SP, in order to reduce the amount of redundant data fetched in a snapshot_read operation, our methodology generally results in decomposing a large array into chunks of moderate sizes, like a few hundred KBytes. This decomposition policy causes a communication contention problem, as illustrated in Figure 6, which shows an array decomposed into an 8×8 mesh of I/O chunks, distributed to 4 I/O servers, using the Hilbert-curve ordering strategy to linearize the set of chunks assigned to each disk. Also, the array is decomposed into a 2×2 mesh among 4 compute processors. As can be seen from Figure 6-(a), in a snapshot_write operation, if all I/O servers gather chunks in Hilbert-curve order, then all I/O servers will try to get data from compute processor 0 for the first set of chunks, then from compute processor 1, and so on. That is, I/O servers will compete with each other in gathering chunk data. Similar contention can occur for reads and for other ordering strategies.

[4] To prevent potential load imbalance, we reduce the size to the next smallest number for which the number of I/O chunks is a multiple of the number of I/O servers.

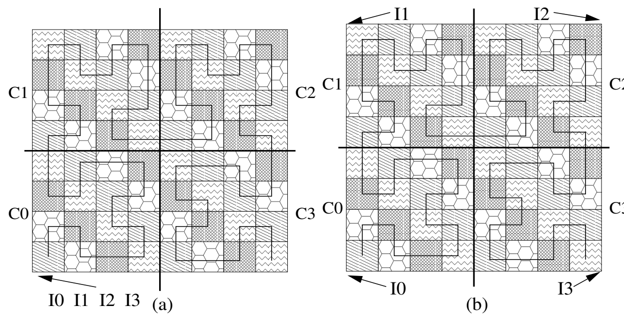

Figure 6: Scheduling I/O: (a) No scheduling of communications. All I/O servers start to gather the smallest-indexed chunks. (b) Scheduling is performed. Each I/O server starts to gather chunks from different parts of the array.

Scheduling I/O has been discussed in [7] to schedule the transfer of fine-grained data between compute processors and I/O servers. When an array is distributed among the compute processors with fine granularity, but distributed among the I/O servers with coarser granularity, each I/O server will communicate with all compute processors to gather each chunk. If not properly scheduled, all I/O servers may try to talk to the same compute processor at the same time. The solution to the problem in [7] was to change the order of communications in gathering an I/O chunk. That is, for example, I/O server 0 would request pieces of the I/O chunk owned by compute processor 0 first; while I/O server 1 would request pieces of the I/O chunk owned by processor 1 first, etc. That is, this scheduling does not change the order in gathering the I/O chunks at each I/O server, but the order in gathering pieces of data of an I/O chunk.

In designing a good layout for range retrievals, as arrays are usually finer-grained on disk than in memory (in order to reduce the amount of redundant data fetched in a snapshot_read operation), gathering an I/O chunk typically requires communication with only one compute processor. Therefore, changing the order in getting pieces of the I/O chunk does not help. The communication contention problem has to be solved by performing another kind of reordering. For example, in the above example, if each I/O server can start gathering data from different parts of the array, as illustrated by the arrows shown in Figure 6-(b), the contention problem can be avoided. Therefore, we added a new optimization in Panda to schedule the communications between compute processors and I/O servers to avoid having multiple I/O servers communicate with the same compute processor at the same time. For example, in a write operation, for the set of I/O chunks that an I/O server will gather, instead of directly ordering them based on the chunks' Hilbert indexes, chunks are first separated into several groups according to the id of the compute processor which owns the largest portion of each chunk. Within each group, chunks are ordered according to their Hilbert index. Each I/O server then gathers or scatters its data using a different group order. For example, for a write with 4 compute processors and 4 I/O servers, server 0 can gather data from group 0, then 1, 2 and 3; I/O server 1 can gather from group 1, then 2, 3 and 0; etc. Because of the reordering, the communication contention problem is lessened to a degree that there is almost no contention in gathering an array.

