An Iteration Aware Multidimensional Data Distribution Prototype for Computing Clusters

Baoqiang Yan and Philip J. Rhodes
Department of Computer and Information Science
University of Mississippi
{baoqiang, rhodes}@cs.olemiss.edu

Abstract

Disk and network latency must be taken into account when applying parallel computing to large multidimensional datasets because they can hinder performance by reducing the rate at which data can be fed to the compute nodes. Existing methods aggregate some number of data requests from cluster nodes to improve overall performance by reducing the number of latency penalties. However, an even more significant reduction can be achieved by taking advantage of prior knowledge of the access pattern expressed as an iteration.

Within the context of the Granite Scientific Database system, we created a new iteration aware data distribution system that accelerates data transfer between a data server and the client cluster. This system reduces both disk and network latency by transforming a large number of small requests into a small number of large requests that fill an n-dimensional cache block on the cluster head node.[1]

1. Introduction

As new data gathering and generation methods produce rapidly increasing dataset volumes, parallel computing has been and will continue to be an effective method of satisfying researchers' appetite for processing capacity. However, improvements in disk and network latency have not kept pace with the progress seen in processing power, storage capacity, and bandwidth. Thus, with idle processors hungering for data, disk and network latency prevents us from taking full advantage of a computer system's processing power. This problem is made worse with multidimensional datasets because according to traditional file system semantics, a separate system call is required for each disjoint portion. Multidimensional subsets of a data volume must be mapped to a large number of one dimensional disk requests. If data is stored remotely, many separate network transactions must be made.

Thus, in an I/O intensive parallel computing environment without an efficient data distribution among compute nodes, parallelism will not be fully realized. We can therefore improve performance by creating a mechanism that can represent the multiple data requests made by compute nodes and transform them into a more efficient pattern for disk or network access.

In a cluster computing environment, this mechanism must be able to overlap I/O operation with computation and must be collective so that separate data requests can be grouped into small numbers of longer sequential disk reads. Support for efficient spatial data extraction is necessary since scientific applications often center around computation on large multidimensional arrays [11]. This data distribution system should also support load balancing and minimize the intercommunication among the compute nodes themselves to prevent network latency and bandwidth from becoming a bottleneck.

Each of these key requirements have separately been the subject of intense research but the work presented here combines all of them at once to support efficient parallel access to multidimensional datasets. This work, done within the *Granite Scientific Database* system [9], combines the benefits of both collective I/O and informed prefetching by assuming that the pattern of access is known in advance, and that it can be expressed succinctly in the form of an iterator.

Our previous work on both local and remote large multidimensional data access shows that full prior knowledge of future access patterns can be effectively exploited to achieve dramatic performance gains in I/O by minimizing disk reads and network transaction numbers and eliminating repeated reloading of data blocks [9, 10]. This provides a strong basis for our data distribution system for cluster computing. In our *iteration aware* data distribution system, the iterator

[1] This work was supported in part by the National Science Foundation under grant CCF-0541239

that Granite relies on to iterate through an *n*-dimensional data space also serves as an integrated descriptor of the access pattern for each compute node as well as the aggregated access pattern employed by the head node. This aggregated pattern is used to create a multi-dimensional cache block that improves performance by reducing the number of times disk or network latency penalties are paid.

After a brief review of previous work on I/O in parallel computing environment, we will give an introduction to the Granite system, focusing on the issues that are related to the efficient data extraction and distribution. We will then describe the framework of our data distribution system. A sample application that implements this framework is demonstrated and its experimental results are provided to confirm our arguments. We end with conclusions and future work.

2. Background

The ever increasing discrepancy between processing capacity and storage I/O hinders the full realization of the computing power. Finding an efficient way to hide disk and network latencies will effectively improve the utilization of existing bandwidth and the data transfer rate to fast processing unit.

According to Gibson [2], there are only four techniques available for solving disk performance problems.

- increasing storage device parallelism
- more effective caching
- overlapping I/O with computation through prefetching
- more effective scheduling by reducing or rearranging data accesses

To address network latency, improvements can be made in the data transfer protocol, or multiple small requests are packed into a small number of large requests to reduce the number of times the latency cost is paid.

2.1. Parallel I/O system

Similar to the idea of introducing parallelism to increase computing speed, scalable I/O has been a topic of intense research interest and forms the basis of many high-performance computing systems [1].

The SIO Study [14] and the work of Nils and Kotz [6] tried to characterize file access in parallel scientific workloads. This provides some guidelines for high-performance parallel file system designers. Both of these studies concluded that a general parallel file system has to deal with both small accesses and large ones. Since performance for small, noncontiguous accesses is much worse than large accesses, the file system interface must change because it forces the programmer to break down parallel I/O activities into small, disjoint requests.

