DPFS: A Distributed Parallel File System

Xiaohui Shen and Alok Choudhary
Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Northwestern University, Evanston, IL 60208
{xhshen,choudhar}@ece.nwu.edu

Abstract

One of challenges brought by large-scale scientific applications is how to avoid remote storage access by collectively using enough local storage resources to hold huge amount of data generated by the simulation while providing high performance I/O. DPFS, a Distributed Parallel File System, is designed and implemented to address this problem. DPFS collects locally distributed unused storage resources as a supplement to the internal storage of parallel computing systems to satisfy the storage capacity requirement of large-scale applications. In addition, like parallel file systems, DPFS provides striping mechanisms that divides a file into small pieces and distributes them across multiple storage devices for parallel data access. The unique feature of DPFS is that it provides three file levels with each file level corresponding to a file striping method. In addition to the traditional linear *striping method, DPFS also provides a novel* Multidimensional striping *method that can solve performance problems of* linear striping *for many popular access patterns. Other issues such as load-balanceing and user interface are also addressed in DPFS.*

1 Introduction

Data intensive applications have presented challenging problems to current parallel computing systems especially I/O sub-systems. One of major problems is the storage capacity. The tremendous increase of data volume of modern scientific applications has significantly out-paced the increase of native storage devices (secondary storage system) of parallel computing systems. Employing external storage resources is a promising solution. A major problem, however, of accessing these remote storage resources is performance. The data has to move a long distance over networks. For example, the HPSS and disk storage resources available for us are located at San Diego Supercomputer Center (SDSC), while our computing systems are located at Argonne National Laboratory and Northwestern University. To address the problems of storage capacity and performance brought by large-scale data intensive applications, an I/O sub-system should be able to (1) employ external storage resources in a locally distributed environment that are easily available to relieve contention at internal native storage of parallel computing systems. Although purchasing new storage could be a solution, it is neither economical nor convenient; (2) employ high performance I/O techniques such as parallel I/O to address performance problems.

Based on the above observations, we are motivated to design and implement a Distributed Parallel File System (DPFS). The features of DPFS are highlighted as follows.

- **Distributed** DPFS is distributed because it collects distributed storage resources from networks. The chance to aggregate sufficient storage volume for large-scale data intensive applications exists since in a typical computing environment, there are many local storage resources not being fully used.
- **Parallel** DPFS adopts parallel I/O techniques to achieve high performance. By striping the file across multiple storage devices, parallel processes can access their portion of data from different storage devices simultaneously. In addition to the general striping method found in many parallel I/O systems, DPFS also proposes novel striping methods such as *multidimensional striping* and *array striping* that can take hints from users to help better place and organize data on storage.
- **File System** DPFS is designed and implemented as a general file system. It provides an Application Programming Interface (API) to help users easily to store and access data over distributed storage devices and provide storage location transparency as well.
- **Database** A significant feature of DPFS that distinguishes it from other parallel file systems is that it uses databases to store meta-data of file system. Database makes the meta-data management easily and reliably in a distributed environment.

Fundamentally, DPFS tries to combine the advantages of Distributed File System (DFS) and Parallel File System

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 IEEE

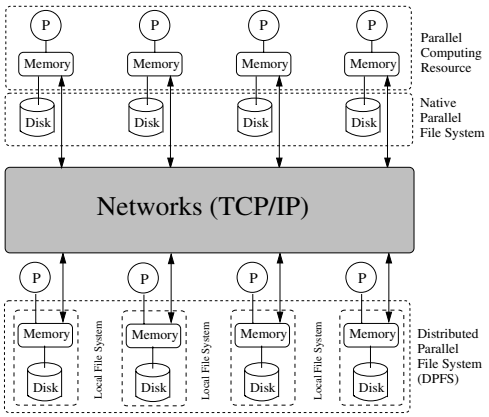

Figure 1. **System Architecture of DPFS.**

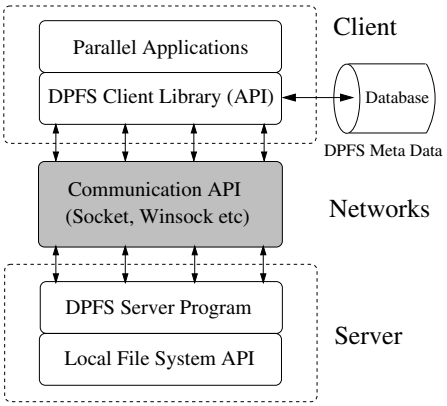

Figure 2. **Architecture of DPFS from Software's Point of View.**

(PFS) with the aid of databases to satisfy both storage capacity and performance requirements of large-scale parallel applications.