4 Performance Results

As we have seen, there are several factors to be considered in designing a good disk layout for snapshot outputs - the ordering strategy, the I/O chunk size, the declustering strategy, and I/O scheduling. For each snapshot request, to choose the globally optimal organization with all the factors considered simultaneously is a complex optimization problem which requires a search-based solution approach, resulting in long optimization time. Therefore, in this work, we only wish to determine which strategies produce the highest average performance for each factor. In the first set of experiments, when comparing the different approaches for each factor, we fix the settings used in other factors. For example, in comparing the performance of various ordering approaches, we fix the declustering strategy to be GDM_Cartesian and fix the I/O chunk size to be the optimal size picked by the model, etc. In the second set of experiments, we show the cumulative effect of employing all selected strategies, which give the highest average performance for each factor respectively, together.

Our experiments were conducted on an 80-node IBM SP2 at ANL (Table 2). In all experiments, we used 16 compute processors and experimented with both arrays and grid hierarchies. For the array data structure, we used a $512 \times 512 \times 512$ (512 MB) array, and a $4 \times 2 \times 2$ memory mesh. For the grid hierarchy structure, we tested a 624 MB grid hierarchy, decomposed among 16 compute processors using a decomposition strategy based on a space-filling curve [20]. For snapshot_write operations, we wrote the entire array or the grid hierarchy to disk and measured the elapsed time in writing out data. For snapshot_read operations, the response time is the average for retrieving 20 randomly chosen subarrays (the same 20 subarrays for all array experiments and a different set of 20 subarrays for all grid hierarchy experiments), with both the base and the shape picked randomly in all three dimensions. When reading a subarray of a grid hierarchy, the overlapped regions from all grids across all levels will be fetched. Retrieving all grids allows a post-processing module to perform an interpolation routine to produce only a single rectangular grid that is interpolated out of the grid hierarchy. Panda does not provide interpolation facilities. In all performance graphs, aggregate throughput is calculated by dividing the response time of the slowest server by the amount of data requested. For a read operation, this may be smaller than the amount read (this is why read performance is lower than for writes). The "fraction of peak throughput" shown in the graphs is calculated by dividing the aggregate bandwidth by the peak potential file system throughput (Table 2) times the number of servers. Also, all the results are the average of 4 runs, and the error bars correspond to a 95 percent confidence interval for the mean.

4.1 Ordering strategy

We compared the performance of the axis-traversal strategy and the Hilbert-curve ordering strategy and Tables 4 and 5 present the average number of seeks to read 20 randomly chosen subarrays out of a 512 MB 3D array and a 624 MB 3D grid hierarchy respectively. When fewer I/O servers are used, the performance impact of different ordering strategies is more significant, due to the larger number of chunks to be fetched from each server. For example, when using 4 I/O servers, the axis traversal strategy can require 70% more seeks than the Hilbert strategy. However, when servers have fewer chunks each, the penalty for using the axis traversal strategy is in the 5-15% range.

Ordering strategy	Number of I/O servers			
	4 I/O	8 I/O	12 I/O	16 I/O
Axis traversal	85	38.1	26.5	24.25
Hilbert mapping	49.8	33.6	25	21.45

Table 4: The average number of seeks generated by the axis-traversal and the Hilbert-curve ordering strategy to read 20 subarrays out of a 512 MB array. Here, GDM_Cartesian is used to decluster chunks and the chunk size is picked by the performance model.

Ordering strategy	Number of I/O servers			
	4 I/O	8 I/O	12 I/O	16 I/O
Axis traversal	97.8	56	63.8	33.9
Hilbert mapping	78	52.5	58.2	30.2

Table 5: The average number of seeks generated by the axis-traversal and the Hilbert-curve ordering strategy to read 20 subarrays out of a 624 MB grid hierarchy. All the other configurations are the same as in Table 4.

4.2 Choosing an optimal chunk size

We measured the performance for both snapshot_write and snapshot_read operations using different I/O chunk sizes, ranging from 64 KB to 1 MB. Figure 7 shows the results for the 512 MB array, with the model-picked chunk sizes shown as the labels in the graph; Figure 8 shows the results for a 624 MB grid hierarchy. For the grid hierarchy experiments, the same I/O chunk size is applied to decompose each grid in the hierarchy except the "model-picked" bars, for which the model chooses a separate chunk size for each grid in the hierarchy.