Little research has yet been done on I/O access patterns for parallel computing systems and there is no single, coherent model available. Most existing parallel file systems are architecturally dependent [4, 6]. Portability, if realized, is done through an abstract level atop multiple parallel file systems, such as ADIO in ROMIO [15], a portable implementation of MPI-IO. Similarly, PPFS [3] is a user-level library that abstracts away platform differences and allows various configuration tunings for experimental performance evaluation. However, it is reported that such systems have average I/O performance dramatically less than their reported peak performance [1, 5]. Parallel I/O system needs to be further standardized. Smirni [13] emphasizes that asynchronous and collective operations are imperative. ROMIO [15] is very effective in incorporating both types of operations through non-blocking I/O and generalized two-phase I/O.

Although Granite does not yet support parallel I/O devices, it is a portable Java interface, and its iterator representation of access pattern is naturally suited to collective I/O via its *n*-dimensional caching mechanism, described in section 3.3. In contrast to systems that conduct caching via a one dimensional view of the file on disk, Granite's *n*-dimensional data view allows the easy construction of caches tuned to an *n*-dimensional access pattern.

2.2. Prefetching and caching

Prefetching can be an effective way to hide or minimize the cost of I/O stalling since it overlaps I/O operations with computation. Prefetching has widely been used in uniprocessor file systems. For parallel file system, PPFS [3] allows user to specify desired prefetching policy and it could be an aggregated one. Kotz *et al.* [4] presented a double predictor strategy, in which they use separate predictors for local-pattern and global-pattern work loads. They judge a prefetching policy to be practical when it is both effective, choosing the correct blocks to prefetch and efficient, having low overhead. This criterion is similar to our well-formed and practical cache block construction [9].

The accuracy of a prefetching policy will heavily rely on the correctness of the hints supplied to the prefetcher module. While other methods try to predict future access pattern based on the history of application's I/O behavior, Granite takes advantage of full prior knowledge of future access pattern to generate caches that only contain the requested data and win best performance by making disk reads as large as possible using its *rod storage model*. In a cluster environment, this prior knowledge of access pattern can be further used to accurately build an aggregated cache block that contains the prefetched data for all the compute nodes. Unlike page based caching systems, Granite's cache blocks are *n*-dimensional, making it particularly effective for efficient processing of spatial data.

Kotz *et al.* [4] use a prediction approach under the assumption that the complete access pattern is not available in advance. However, applications such as the Fast Fourier Transform, Wavelets, feature detection and the visualization demonstrated in this paper have access patterns that are know *a priori* and can be described using a Granite iterator. In such cases, the next access is known with certainty and does not have to be guessed.

2.3. Collective I/O

Collective I/O has won much reputation in the data intensive parallel computing world. It is essentially an optimization of parallel I/O that combines application-specified information and system-level support to achieve an optimal scheduling that reduces disk latency and increases throughput. The underlying principle of collective I/O operations is to aggregate multiple requests into a single long I/O request, thereby minimizing the number of disk accesses. It does this by sorting requests from multiple processes according to the location of the data on disk, which requires (non-portable) operating system level support.

Collective I/O can be categorized as user-level or system-level. Examples of user-level methods include *two-phase I/O*, while system-level methods include *Disk Directed I/O (DDIO)* [5]. Both involve request sorting, but in two-phase I/O [8] the permutation phase does not overlap with I/O and it requires additional buffer space for the intermediate request shuffling, while DDIO allows the I/O nodes to sort the physical block request and transfer the requested blocks directly to the requesting processors.

Collective I/O is generally effective in that it is suitable for many parallel access patterns and assumes no specific physical data layout on disk. However, collective I/O does not overlap with computation since there is no prefetching involved. Collective I/O also assumes the availability of a high-performance message passing network because it requires a lot of communication on either client or server side to realize the real-time sorting among the requests.

2.4. Informed prefetching for collective I/O

According to [5], further optimization of collective I/O can be achieved by integrating prefetching with collective I/O, which brings the benefit of overlapping I/O with computation. Also, the problem of non-portable OS support required for request sorting can be avoided by using a more general framework which uses application supplied hints to inform the file system of future requests and relies on the file system disk queuing mechanism to implicitly do the request reordering according to the physical layout of the job blocks. This reordering delegation method is sensitive to the prefetching depth which takes effect by affecting the depth of the disk queues. The deeper the queues, the more reordering is possible and the higher the throughput. Since the prefetching depth has to be built up gradually based on previous data references, this method has a warming up phase that is used to gather enough hints to form deeper prefetching.