The reminder of paper is organized as follows. In Section 2 we describes the overall architecture of DPFS. In Section 3 we present three striping methods employed by DPFS. In Section 4 we present the striping algorithms and an optimization called request combination is also described. In Section 5 we introduce the meta data management of DPFS and database tables. In Section 6 we present DPFS API. We describes user interface of DPFS in Section 7. In Section 8 we show the performance numbers and In Section 9 we introduce the related work. We conclude the paper in Section 10.

2 Architecture of DPFS

DPFS adopts Client-Server architecture: the client (compute node) sends requests to the server (I/O node) whenever it needs to perform input or output. The server which resides on a specific storage device is responsible for sending the requested data to the client or storing data from the client on local storage. The parallelism can be found when multiple servers work in parallel to service clients' requests. The concurrency exists on each server since multiple requests from clients to the server can be serviced concurrently by server's spawning multiple processes or threads to handle them. The parallelism and concurrency of servers are the key contributions to high performance in DPFS. Figure 1 shows the overall system architecture of DPFS. This architecture can be grouped into several layers.

- **Parallel Computing Resource** At the top is the parallel computing resource, which could be distributed memory systems such as IBM SP2, Network Of Workstations (NOW) and shared memory systems such as SGI Origin 2000.
- **Native Parallel File System** Under the computing resource is the storage sub-system for parallel systems. This layer makes use of the local disk storage associated with the computing resource and form a native parallel file system to achieve high performance. The problem of this storage resource, however, is that it is tightly coupled with the computing resource, therefore, it is very hard to scale with the increasing storage capacity requirement of data intensive applications.
- **DPFS** At the bottom is our proposed Distributed Parallel File System (DPFS). DPFS utilizes unused storage resources distributed over networks. These resources can be found on various commodity workstations and personal computers. For example, at our ECE department of Northwestern University, it is very easy for a typical user to access tens of workstations. Since the main storage space of a user is located at a Networked File System (NFS), much of the local disk space of these machines is unused. Aggregating these disjointed storage by DPFS, we can have a very large storage space to satisfy large-scale data intensive applications' storage requirement[1].
- **Networks** The network makes accessing various distributed storage resources conveniently and it is also very easy to find enough storage space on network to satisfy storage space requirement of large-scale applications.

Figure 2 shows the layered architecture of DPFS from the software's point of view. At the top is the parallel ap-

[1] DPFS is built on top of the local file system of each storage resource, therefore, there is no need to change the underlying system software of local file system and DPFS can take advantage of I/O optimizations such as caching and prefetching of the local file system to access the actual data on disks.

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 IEEE

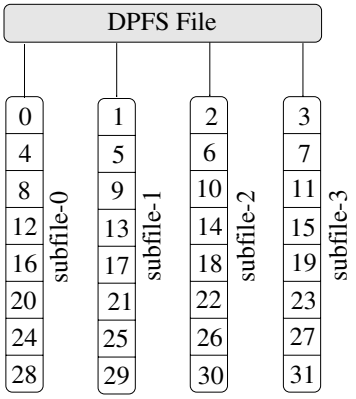

Figure 3. **DPFS File View.**

plications. The user uses DPFS API whenever she needs to perform I/O. DPFS API then calculates the *brick* (basic striping unit in DPFS) numbers according to the portion of data held by each processor. Next, DPFS-API consults database to obtain meta data information of the file such as which I/O nodes store these bricks and what their offsets are in the local *subfile*. Then, DPFS API invokes system communication API such as socket on UNIX [20] to send the request to the server which runs on the local file system of storage devices. As long as the server receives the request, it uses the local file system API to actually perform I/O.

3 DPFS Striping Methods and File Levels

The most significant feature of parallel file systems is that a parallel file is striped across multiple storage devices with each storage device holding a portion of data. The key of DPFS, like other parallel file systems, lies in the striping methods. A striping method decides the shape and size of a striping unit which is the basic accessing unit and building block of a DPFS file. A basic striping unit of DPFS is also called a *brick*. A DPFS file consists of a sequence of bricks numbering from zero to $filesize/bricksize$. Given multiple storage devices, a brick is assigned to a storage device according to the *striping algorithm* such as round-robin when a DPFS file is created. A storage device may be assigned multiple bricks and these bricks form a subfile in the local file system. Figure 3 shows a DPFS file striped across four I/O devices by round-robin algorithm.

In most parallel file systems, they assume a single type of striping method. The problem of this fixed striping method is that for many access patterns, it may cause very poor performance. The striping method in DPFS, however, is flexible because DPFS has multiple methods can be chosen. The DPFS-API provides a hint structure that can help choose a suitable striping method.