The graphs show that when writing array snapshots, performance degrades by up to 8% as the chunk size decreases, because more chunks mean more time to process and gather. When taking snapshots of a grid hierarchy, the effect of using different chunk sizes is not so significant. This is because each grid in the hierarchy is decomposed and declustered separately. Using a larger chunk size results in fewer chunks to be declustered, and therefore the amount of data assigned to each server is likely to be unbalanced. The effect of imbalance is not significant in writing out a single grid, but is visible cumulatively. That is, using a larger chunk size results in less-balanced data allocations among the servers, but

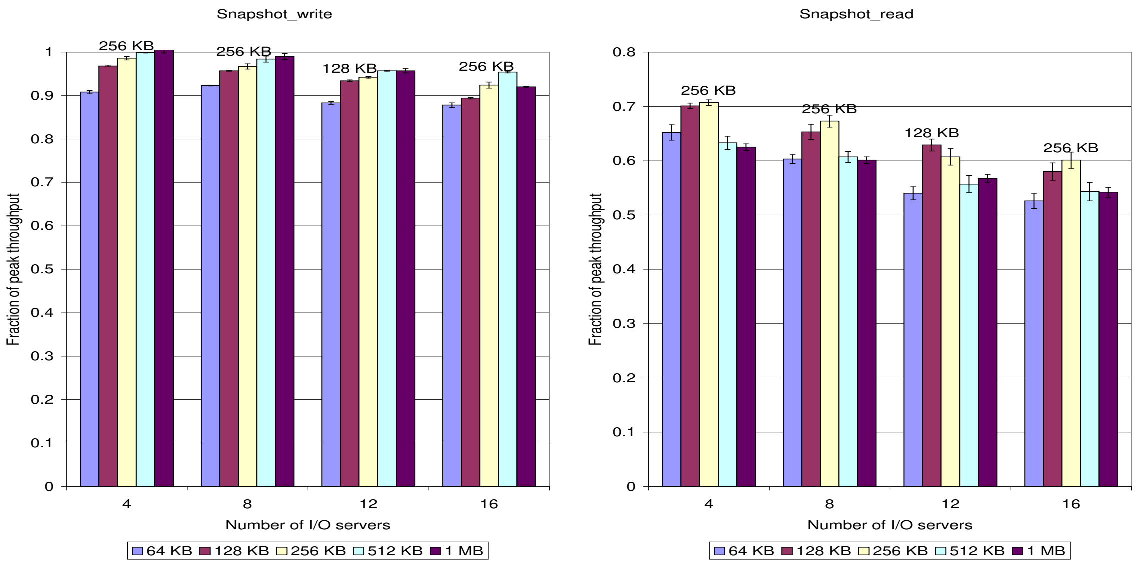

Figure 7: Effect of I/O chunk size on the performance of snapshot_write and snapshot_read operations on a 512 MB array, using the GDM_Cartesian declustering strategy and the Hilbert-curve ordering strategy. The labels shown in the graph give the chunk size picked by the model for different configurations.

reduces the number of chunks to be processed and gathered. Using a smaller chunk size has the opposite effect.

For snapshot_read operations, the trends are more complex, as multiple performance factors are involved, as explained in Section 3.2. With a poor I/O chunk size, read performance drops as seeks increase or more unwanted data is fetched. As suggested by the performance model validation and confirmed in the graphs, the model can successfully pick an appropriate chunk size.

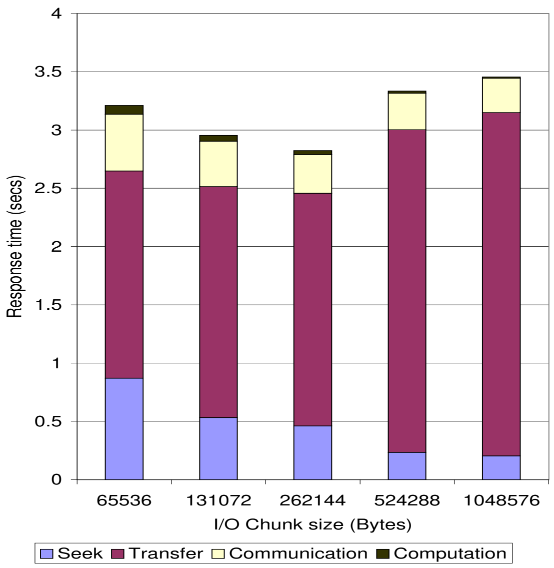

Figure 9: Breakdown of the response time to read a randomly chosen 128 MB subarray with 8 I/O servers.