2.5. Storage optimization of data on disk

All the methods mentioned above assume that work is done on original datasets. But instead of adjusting application behavior or optimize disk scheduling to match the storage layout of original data, I/O performance can be improved in another way by changing data storage layout to suit the application requirement. For spatial scientific datasets, perhaps the best known method is chunking [11]. Chunking reorganizes a dataset into n-dimensional chunks according to the expected access pattern. Chunks, which may correspond to job blocks in a parallel context, are now units of data retrieval. Each chunk can be read with a single read operation.

Although Sarawagi et al. describe a generic chunking that does not require prior knowledge of the access pattern, the best results are obtained when the file is reorganized to suit a particular pattern. Unfortunately, for extremely large datasets it is impractical to make a copy of the dataset for each expected access pattern.

2.6. The advantages of granite iteration aware data distribution system

With the exception of support for parallel I/O devices, our Granite iteration aware data distribution system for cluster computing includes the advantages of all other methods discussed above while avoiding their shortcomings.

First, armed with full prior knowledge of future access pattern, our spatial prefetching excludes any speculation about future access pattern that is common to hint-based prefetching techniques. This elimination of guessing completely avoids loading the wrong data. It also greatly improves I/O performance since explicit knowledge of the access pattern allows optimal scheduling that requires the least number of disk reads or network transactions. Such scheduling can be done without relying upon non-portable OS level support. This accurate prefetching can be done as deeply as available memory allows without any warm-up phase as in hint-based methods. Lastly, our data distribution system is a special version of collective I/O that does not employ dynamic data request reordering and thus can provide efficient performance without a high-performance message passing network.

3. Granite scientific database system

The work presented here is done within the context of the datasource component of the Granite system, which handles array-based data. A datasource is conceptually an *n*-dimensional array containing a set of sample points. The array indices define the *index space*, also called a data volume. Each index space location has a collection of associated data values, called a *datum*. Datasources must handle two basic kinds of queries. A *datum query* specifies a single index space location, and is satisfied by the return of a single datum. A *subblock query* specifies an n-dimensional rectangular region of the index space, and is satisfied by the return of a data block, which is conceptually an n-dimensional array of datums. Since the spatial data distribution entails block structured subset extraction, the subblock query is what we use here to demonstrate the efficiency of our data distribution system in a cluster environment.

3.1. Storage ordering

A datasource is responsible for satisfying queries expressed in its n-dimensional index space by reading data from the file that is a one dimensional entity. It must therefore map its index space to file offsets, and does this with the help of an *axis ordering*. An axis ordering is simply a ranking of axes from outermost to innermost. The innermost axis of a storage ordering is known as the *rod axis*, where *rods* are series of elements that are contiguous in both the data volume and the one dimensional file. We call the axis ordering that maps the datasource to the file a *storage ordering.*

From I/O point of view, a rod represents a single sequential disk read and the rod length represents the size of that sequential read. So the number of rods and their length contained in a cache block are the keys to measure the I/O performance of a prefetching and caching technique that uses an *n*-dimensional cache block. For example, to iterate through the same data volume with same amount of cache memory, the algorithm that generates cache blocks with longer and fewer rods is better because it needs fewer disk reads and generates better scheduling.

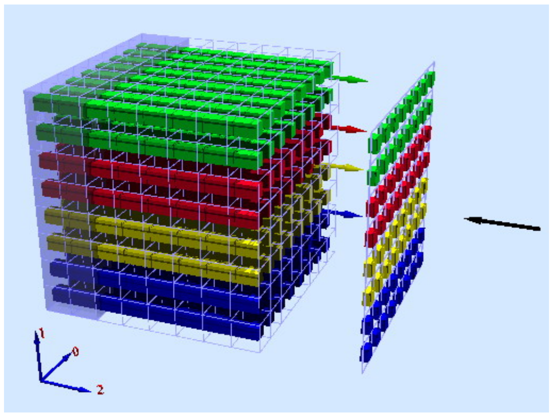
3.2. Access pattern and iterator

In the Granite system, the access pattern is represented as an iterator. The basic elements required to construct an iterator include a data source object, an iteration space, iteration block shape if doing block iteration, step in each dimension and the iteration axis ordering. An access pattern would not be complete if any of these is missing. In this case, the corresponding iterator can not be constructed or the access pattern can not be expressed by an iterator and conveyed to a datasource object. But, although the access pattern sent to datasource object has to be complete, user doesn't have to specify a definite one. An *indefinite access pattern* impose less restriction on the iterator construction and leave more chances for Granite to choose one that best match the data storage layout on disk. This *iteration ordering* determines the direction in which the iterator proceeds through the iteration space. The iterator has a value that changes with each invocation of the iterator's next() method. This value can denote either a single datum or in the case of a *block iterator*, an iso-aligned *n*-rectangular region.