In this section, we present three striping methods in

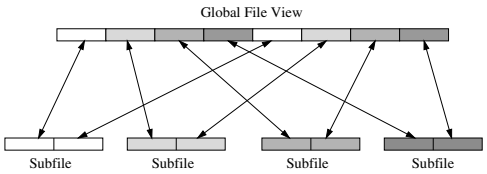

Figure 4. **Linear Level of DPFS File.**

0	0	1	2	3	4	5	6	7	1
2	8	9	10	11	12	13	14	15	3
4	16	17	18	19	20	21	22	23	5
6	24	25	26	27	28	29	30	31	7
8	32	33	34	35	36	37	38	39	9
10	40	41	42	43	44	45	46	47	11
12	48	49	50	51	52	53	54	55	13
14	56	57	58	59	60	61	62	63	15

Figure 5. **Linear Striping Problem. Brick numbers are marked at the left and right sides. Bricks with the same shading belong to the same subfile.**

DPFS, each method corresponding to a DPFS *file level*. When a DPFS file is created, the user can convey a *file level* hint through DPFS API, then the file will be striped using corresponding striping method.

3.1 Linear Striping

The sequential UNIX file is a stream of contiguous bytes. In many parallel file systems, although files are striped across multiple storage devices, they are still treated as linear logically. The reasons are that linear model is consistent with the sequential file and many parallel files still need to be transfered to sequential workstations for post-processing such as data analysis and visualization. In the linear striping method, a striping unit, called linear brick, is also a linear sequence of bytes. The file is called linear file if it consists of linear bricks. Figure 4 illustrates the linear brick and linear file. No extra effort is needed when a linear parallel file is transferred to a sequential file system, but for many popular access patterns, linear striping could cause serious performance problems.

3.2 Multidimensional Striping

The problem of linear striping in the above sub-section can be illustrated in Figure 5. Consider a 8×8 two dimensional array and suppose a brick size is of 4 elements. In the linear striping, the two dimensional array is flattened to

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 IEEE

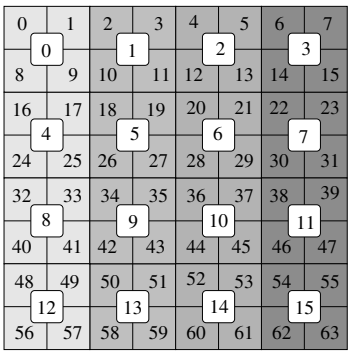

Figure 6. **Multidimensional Striping Method. Each striping unit is multidimensional (2×2). Brick numbers are marked at the center of each brick and bricks with the same shading belong to the same subfile.**

a linear sequence of elements as in Figure 4 and the linear brick is a contiguous 4 elements. Suppose the array is striped across 4 I/O devices and the brick numbers are marked at the left and right sides of the array in Figure 5. Brick 0 contains array elements 0, 1, 2 and 3, and brick 1 contains 4, 5, 6 and 7 and so forth. The array is distributed by round-robin algorithm, so the bricks with the same shading forms a subfile in the local file system. Now suppose there are four processors accessing the array with each processor accessing a chunk of array (BLOCK, *), then each processor will access exactly two rows and 4 bricks are accessed. But what if the processors access in a (*, BLOCK) way ? That means each processor accesses two columns, so 8 bricks are needed and only part of each brick (2 elements) are useful for each processor. For example, processor 0 will access the first two columns, so it has to access brick 0, 2, 4, 6, 8, 10 , 12 and 14, and only the first two elements of each brick are really useful, the second half will be discarded. Consider a real-world array: a $64K \times 64K$ two dimensional array. Suppose the brick size is 64K, so each row forms a brick and there are totally 64K bricks for linear striping. When a processor accesses a column of data, all the bricks (64K=65536) will be needed. Accessing columns of data is very common for many applications such as matrix multiplication, but the linear striping results in too many brick requests from client to server and only a small portion of the brick is useful. A deep study shows that this problem is rooted in the nature of linear file model, in which a file is treated as a sequence of linear bytes. For many scientific applications, the multidimensional array is the basic operating unit from the user's point of view. However, when linear striping is applied, the multidimensional array has to be flattened to a linear file or one dimensional array (Figure 4) in order to perform the striping. Doing this, however,