To show the effect of these factors, Figure 9 decomposes the response time for a sample subarray read request into four parts: computation - the time in processing chunks; transfer - reading data from disks to servers; communication - moving data from servers to compute processors; and seek overheads. Clearly, the seek overhead increases as the chunk size decreases, as more chunks result in higher chances of seeks. The transfer time increases as the chunk size increases, as using a larger chunk size results in higher chances of accessing more unwanted data. The computation time increases as the chunk size decreases, as more chunks mean more time to process, though normally, the computation time is negligible in the total response time. Also, the communication time increases slightly as the chunk size decreases, due to the larger number of smaller messages; however, the difference is not significant. Thus, in general, the optimal chunk size occurs when the extra seek and computation overheads can be offset by a reduction in unwanted data fetched.

4.3 Scheduling I/O

We measured the performance of snapshot_write operations with and without I/O scheduling and the results are shown in Figure 10. Snapshot_read performance is not presented as scheduling I/O is not as important a performance factor as seek overheads and redundant reads.

Figure 10 shows that the importance of scheduling grows when the number of servers increases, as more I/O servers means more competition among the servers. When 12 I/O servers are used for the array, scheduling has a particularly large performance impact. This is because the model picked 128 KB as the optimal I/O chunk size when 12 I/O servers are used and picked 256 KB bytes when 16 servers are used. Thus, there are more data messages to be transferred between the compute processors and I/O servers when 12 servers are used and more communication contention may occur.

For some configurations, the performance gain in scheduling I/O is less significant for grid hierarchies than for arrays. The reason is that scheduling is applied to each grid separately. Thus, I/O servers working on different grids may still try to talk to the same compute processor in getting chunk data from different grids. A

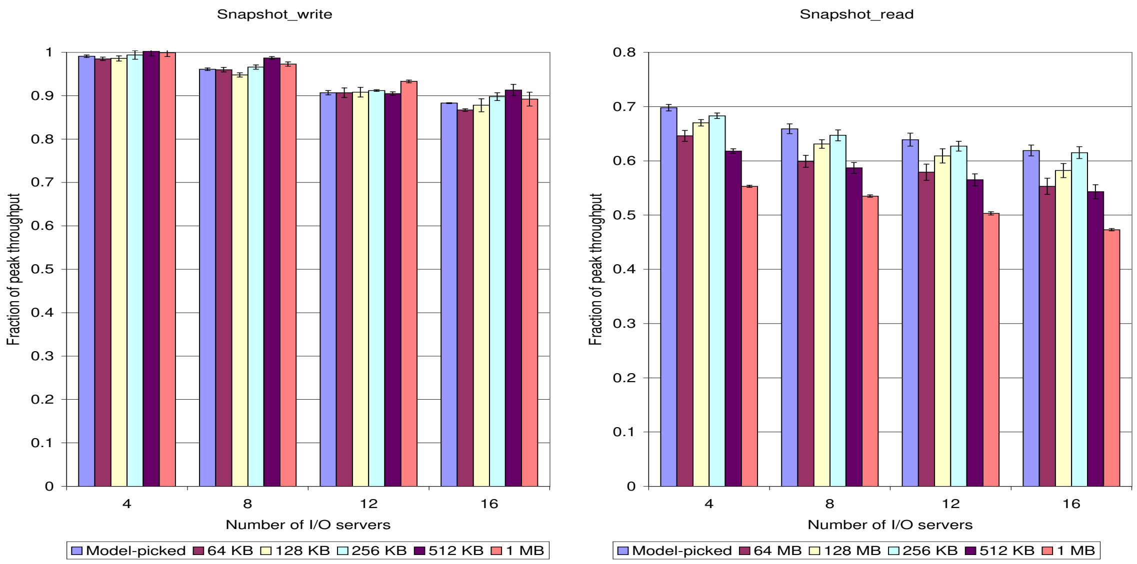

Figure 8: Effect of I/O chunk size on the performance of snapshot_write and snapshot_read operations on a 624 MB grid hierarchy. All other settings are the same as in Figure 7.

global scheduling function can relieve this problem.