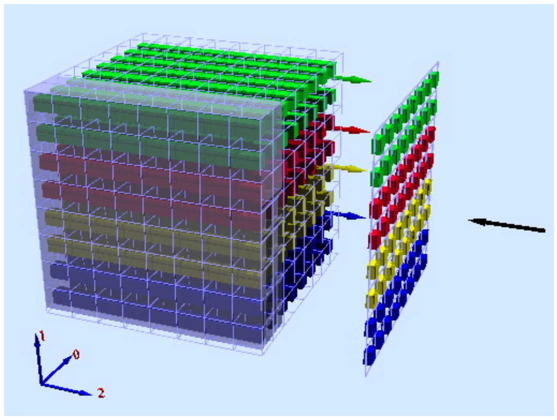
Block iterators are a natural fit for a data intensive parallel computing environment that needs a spatial job block distribution mechanism. Generally, the performance of iteration over data stored on disk depends upon the relationship between the storage ordering and the iteration ordering. It is desired that the iteration ordering matches storage ordering because it would generate the longest rods and need to do fewer disk reads while satisfying data requests. We elaborate on this point in section 4.

3.3. Spatial prefetching

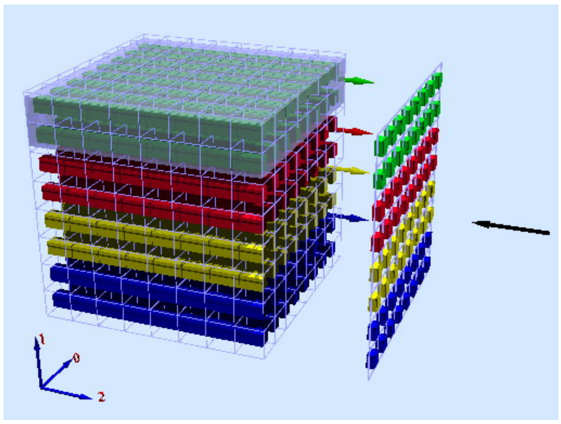
A significant component of our caching strategy is to shape the cache blocks to most effectively improve I/O performance. We call this approach *spatial prefetching (SP),* a form of *Iteration Aware Prefetching*.



(a)



(b)



(c)

Figure 1. As the image plane passes through the volume, consecutive slices of data are blended onto the image plane. Here, the plane has been divided among four compute nodes.

In previous work [9], we described an algorithm for shaping *n*-dimensional cache blocks for iso-aligned datum, block and plane iterations. An important property of those algorithms is that they produce cache blocks that are *well formed* with respect to the iteration. This means that once the iteration leaves a cache block, it never revisits it. Therefore no replacement policy is necessary, and often only a single cache block is needed. In [10], we demonstrated that the well formed property is particularly valuable for remote access because of the expense of reloading discarded blocks across a network.

3.4. Remote data access

The evolving high-performance Grid computing environment requires efficient remote data access and distribution over a long distance network. Improvements in disk access will not result in performance increases unless network latency is also addressed.

Collective I/O such as two-phase I/O and DDIO focuses on the reordering of requests for optimal disk scheduling to hide the disk latency. They don't pay much attention to remote data access which is very common in Grid computing environments where large multidimensional datasets are shared by different scientists. Instead, they rely on frequent network chats to do the intercommunication among the I/O nodes, and don't take into account the n-dimensional nature of spatial scientific data.

4. Application description - volume rendering using back-to-front blending

The application that we use to demonstrate our granite system in a cluster environment is a typical volume rendering application using back-to-front (BTF) alpha blending. Alpha blending is a convex combination of two colors allowing for transparency effects in computer graphics [12]. For example, in figure 1, given a user specified viewing direction relative to a transparent data volume, the blending will start with the back of the volume and get the color data slice by slice. Each time a new data slice comes in, the data is assigned a color using a *transfer function* and is blended with the previous slice pixel by pixel using the following RGB alpha blending functions shown in figure 2. When this process eventually reaches the front surface of the volume, the final blended result color will be displayed on the image plane.

$$R = A_s R_s + (1-A_s)R_b$$
$$G = A_s G_s + (1-A_s)G_b$$
$$B = A_s B_s + (1-A_s)B_b$$
$$A = A_s + (1-A_s)A_b$$

Figure 2. RGB Alpha blending functions

This process requires iterating through the whole volume. However, there are many options to get the same final result even with the same viewing direction. To effectively implement it in a computing cluster environment, we have to figure out a way to evenly divide the jobs among the compute nodes while being able to efficiently load the data into memory when the dataset is too large to fit in the collective memory of the compute nodes.

In the general case the viewing direction is arbitrary, but for simplicity we have limited it here to the three major axes, as shown in figure 1.