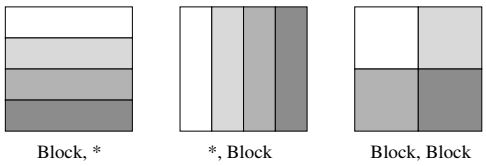

Figure 7. **Array Level of DPFS File. The striping unit is large *chunk*.**

makes convenience of high level array manipulation lost at the low level of file system. For example, the convenient way to express a column of data in a two dimensional array can not be easily manipulated at file system level. To maintain the original multidimensional information and the manipulation flexibility of the array, we propose a novel striping method called *Multidimensional Striping* to address this problem. The striping size of each brick does not change, but each striping unit (brick) is multidimensional. The file is called multidimensional file if it consists of multidimensional bricks. Figure 6 shows a 8×8 array striped by 2×2 multidimensional bricks. When the processor 0 accesses the first two columns again, it only needs to access 4 bricks (0, 4, 8 and 12) and no extra data is accessed. For the $64K \times 64K$ array example, each brick size would be 256×256, so only 256 bricks are needed. When a multidimensional file is converted to a sequential file, extra in-memory data reorganization is needed. Compared to expensive I/O access, in-memory cost is very small.

3.3 Array Striping

Many scientific applications' access pattern to an array can be described in a High Performance Fortran (HPF) notation, such as (Block, *), (Block, Block) and so on. Figure 7 shows a two dimensional array accessed by 4 processors. Each processor writes and then reads a coarse-grain *chunk* of data. In this case, storing each chunk as an integral unit is more efficient than further dividing it into smaller fine-grain bricks because each chunk will be accessed as a whole unit, striping into bricks only cause extra work and is not necessary. One example to demonstrate this scenario is that many large-scale scientific applications periodically dump check-pointing data. Each processor writes the data it holds to storage and simply reads it back later when the application resumes from this point. In this case, striping data on each processor into smaller bricks results in a large number of small data requests which is totally unnecessary. To address this problem, DPFS provides its third striping method: *array striping*. The striping unit, also called array brick, is a large coarse-grain chunk (Figure 7). The data distribution of the array observes HPF convention which fits many scientific applications. The file is called array file if it consists

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 IEEE

```
B = num of bricks;
S = num of servers;
initialize p[j], j = 0 to S;
A[j] = 0, j = 0 to S;
for i=0 to B {
   find k, where A[k]+P[k] <=
     A[j]+P[j] for all j = 0 to S;
   assign brick i to server k;
   A[k] = A[k] + P[k];
}
```

Figure 8. **Greedy Algorithm.**

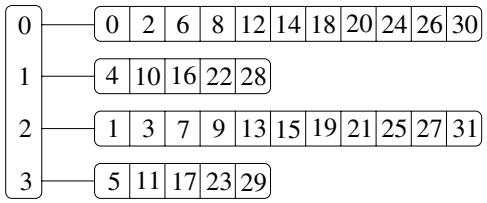

Figure 9. **Greedy Algorithm Example.**

of array bricks.

The three levels of DPFS file bring more flexibility and convenience as far as performance is concerned. The low level such as linear level is more general than others but may cause performance problems for some access patterns. On the other hand, the high level such as array level can bring significant performance improvement but its domain is limited in the context of HPF notation, although many applications fall into this category. To achieve both generality and high performance, the file system itself is lack of enough information. Therefore, the user's involvement is crucial because only the user has the best picture of how her data will be utilized. Section 6 will present a hint structure in DPFS API to make user's involvement easily.

4 Striping Algorithms and Request Combination

4.1 Greedy Striping Algorithm

Given a brick, the striping algorithm of DPFS decides which storage device should be assigned to take the brick when the file is created. A straightforward algorithm is round-robin, in which each storage device is assigned a brick alternately. Figure 3 shows a DPFS file which has 32 bricks, striped across four storage devices by round-robin algorithm. The round-robin algorithm expects the striping units being evenly distributed across storage devices, thus trying to keep a balanced workload on storage. In the distributed environment like DPFS, however, round-robin may not yield a 'fair' workload distribution. For example, our DPFS may consist of heterogeneous storage resources that are distributed over networks with different communication bandwidth. Accessing a brick on fast storage and high-bandwidth network could be much faster than accessing a brick on slow storage and low-bandwidth network. Therefore, evenly distributing bricks across storage devices may actually cause load un-balance. The basic idea to solve this problem is to let the fast storage be assigned more bricks than slow storage. We designed a Greedy Striping Algorithm to address this problem. We assign each storage device a normalized performance number according to their access time for one brick. The value for the fastest storage is 1, and a integer number larger than 1 for others. The algorithm is depicted in Figure 8. The idea of the algorithm is that given a brick i, it will be assigned to the storage device that will minimize the maximum value of the sum of performance numbers of each storage among all storage devices. Figure 9 shows brick distribution by our greedy algorithm for the file in Figure 3.