4.4 Putting it all together

In this section, we show the cumulative effect of employing all strategies - the GDM_Cartesian declustering strategy, the Hilbert-curve ordering strategy and a model-picked chunk size - together. We chose Panda 2.1's default settings as a comparison, which use the Round-Robin declustering strategy, the axis-traversal ordering strategy, and a 1 MB chunk size. Figure 11 shows the results of the comparisons for both snapshot_write and snapshot_read operations on a 512 MB array.

For snapshot_write operations, using Panda's default settings performs a little better ($< 2\%$) than using the new strategies. The main reason for the difference is the choice of the I/O chunk size - 1 MB versus 256 KB or 128 KB. As illustrated before, as the chunk size decreases, the computation time to process the chunk increases and the chance of communication contention increases. By using the I/O scheduling algorithm, the communication contention overhead can be successfully minimized so that the write performance does not drop a lot compared to using the default setting.

For snapshot_read operations, the combination of all the strategies provides a significant performance boost, 20-25%. This is because employing the strategies together can reduce the total amount of unwanted data accessed (using good chunk size), balance the amount of data to be fetched from each disk (using the GDM declustering algorithm) and minimize the seek overheads (using the Hilbert-curve ordering). Based on the performance of both write and read operations, we conclude that an extra overhead of less than 2% in write operations is worth the 25% gain in performance for read operations.

5 Related Work

Previous work in declustering, ordering, scheduling and choosing an optimal chunk size has been described in Section 3 and will not be repeated here. This section focuses on multi-dimensional databases - what techniques have been proposed to achieve better visualization performance.

Several database systems have been developed to manage multi-dimensional datasets [1, 2, 3, 9, 22]. Most of these systems were designed for geographic information systems (GIS) on sequential machines. RasDaMan [1] is a DBMS designed to manage multi-dimensional discrete data. Their storage support combined tiling and a spatial index (R-tree) to access tiles. Paradise [9] used a client-server architecture with support for tiling and compression for 2-D raster data. Panda differs from these two systems in that we target dense arrays and grid hierarchies rather than sparse datasets. The issues we considered in this paper - declustering, ordering, scheduling and choosing an appropriate tile size - were not addressed in these systems; some of our approaches could be applied to their target datatypes. However, both systems provided a SQL-like query language to allow flexible retrieval, which is not yet supported in Panda. [22] enhanced the POSTGRES DBMS to support multidimensional array data, one of Panda's target datatypes. They proposed using chunked data layouts on disk (one chunk per disk block) to store the arrays on sequential machines and chose an optimal chunk layout based on the actual queries to be issued by scientists of the Sequoia project [24]. Our approach differs in that we assume that future access patterns are unknown at the time that the data layout must be selected. Also, we view chunk size as a parameter to be optimized.

Titan [3] is a parallel remote sensing DBMS that provides efficient support for declustering and placement of multi-dimensional datasets on disks. Its storage system first partitions the dataset into a set of 128 KB chunks (the size was chosen empirically), and then applies the *minimax spanning tree* algorithm [17] to decluster chunks to disks and the *short spanning path* algorithm [12] to order chunks on each disk. T2 [2] is a customizable parallel database that generalizes Titan. Our work adds an analytical methodology for tailoring chunk size to the dataset at hand, an I/O scheduling methodology, and uses a different ordering strategy, all of which could be employed in Titan or T2.