4.1. Finding min and max

The data values in a real dataset have arbitrary range, but must be mapped by the transfer function to RGB values between 0.0 and 1.0. This mapping requires us to know the range of data values before the blending process begins. We must therefore iterate through the dataset to find the minimum and maximum values. Fortunately this kind of preprocessing of the original dataset is only done once, and the result stored in a metadata file.

This preprocessing step is typical of an "embarrassingly parallel" application in which the job distribution mode does not matter at all. In particular, the result will be correct regardless of the order in which we iterate through the dataset, so the best strategy is just to choose the iteration ordering that matches the storage ordering and use available memory to build a cache block with the longest rods possible. Since the application does not specify a particular access pattern, we use the storage ordering to determine how we proceed through the data.

4.2. Cache block shape and splitting

Given a user specified viewing direction, the Granite system must perform two tasks in order to maximize the performance of the cluster. First, it must choose the shape of the *n*-dimensional cache block that the head node uses to access the disk or remote server. Second, this cache block must be *split* into *n*-dimensional *job blocks* which are then distributed to the cluster compute nodes.

The shape of the cache block on the head node will determine the number of times disk and network latency costs are paid. To minimize disk latency penalties, we choose a cache block shape that maximizes the length of the rods comprising the cache block. Figure 1 shows the three possible cache block shapes for an example data volume. Clearly, the cache block shown in figure 1a has much shorter rods than in the other two cases, so performance would suffer due to a larger number of disk latency penalties. Since the cache block rod lengths are the same in figures 1b and 1c, choosing between them must be done by considering the number of cache blocks that must be read to process the entire data volume. If the data is stored remotely, each cache block retrieved from the server will incur network latency costs, so we should choose the cache block shape that minimizes the number of network transactions. Of course, in some cases, both disk and network latency costs will be equal for various shapes, and any shape can be chosen.

After the cache block shape has been chosen, we must decide how to split the block into the job blocks that are distributed to the cluster compute nodes. It is important that each section of the image plane be the sole responsibility of a single compute node in order to avoid dependencies and resource contention. For this reason, we never split the cache block along the view axis. Currently, we simply divide the cache block into n job blocks along one of the remaining axes, where n is the number of compute nodes. Since our current application does not require compute nodes to communicate with each other, we have found that choice of axis along which the data is split makes little difference to performance. However, in future we plan to consider the more complex case where compute nodes are not independent and the shape of the job blocks determines the number and size of the communications between them.

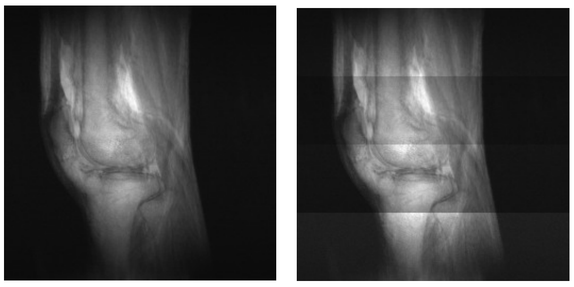

(a) (b)

Figure 3. 3dknee images generated using our data distribution system on Orion cluster. The knee data is courtesy of Siemens Medical Systems, Inc.

4.3. Data processing

In our implementation, the data processing includes two steps, *slicing* and *blending*. After the job block is transferred to a compute node, it is still *n*-dimensional. Slicing is actually a block-iteration along the viewing axis of the in-memory data block itself. Since the slice is of same shape, so blending can be easily done.

4.4. Image combination

As described at the end of section 4.2, each compute node is responsible for its own section of the image plane. In order to form a complete image, these disjoint sections must be combined to form a single result image.

Figure 3 shows two 3dknee images generated using our data distribution system on a 12-node Orion cluster. Image 3*b* is intentionally left as seemingly partitioned to demonstrate the effect of job splitting and result image combination. Image *3a* is the actual generated image.

5. Iteration Aware data distribution prototype

The above application is based upon our iteration aware data distribution prototype.

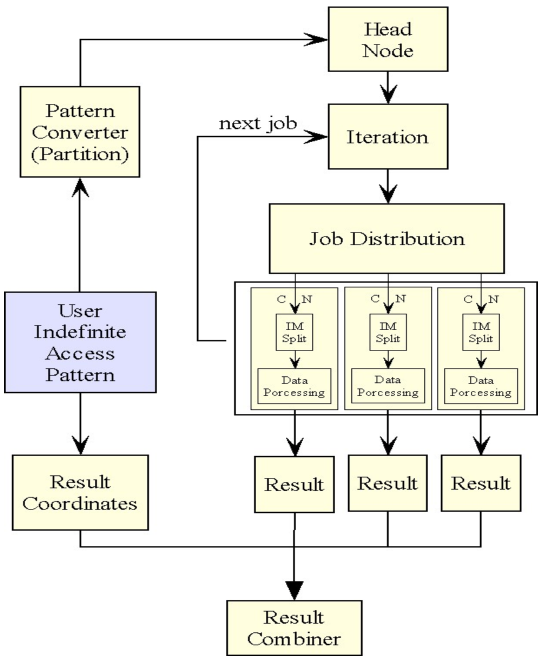


Figure 4. Framework of Granite IADDP – Iteration Aware Data Distribution Prototype