4.2 Request Combination

Another issue brings to our attention is that a processor's data request may consists of multiple bricks. For example, consider four processors accessing a DPFS file which has 32 bricks (Figure 3). Suppose processor 0 accesses brick 0 to 7 and processor 1 accesses 8 to 15, and so on. In a general approach, each processor issues a request for a brick and continues until all the bricks are accessed. The problem of this approach is two-fold. First of all, to access each brick, a request from client is needed. So there are many requests sent to servers. The servers have to spawn a process or thread to handle each request for only one brick. This could make a server too busy to handle all the requests if many requests happen to be sent to it simultaneously. The un-handled requests have to try again later. Second, consider Figure 3 again. In the general approach, when all processors start to access data, processor 0, 1, 2 and 3 will access brick 0, 8, 16 and 24 respectively. Note that brick 0, 8, 16 and 24 are on the same storage device. Then for the next brick, they access brick 1, 9, 17 and 25 respectively. These bricks are still on the same storage. This problem would happen until all the bricks are accessed. Although our DPFS server can service concurrent requests, the actual I/O has to be sequentialized locally due to the nature of sequential storage device. Therefore, the potential parallelism provided by file striping is not been fully exploited.

Based on the above observations, we propose a request combination scheme to address this problem. Let's take processor 0 as an example. Processor 0 accesses totally 8 bricks (0 - 7) which are striped across over four devices with 2 bricks on each device. The combined approach will let processor 0 access brick 0 and 4 in one request because they

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 IEEE

reside on the same storage. Next, it accesses brick 1 and 5 in another single request, and so on. Therefore, there are only 4 requests needed for each processor, much smaller than 8 requests of general approach. The less requests, the less network traffic and server processing. In addition, we can better schedule the request sequence of each processor after requests are combined. For example, we can easily let processor 0 starts its access from subfile-0 (brick 0, 4), while processor 1 starts from subfile-1 (brick 9, 13), processor 2 from subfile-2 (brick 18, 22) and processor 3 from subfile-3 (brick 27, 31). As these combined bricks are located on the different physical storage devices, the maximum parallelism can be exploited by avoiding multiple processors congesting requests at one storage device.

5 Meta Data and Database

We have chosen a database as repository for DPFS meta data. Using a database solution has many advantages. It can save programming efforts since SQL is a very high level and reliable interface compared to manipulating low level file directly. Moreover, the transaction mechanism provided by database systems can help maintain meta data consistency easily, especially in a distributed environment.

The DPFS meta data should keep such information as what servers are available for I/O, how the data bricks are distributed across servers and what DPFS directories and files are currently maintained by DPFS and so on. We use four database tables to maintain these information and they are described as follows.

- **DPFS-SERVER** This table has three attributes: *server-name* which stores all the server names available for clients; *capacity* which tells the user how much storage space available on this server; and *performance* which stores the normalized performance number of servicing client's request by that server. Our greedy algorithm in Section 3 use this attribute to make a balanced brick distribution.

- **DPFS-FILE-DISTRIBUTION** This table maintains information about how DPFS bricks are distributed across different servers. It has three attributes: *server* which stores the server name; *filename* which keeps the subfile name of DPFS distributed on that local file system. Usually, we use the same name as the DPFS name. This subfile name also includes DPFS path; and *bricklist* which stores a list of bricks that is maintained by this server. These bricks form a file from local file system's point of view or subfile from DPFS's point of view.

- **DPFS-DIRECTORY** This table keeps the DPFS file and directory tree structure. There are three attributes: *main-dir* which is the name of directory name in DPFS; *sub-dir* stores the sub-directories under *main-dir*; *files* stores the DPFS file names under *main-dir*. When a new file is created, attribute *files* is updated to include the new file. When a new directory is created, the row containing the current directory will be updated to include the created sub-directory and a new row, with the created directory name as the *main-dir*, is inserted into the table.

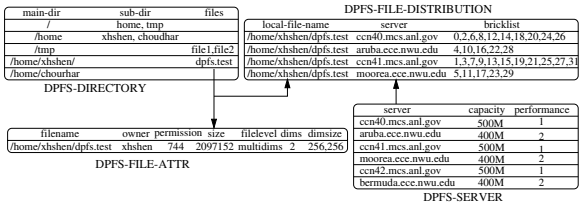

Figure 10. **DPFS Meta Data Tables.**

- **DPFS-FILE-ATTR** This table maintains such information as owner of the file, access permissions, file size etc. As DPFS has three levels, this table also has an attribute *filelevel* to reflect the characteristics of file striping. The shape of striping unit is described by the attributes *stripe-dims* and *stripe-size*. For the array level file, attribute *pattern* decides how the array is chunked by HPF notation.

The relationships of these tables are shown in Figure 10.

6 Application Programming Interface

DPFS API supports both contiguous and non-contiguous data access. DPFS adopts MPI-I/O's derived data type [23] approach to allow the user to express non-contiguous data conveniently. The main functions of DPFS APIs are described as follows.