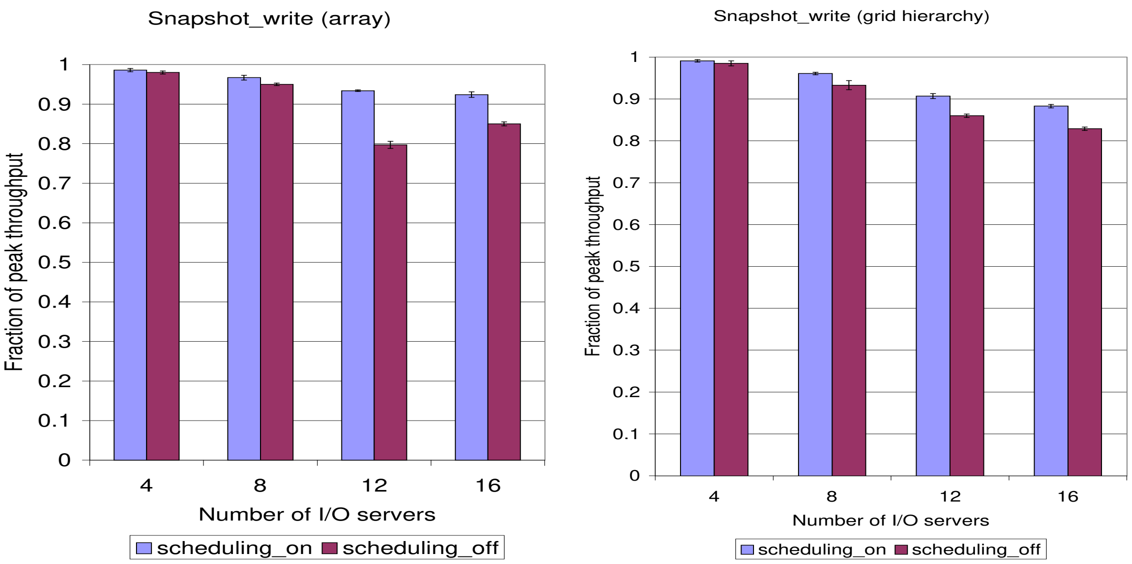

Figure 10: Effect of I/O scheduling on the performance of snapshot-write operations on a 512 MB array and a 624 MB grid hierarchy with 4, 8, 12 or 16 I/O servers, using the GDM_Cartesian declustering strategy, the Hilbert-mapping ordering strategy and an optimal chunk size picked by the performance model.

6 Conclusions and Future Work

In this paper, we showed how a Hilbert-curve ordering strategy can be used to order chunks on each disk to reduce the average number of seeks required to service a range query for a visualization application. We developed a performance model which can be used to pick an appropriate I/O chunk size to balance the overheads between the extra seeks required to read smaller chunks and the unwanted data read when larger chunks are used. We also proposed a scheduling algorithm which can reorder the transfer of data between compute processors and I/O servers so that multiple servers will not try to talk to the same compute processor at the same time. In the context of the Panda parallel I/O library on the SP, we showed that these strategies deliver higher performance than previously existing approaches. In some cases, the performance impact of a single strategy can be dramatic, but more typically the use of a single strategy results in a 5-15% performance improvement. However, the cumulative effect of employing all strategies together, along with our GDM_Cartesian declustering strategy, is quite large (20-25%). Further, the strategies are orthogonal to one another, so that they can be mixed and matched with preexisting strategies embedded in other parallel I/O systems.

References

[1] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. Geo/environmental and medical data management in the RasDaMan system. In *Proceedings of the 23th International Conference on Very Large Data Bases*, pages 548–552, 1997.

[2] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.

[3] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a high-performance remote-sensing database. In *Proceedings of the Thirteenth International Conference on Data Engineering*, April 1997.

[4] L. Chen and D. Rotem. Declustering objects for visualization. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 85–96, 1993.

[5] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization in Panda. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[6] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam, and K.E. Seamons. Performance modeling for the Panda array I/O library. In *Proceedings of Supercomputing '96*, November 1996.

[7] Y. Cho, M. Winslett, Y. Chen, and S. Kuo. Parallel I/O performance of fine grained data distributions. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 163–170, 1998.

[8] N. Chrisochoides, I. Kodukula, and Keshav Pingali. Compiler and run-time support for semi-structured applications. In *Proceedings of the 11th ACM International Conference on Supercomputing*, 1997.

[9] D. Dewitt, N. Kabra, J. Luo, J.M. Patel, and J.B. Yu. Client-server Paradise. In *Proceedings of the 20th VLDB Conference*, 1994.