5.1. Pattern converter

When the user does not know or does not care about the exact pattern data is accessed, the user can specify as little constraint as they want on the access pattern. All other job is done automatically by what we call pattern converter. A pattern converter is a facility that accepts an indefinite access pattern and generates a complete one by putting more constraints on it. The new constraints would make the final access pattern one that can create the best cache shape mode and thus bring best I/O performance.

In cluster environment, the pattern converter takes the responsibility of integrating job distribution into iterator construction. After that, the iteraor not only contains the description of the iteration, but also contains the job distribution pattern. Armed with this enhanced complete iterator, the datasource object would not only extract the right data but extract it most efficiently and distribute them evenly.

At the same time, it also generates a coordinate system that is used to combine the results later.

5.2. Optimizing scheduling of job distribution

Our data distributor is responsible for satisfying the data requests from compute nodes as requested. But the processing speeds of different nodes might be different though not of a quite high probability in cluster environment. Since our system does not support parallel device yet, the datasource object relies on an iterator calling next() to specify the job block bounds and then issue jobs to compute nodes. Since this call can only be made one at a time, latter nodes will have to wait until all other front nodes are satisfied. To avoid this non-frequent but existing waiting, we use a data pool to hold the extracted data for each compute node even the data requests are not received yet. The earliest request for each round of data distribution will spark the data assignment and extraction from disk if not in memory. We call this data pre-allocation. This is an optimization with an assumption that the compute nodes are not so heterogeneous in terms of their speed, because that would cause much memory overhead. That can speed up the prefetching and increase the overlapping between disk I/O and data processing with a reasonable amount of memory overhead in cluster environment. Actually, it might be better to parameterize it whether using this or not.

5.3. In memory job block splitting and data processing

Once each compute node gets a job block, there is a in-memory data block splitter which is actually a worker datasource object that takes the new job block, updates its content and generates further application specific data iteration and distribution in arbitrary ways. So, this two-level datasource design enables the reconciliation between the iteration ordering that is suitable for disk data extraction and the iteration ordering suitable for application when they conflict. Since the data is already in the high speed main memory, different data splitting methods for in memory data block do not show much performance difference anymore, as confirmed in the results. So, the main performance gain using spatial prefetching is on hiding disk latency.

Generally speaking, low level memory hierarchy is the bottleneck of whole I/O system. The more expensive data extraction is delayed to later high level memory hierarchy, the more performance increase we will get. Our goal is to transform the more detailed data splitting, which otherwise means more operations on high latency device, to large efficient I/O from secondary storage and network. This has to be realized by knowing the exact access pattern beforehand and supporting collective I/O operations. So, we believe our spatial prefetching and caching apply to the whole storage hierarchy.

5.4. Result combiner

As explained in section 4.2, our current implementation does not yet support interdependent processing. Each compute node is responsible for producing results without communicating with other nodes. Unless the user requires them be stored separately, all the partial results will be combined into one single result.

When required to combine partial results, the Granite system will maintain the relative position of the results using a coordinate system to ensure proper assembly of the final result. The final whole result could have a 2D plane shape as in our sample application, or a 3D volume as with a volume filtering or feature detection program.

When the user wants to keep the partial results stored separately, our system can generate the corresponding metadata so that the final result can be constructed at a later time.

Such a facility is of particular interest with extremely large datasets where the results of processing are inconveniently large. For example, researchers in a Grid computing environment might prefer to locate and retrieve selected subsets of the result of a computation, rather than have to download the entire volume of results. Such functionality requires metadata support for n-dimensional subsets of data volumes, an area that we are currently investigating [7].

Table 1. Local cluster data distribution performance results using Granite spatial prefetching and caching. All results are in seconds.

View Axis	256MB SP Cache					
	Case A		Case B		Case C	
	H	V	H	V	H	V
0	65	63	164	152	84	79
1	83	81	163	153	65	62
2	161	163	88	88	72	72

Table 2. Local cluster data distribution performance results without using Granite spatial prefetching. All results are in seconds.

View Axis	File System Cache Only					
	Case A		Case B		Case C	
	H	V	H	V	H	V
0	82	78	988	227	93	198
1	94	72	999	149	82	60
2	234	152	190	80	85	67

6. Experimental results

In order to verify that Granite makes the correct choices when choosing the cache and job block shapes, we ran tests for a variety of different configurations. The tests were performed on a 12-node Orion cluster with a Seagate Barracuda drive with 11.5ms average seek time. The number of working compute nodes is adjustable. Remote tests transferred data between a client at the University of Mississippi and a server at the University of New Hampshire. The server is a single processor Pentium 4 machine with a 2.4Ghz CPU and 2GB of RAM running the Linux operating system. The server has a 7200 RPM disk with a 9.3ms average read seek time.