- **DPFS-Open()** The main arguments include a pointer to DPFS file handle, file name, access mode (read or write) and the suggested number of I/O nodes by the user (for write operation only). This routine opens a DPFS file and then returns a pointer to the DPFS file handle for later usage.

- **DPFS-Write()** The main arguments include an opened DPFS file handle, a buffer holding the data to be written, the derived data type to express non-contiguous data and a *hint* structure that allows the user to select a suitable file level etc.

- **DPFS-Read()** The main arguments are similar to the DPFS-Write() except that the buffer is to receive data

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 **IEEE**

and some hint information in hint structure is not used for read operation.

- **DPFS-Close()** The only argument is the opened DPFS file handle. This routine closes the DPFS file. It will free allocated memory for the file handle and close the database connection etc.

A significant feature of DPFS is that it allows user's involvement in low level file organization and manipulation. The hint structure provided by DPFS API, is the tool to convey user's knowledge to the low level systems. The most important information in the hint structure is the file level when the file is created. As only the user has the best picture of how her data will be utilized in the future, she can suggest a file level in hint structure, the file system then uses corresponding striping method to perform file striping.

7 User Interface

Like traditional UNIX file system, DPFS also provides a user interface which provides users with a bunch of commands that can help manage files and directories in the file system. These commands include *cp mkdir*, *rm*, *ls*, *pwd* and so on. DPFS also allows data transfer between sequential files and DPFS. This is very convenient for the user. since many data generated by parallel applications need to be transfered to a sequential workstation for post-processing such as data analysis or visualization.

8 Performance Evaluation

In this section, we present a variety of performance numbers of our experiments. The compute resource is an IBM SP2 located at Argonne National Laboratory. Each node of the SP-2 is RS/6000 Model 390 processor with 256 megabytes memory.

The external storage resources we employed in this work are disk storage on many workstations from Argonne National Laboratory and Northwestern University. These disk storage devices can be grouped into three classes according to location and network. One class (referred to as class 1 thereafter) is a bunch of Linux machines at Argonne, they are connected to compute resource SP2 by local area networks(a Fast Ethernet plus a ATM). The other two classes are located at Northwestern University. One (*class 2*) is 8 HP workstations on a local 10M Ethernet, and the other (*class 3*) is 8 SUN workstations on a local 155M ATM network. These workstations are connected to our compute resource at Argonne over a metropolitan network.

The database to maintain DPFS meta data is POSTGRES [21] installed on a Linux machine at ECE department of Northwestern University. The database access interface is standard SQL.

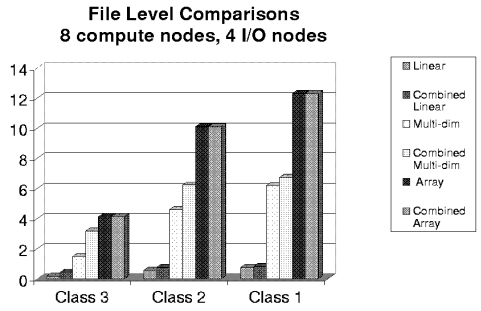

Figure 11. **I/O Bandwidth (MBytes/sec) comparisons of different File Levels.**

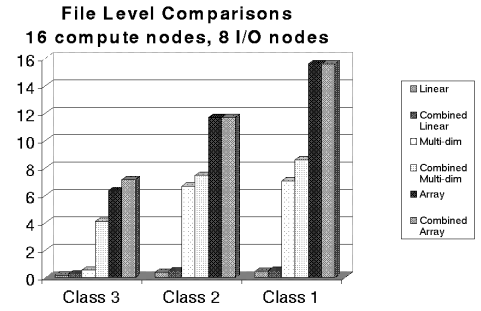

Figure 12. **I/O Bandwidth (MBytes/sec) comparisons of different File Levels.**

8.1 File Level Comparisons

As stated in Section 3, DPFS has three file levels and expects the user to choose one when the file is created. The higher the file level, the more parallelism and performance improvement can be expected. Figure 11 and 12 compare the performance difference of three file levels. The data file is a $32K \times 32K$ two dimensional array (256M). For the linear file level (linear striping), when the processors' access pattern is $(*, Block)$ which means each processor accesses a chunk of columns in the array, each processor has to access all the bricks (16K = 16384) and only part of data is really needed. This results in very poor I/O bandwidth even if request combination (Section 3) is used (Figure 11 and 12, Linear and Combined Linear). Now if the user knows that the data file is an array and it may be accessed in a $(*, Block)$ distribution, then when the file is created, she can suggest multidimensional striping through the *hint* structure provided by DPFS API (Section 6). Then the file is striped multi-dimensionally by a two dimensional 256×256 striping unit. When the file is accessed later in $(*, Block)$, each processor accesses only 128 bricks, much less than 16384 of linear striping. The performance can be improved