[10] H. Du and J. Sobolewski. Disk allocation for cartesian product files on multiple-disk systems. *ACM Transactions on Database Systems*, 7(1):82–101, March 1982.

[11] C. Faloutsos and P. Bhaywat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, 1993.

[12] M. Fang, R. Lee, and C. Chang. The idea of declustering and its applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 181–188, 1986.

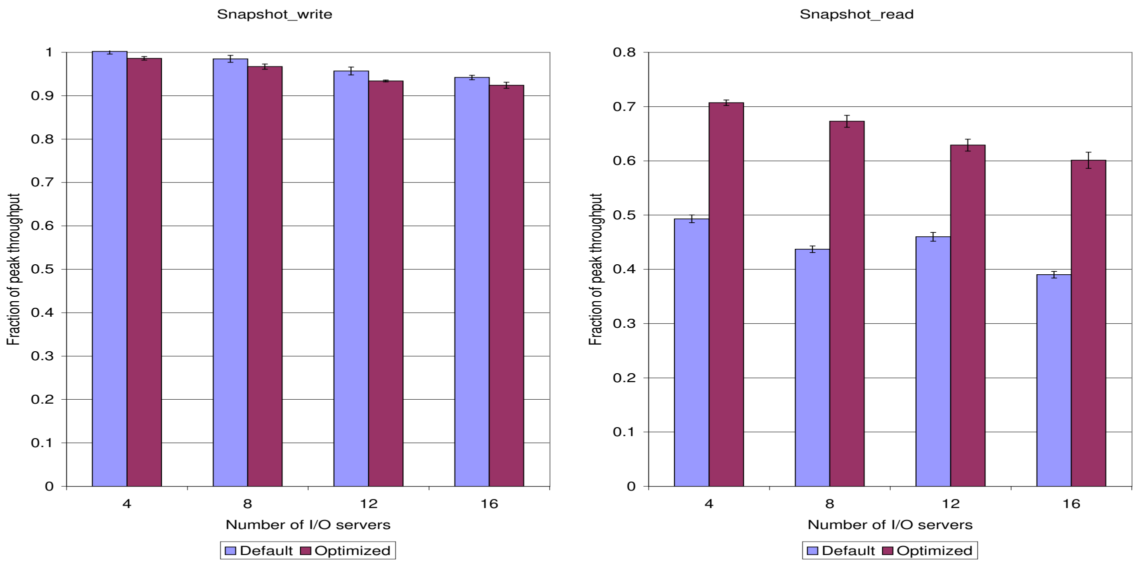

Figure 11: Effect of combining all strategies together on the performance of snapshot-write and snapshot-read operations on a 512 MB array.

[13] H. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 332–342, 1990.

[14] M. Kim and S. Pramanik. Optimal file distribution for partial match retrieval. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1988.

[15] S. Kuo, M. Winslett, Y. Chen, and Y. Cho. Efficient I/O of grid hierarchies for AMR computations on parallel disks. In *Proceedings of the 10th International Working Conference on Scientific and Statistical Database Management*, pages 12–21, 1998.

[16] S. Kuo, M. Winslett, and Y. Cho. New declustering methods for parallel range queries. Available on line at http://drl.cs.uiuc.edu/panda/.

[17] B. Moon, A. Acharya, and J. Saltz. Study of scalable declustering algorithms for parallel grid files. In *Proceedings of the 10th International Parallel Processing Symposium*, 1996.

[18] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. Technical Report CS-TR-3611, University of Maryland, March 1996. Submitted to IEEE Transactions on Knowledge and Data Engineering, 1996.

[19] J. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 326–336, 1986.

[20] M. Parashar and J. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, January 1996.

[21] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. Abbadi. Cyclic allocation of two-dimensional data. In *Proceedings of the 14th International Conference on Data Engineering*, pages 94–101, 1998.

[22] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the 10th International Conference on Data Engineering*, pages 328–336, 1994.

[23] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[24] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 storage benchmark. In *ACM SIGMOD International Conference on Management of Data*, 1993.

[25] J. Yu and D. DeWitt. Processing satellite images on tertiary storage: A study of the impact of tile size on performance. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, 1996.