6.1. Local test results

In all result tables, cases A, B and C correspond to the different cache block shapes as shown in figure 1. H and V denote horizontal and vertical splitting for the same cache block shape.

Tables 1 and 2 show the times for a 512x512x1024 subset volume rendering of a 1024x1024x1024 4GB float data volume using 256MB SP Cache and 8 compute nodes. These results confirm that by using spatial prefetching and caching, our data distribution system outperforms file system caching alone when dealing with large n-dimensional datasets.

The exception comes when the job splitting does not break the original long rods otherwise formed by our spatial caching. In this case, spatial prefetching offers no advantage, and incurs some additional overhead. In the other cases, however, the benefit of collective I/O and spatial prefetching in our distribution system beats file system caching. We are speeding up the rate of data transfer from head node to compute nodes by minimizing the number of read operations made to disk. This optimization is made possible by maintaining an *n*-dimensional view of the data throughout the system.

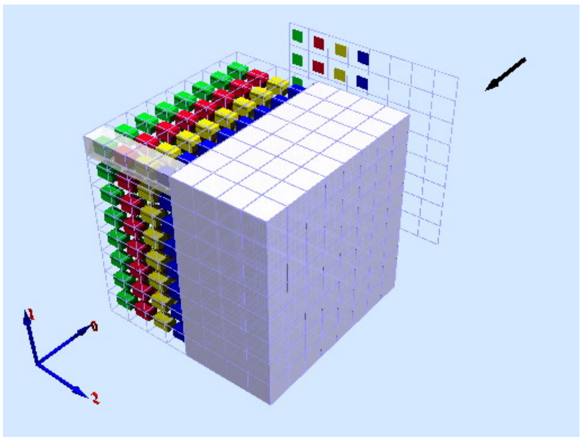


Figure 5. Case 0BH. Without spatial prefetching and caching, total disk reads increase 4 times since the rod is split into 4 separate disk reads.

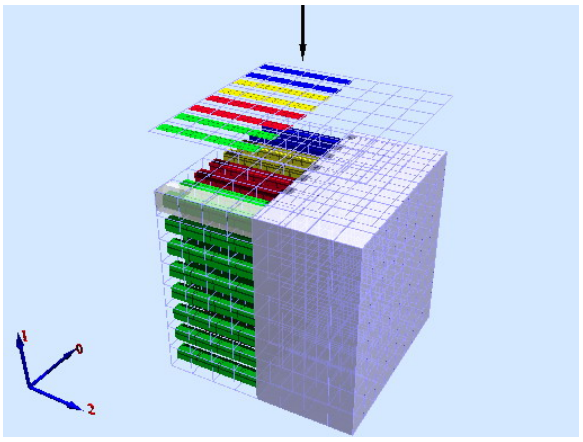


Figure 6. Case 1CH. Without spatial prefetching and caching, the total disk reads do not change since the rod is not split.

It is also apparent that the spatial prefetching results show little sensitivity to the splitting mode. By using collective spatial prefetching and caching, the cache block is now the logical unit of I/O operation. So, the overall I/O performance is determined mostly by how fast the collective cache block can be loaded into memory and how many cache blocks are required for the whole data volume. Since the job splitting is done in memory after the data is extracted from disk or received from network, the total I/O time is not affected by splitting modes very much. A specific example is shown in the comparison of the two cases in figure 5 and figure 6, which correspond respectively to case 0BH and 1CH respectively in the result table.

In the data distribution without spatial prefetching, job blocks read directly from disk result in a large number of small read requests. If the rod length is much shorter than before, as shown in case 0BH, the performance would also drop strikingly (e.g. a factor of six) compared with spatial prefetching.

In other cases the magnitude of performance gains due to spatial prefetching are not quite so big. This is because the job blocks given to the compute nodes are fairly large. So, even without spatial prefetching, the read requests made to disk are relatively long, and have good locality. In such cases, the marginal benefit of spatial prefetching is lower, but still noticeable. Performance gains are determined largely by the ratio between length of the rods used to fill the cache block on the head node and the rods used to fill job blocks read without the benefit of spatial prefetching. This ratio can be improved either by allocating more memory to the spatial prefetching cache on the head node, or by reducing the size of job blocks, perhaps due to the addition of more compute nodes. In this second case, benefits should be even more apparent because the additional computational power will make I/O costs a larger part of the total execution time.