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 **IEEE**

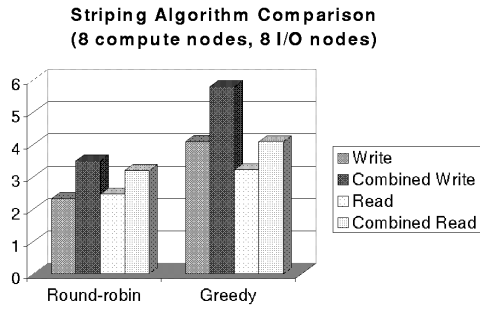

Figure 13. **I/O Bandwidth (MBytes/sec) Comparisons of Different Striping Algorithms. Half of the storage is from class 1 and half from class 3.**

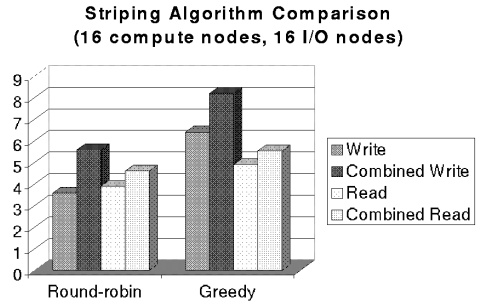

Figure 14. **I/O Bandwidth (MBytes/sec) Comparisons of Different Striping Algorithms. Half of the storage is from class 1 and half from class 3.**

10 to 20 times on all three classes of storage. When request combination technique is used, further performance improvement is also obvious (Figure 11 and 12, Multi-dim and Combined Multi-dim). Further, if the user knows that the array will be operated in a high level that can be expressed in HPF notation, she can suggest a file level of *array* and the file is divided into large chunks when the file is generated. When the file is read back later, each processor can access chunks it needs in much less requests (two in this example). The performance improvement nearly doubles compared to multidimensional striping. Request combination can not further improve performance since the number of requests of each processor is already very small (Figure 11 and 12, Array and Combined Array).

8.2 Striping Algorithm Comparison

What makes DPFS different from other parallel file systems is that DPFS employs heterogeneously distributed storage resources: the storage devices could be different from each other and they may be located on different networks. The heterogeneity brings out load balance issues in DPFS. In Section 3 we proposed a greedy striping algorithm to address load balance issues in DPFS. Figure 13 and 14 shows the comparison between greedy algorithm and round-robin algorithm. The storage includes half class 1 and half class 3 devices. Accessing a brick from class 1 is about 3 times faster than from class 3, so the greedy algorithm will assign class 1 storage as three times number of bricks as class 3. We can see that the performance has been improved obviously compared to the popular round-robin algorithm. The further performance improvement is contributed by request combination optimization.

9 Related Work

The related work can be divided into four groups.

One is distributed file systems such as NFS [19], xFS [2], and Coda [1]. These file systems provide easy access to distributed resources, but they are not designed for high performance parallel data access required by parallel applications.

Another body of work is parallel file systems, including IBM Vesta [9] and PIOFS [10], Intel Paragon [16], HP Exemplar [5], Galley [15] and so on. These parallel file systems, either commercial or experimental, take advantage of parallel I/O techniques, caching, prefetching etc to achieve significant performance improvement. The storage resources of these systems, however, are tightly coupled with the compute nodes, so they do not scale well in capacity with the increase of applications' requirements. PVFS [6] is built on Linux clusters and it does not employ external storage either.

The third group includes run-time systems such as MPI-I/O [25, 23], PASSION [7, 22], PANDA [17] and others [18]. These systems provide high level structured interfaces on top of low level native parallel file systems [12] and try to match the applications' data structure which is usually multidimensional array. Again, these systems do not help when application size increases.

The forth group includes meta data management systems [3, 18, 4, 11, 13, 8, 14]. These systems use database's query capability to automatically keep track of huge amount of datasets generated by data intensive applications.

10 Conclusions

In this paper, we have presented a Distributed Parallel File System (DPFS). DPFS collects distributed storage resources and construct them as a parallel file system to satisfy both storage capacity and performance requirements of large-scale data intensive applications.

DPFS adopts Client-Server architecture and employs TCP/IP as low level communication infrastructure. DPFS is

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 **IEEE**

built on top of the storage's local file system without changing the low level system software that can directly make use of optimization techniques such as caching and prefetching of local file system. The meta data of DPFS is maintained at the database which provides more convenience and reliability in a distributed environment. DPFS has three file levels that can make file access more efficiently. The user's involvement is highly recommended since the user is clear how her data would be accessed. DPFS API's provides a hint structure that allows user's involvement easily. DPFS also provides a *request combination* optimization approach to further improve the performance and a greedy striping algorithm to solve load balancing problems. A convenient user interface is also provided by DPFS. The user can use common file system commands such as *cp*, *mkdir*, *ls* etc to operate DPFS files and directories.

In the future, we will use DPFS for some real world applications such as astrophysics application and use DPFS as a low level system to service a high level interface such as MPI-I/O [24] and MDMS [18].

Acknowledgments

This research was in part supported by Department of Energy under the Accelerated Strategic Computing Initiative (ASCI) Academic Strategic Alliance Program (ASAP) Level 2, under subcontract No W-7405-ENG-48 from Lawrence Livermore National Laboratories. We would like to thank Prof. Banerjee and Prof. Taylor of ECE department at Northwestern University for allowing us to use their students' workstations running DPFS server.

References

[1] Coda file system. In *http://www.coda.cs.cmu.edu*.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126, 1995.

[3] C. Baru, R. Frost, J. Lopez, R. Marciano, R. Moore, A. Rajasekar, and M. Wan. Meta-data design for a massive data analysis system. In *Proc. CASCON'96 Conference*, 1996.

[4] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *Proc. CASCON'98 Conference, Dec 1998, Toronto, Canada*, 1998.

[5] R. Bordawekar, S. Landherr, D. Capps, and M. Davis. Experimental evaluation of the hewlett-packard exemplar file system. In *ACM SIGMETRICS Performance Evaluation Review*, pages 25(3):21–28, 1997.

[6] P. Carns, W. Ligon, R. Ross, and R. Thakur. Pvfs: A parallel file system for linux clusters. In *Proc. of the 4th Annual Linux Showcase and Conference, Atlanta, October 2000*, pages 317–327, 2000.

[7] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. Passion: parallel and scalable software for input-output. In *NPAC Technical Report SCCS-636*, 1994.

[8] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur. Data management for large-scale scientific computations in high-performance distributed systems. In *Proc. of the 8th IEEE International Symposium on High Performance Distributed Computing, Redondo Beach, California*, 1999.

[9] P. Corbett and D. Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[10] P. Corbett, D. Feitelson, J.-P. Prost, G. Almasi, S. J. Baylor, A. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. Morgan, and A. Zlotek. Parallel file systems for the ibm sp computers. *IBM Systems Journal*, 34(2):222–248, Janury 1995.

[11] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project.

[12] D. Kotz. Multiprocessor file system interfaces. In *Proc. the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[13] W. Liao, X. Shen, and A. Choudhary. Meta-data management system for high-performance large-scale scientific data access. In *Proceedings of the 7th International Conference on High Performance Computing, December*, 2000.

[14] Mcat. In *http://www.npaci.edu/DICE/SRB/mcat.html*.

[15] N. Nieuwejaar and D. Kotz. The galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing, Philadelphia, PA, May 1996*, pages 374–381, 1996.

[16] B. Rullman. Paragon parallel file system. In *External Product Specification, Intel Supercomputer Systems Division*.

[17] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Proceedings of Supercomputing '95, San Diego, CA, December*, 1995.

[18] X. Shen, W. Liao, A. Choudhary, G. Memik, M. Kandemir, S. More, G. Thiruvathukal, and A. Singh. A novel application development environment for large-scale scientific computations. In *International Conference on Supercomputing, May 8-11, 2000, Santa Fe, New Mexico*, 2000.

[19] H. Stern. *Managing NFS and NIS*. O'Reilly and Associates, 1991.

[20] W. Stevens. *UNIX Network Programming, Networking APIs: Sockets and XTI, Volume 1, Second Edition*. Prentice Hall PTR, Prentice-Hall, Inc., Upper Saddle River, NJ 07458, 1998.

[21] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proc. the ACM SIGMOD'86 International Conference on Management of Data*, pages 340–355, 1986.

[22] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. Passion runtime library for parallel i/o. In *Proc. of the Intel Supercomputer User's Group Conference*, 1995.

[23] R. Thakur, W. Gropp, and E. Lusk. A case for using mpi's derived datatypes to improve i/o performance. In *Proc. of SC98: High Performance Networking and Computing*, 1998.

[24] R. Thakur, W. Gropp, and E. Lusk. *On implementing MPI-IO portably and with high performance*. Preprint ANL/MCS-P732-1098, Mathematics and Computer Science Division, Argonne National Laboratory, 1998.

[25] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Proc. the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

Proceedings of the 2001 International Conference on Parallel Processing (ICPP'01)
0190-3918/01 $10.00 © 2001 **IEEE**