6.2. Remote test results

The dramatic performance gains in our iteration aware remote data access encourage us to integrate our data distribution system with remote data access. This is very important direction because Grid computing is more and more popular since it advocates resource sharing. Efficient data access within Grid environment will improve the exploitation of the whole Grid's computing capacity.

Tables 3 and 4 on next page show the times for a 512x512x512 subset volume rendering of 518x518x518 556MB float data volume using 128MB SP Cache and 8 compute nodes.

As seen from these two tables, using Granite collective spatial prefetching and caching for remote data access is also very effective for hiding network latency and increasing the utilization of network bandwidth. As in local tests, greater performance gains are achieved when the collective prefetching forms a cache block that would otherwise be broken into many small data requests requiring small separate network transfers. Granite realizes this by creating a multidimensional cache block that groups multiple separate data requests into a small number of large requests and thus reduces the number of payment of network latency penalties.

Without using our spatial prefetching and caching, the data distribution is realized by sending separate request for each job, greatly increasing the number of network transactions. This greatly slows down the whole data transfer rate and the speed to feed data to compute nodes.

Table 3. Cluster data distribution performance results using Granite spatial prefetching and caching to cache data on client side for remote data access. All results are in seconds.

View Axis	128MB Client Side Cache					
	Case A		Case B		Case C	
	H	V	H	V	H	V
0	387	369	392	386	392	370
1	404	381	397	395	381	373
2	401	393	421	382	436	385

Table 4. Cluster data distribution performance results without using Granite spatial prefetching and caching for remote data access. All results are in seconds.

View Axis	No Client Side Cache					
	Case A		Case B		Case C	
	H	V	H	V	H	V
0	529	492	660	518	523	481
1	524	480	668	517	541	521
2	509	498	484	485	506	488

7. Conclusion and future work

We present an iteration aware data distribution system for computing cluster environment. This system takes advantage of full prior knowledge of future access pattern and can greatly reduce disk and network latency that hinders the I/O performance to increase the speed of feeding data to compute nodes. Our system encompasses the advantages of collective I/O, informed prefetching without the need for data storage optimization.

In the near future, we are going to explore the data distribution within environments that enforce data interdependency and thus the communication among compute nodes. Also, we need to put more efforts into the specification of application iteration constraint, and establish a standard model for our pattern converter based on the specification.

8. References

[1] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, K. Yelick, "Cluster I/O with River: Making the Fast Case Common", *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, May 1999

[2] G. Gibson, J. Vitter and J. Wilkes, "Strategic directions in storage I/O issues in large-scale computing", *ACM Computing Surveys*, Volume 28 , Issue 4 , December 1996

[3] J. Huber, C. Elford, D. Reed, A. Chien and D. Blumenthal, "PPFS: A high performance portable parallel file system", *Conference proceedings of the 1995 International Conference on Super-computing*, pages 385-394, ACM Press, July 1995

[4]D. Kotz and C. Ellis, "Practical Prefetching Techniques for Parallel File Systems", *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, 1991

[5] T. Madhyastha, G. Gibson, and C. Faloutsos, "Informed Prefetching of Collective Input/Output Requests", *Proceedings of SC99: High Performance Networking and Computing*, 1999

[6] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, M. Best, "File-Access Characteristics of Parallel Scientific Workloads", *IEEE Transactions on Parallel and Distributed Systems*, Volume 7 Issue 10, October, 1996

[7] S. Ramakrishnan and P. Rhodes, "Multidimensional Replica Selection in the Data Grid", *Proc. 15th IEEE International Symposium on High Performance Distributed Computing*, pp. 373-374, 2006

[8] J. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy", *IPPS '93 Workshop on Input/Output in Parallel Computer Systems,* pp 56-70, 1993

[9] P. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr, "Iteration Aware Prefetching for Large Multidimensional Scientific Datasets", *Proc. SSDBM '05*

[10] P. Rhodes and S. Ramakrishnan, "Iteration Aware Prefetching for Remote Data Access", *Proc. 1st International IEEE Conference on e-Science and Grid Technologies (e-Science05)*

[11] S. Sarawagi, M. Stonebraker, "Efficient Organizations of Large Multidimensional Arrays", *Proc. of the Tenth International Conference on Data Engineering*, Feb. 1994

[12] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*, Kitware, 2002

[13] E. Smirni, R. Aydt, A. Chien, and D.Reed, "I/O Requirements of Scientific Applications: An Evolutionary View", *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pp 49-59, August 1996

[14] E. Smirni, and D. Reed. "Workload characterization of input/output intensive parallel applications", *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pp 49-59, August 1996

[15] R. Thakur, W. Gropp, and E. Lusk. "Optimizing noncontiguous accesses in MPI-IO", *Parallel Computing*, 28(1):83-105, Jan. 2002