

# Remote Partial File Access Using Compact Pattern Descriptions\*

Thorsten Schütt, André Merzky, Andrei Hutanu, Florian Schintke

Zuse Institute Berlin (ZIB)

E-mail: {schuett,merzky,hutanu,schintke}@zib.de

## Abstract

*We present a method for the efficient access to parts of remote files. The efficiency is achieved by using a file format independent compact pattern description, that allows to request several parts of a file in a single operation. This results in a drastically reduced number of remote operations and network latencies if compared to common solutions. We measured the time to access parts of remote files with compact patterns, compared it with normal GridFTP remote partial file access and observed a significant performance increase.*

*Further, we discuss how the presented pattern access can be used for an efficient read from multiple replicas and how this can be integrated into a data management system to support the storage of partial replicas for large scale simulations.*

## 1. Introduction

In the domain of scientific computing, large, distributed data intensive jobs become more and more popular with the emerging Grid technology. In such environments, parts of huge datasets are often accessed remotely. It is difficult to do this effectively, because a potentially large number of requests and answers have to be sent across the network to select the appropriate parts of a file. Each request and answer message includes the latencies of potentially wide area network links, protocol encoding and decoding efforts, and protocol overheads. This can drastically reduce the achieved network transfer rate. In extreme cases it can be faster to request and transfer the whole file than reading parts of it.

In one of our scenarios, we try to give the user feedback on the progress of his simulation by the visualization of intermediate results. To not exhaust the available bandwidth,

these intermediate results must be sub-sampled by remote partial file access.

Additionally, for improved interactivity, we load and visualize simulation results progressively by reading regular patterns of the data in steps of increasing resolution.

We developed and implemented a new protocol for remote file access which uses a pattern description called 'nested FALLS' [12], a compact but powerful description of regular subsets of byte arrays (see Section 2). A single request containing this compact description is sent across the network and the remote server sends all requested parts of the file without further ping-pong messages. This solution is not restricted to a particular file format, but works for many different binary formats.

An API similar to the POSIX file access API is supported by our library, for an easier migration of existing applications to (partial) remote file access. Only minor changes in the code are necessary to enable access to remote files with this library. In an application, those parts that benefit from remote pattern reads have to be determined and read calls in these parts have to be replaced with *pattern* read calls.

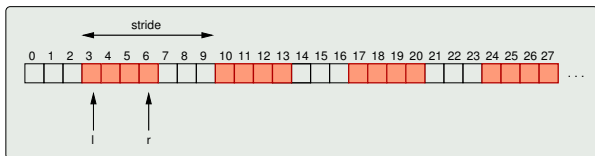
The next section describes the compact pattern description language, followed by some examples which illustrate how to apply *FALLS* to a broad range of file formats. Thereafter we present our implementation for pattern based remote file access in detail. Section 5 describes various facets of the application in data management systems. Finally, Section 6 presents the results of performance measurements for our implementation, and compares them to the performance of selected traditional methods.

## 2. Compact Pattern Description Language

We present a compact but powerful language to describe regular subsets of byte arrays, called 'nested FAmiLy of Line Segments' (nested FALLS). This schema was first used in the PARADIGM compiler for array distribution [16] and later modified by [11] for a cluster filesystem. We use it in our work for efficient remote file access.

---

\* This work was partially funded by the European Commission (the EC GridLab project, grant IST-2001-32133), and by the German Government (the DFN GridKSL project, grant TK 602 – AN 200).



**Figure 1. Family of line segments (*FALLS*), described by the pattern (3, 6, 7, 4).**

*Line Segments.* The basic building block of *FALLS* is the line segment (*LS*). A line segment ( $l, r$ ) is a contiguous part of a byte array defined by the offset of the left-most ( $l$ ) and by that of the right-most ( $r$ ) byte. The first 4 highlighted boxes in Fig. 1 can be described by the line segment (3, 6).

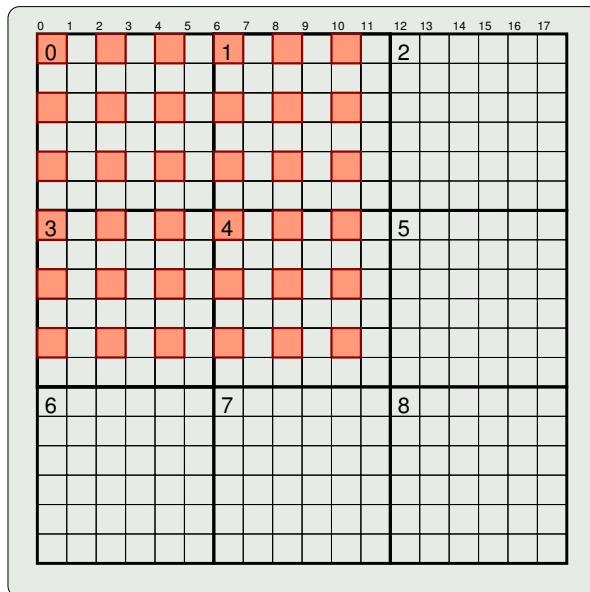
*Family of Line Segments.* A FAmiLy of Line Segments (*FALLS*) represents a set of equally spaced and equally sized line segments. A *FALLS* is defined by a 4-tuple  $(l, r, s, n)$  where  $l$  and  $r$  define the first line segment,  $s$  is the stride between two consecutive line segments and  $n$  is the number of repetitions. In Fig. 1 the highlighted boxes can be described by (3, 6, 7, 4). (3, 6) describes the first line segment, 7 is the stride and 4 specifies how often this pattern is repeated.

*Nested Family of Line Segments.* A nested family of line segments (*nested FALLS*) is defined using *FALLS*. Line segments select a contiguous part of a byte array. The result is again a byte array. So we can apply one *FALLS* to a byte array and apply another *FALLS* to each element of the result. The result is always a set of line segments. Nested falls are defined recursively where the first 4 parameters have the same meaning as for *FALLS* whereas the last parameter is either a *FALLS* or another *nested FALLS*  $(l_0, r_0, s_0, n_0, (l_1, r_1, s_1, n_1))$ .

## 2.1. Example

In the following we give an example for a *nested FALLS* to read a subset of a 2-dimensional matrix (see Fig. 2). For the moment we ignore that the matrix is distributed across 9 nodes and assume that the matrix is completely stored in a local file.

The task in this example is to read the highlighted elements. The easiest way to construct the *nested FALLS* is to start with the inner *FALLS*. Therefore we begin by extracting every other element from a line of 12 elements. The first selected element starts at offset 0 and spans one byte, the stride between two elements is 2 and we need to read 6 of these elements. This results in pattern (0, 0, 2, 6). On the outer pattern every other line from the 18x18 matrix is read: (0, 17, 36, 6). The final *nested FALLS* is created by nesting both patterns: (0, 17, 36, 6, (0, 0, 2, 6)).



**Figure 2. Two dimensional matrix distributed across 9 nodes**

Note that for *nested FALLS* there are often several patterns which describe the same subset. In this example we could replace the outer *FALLS* by (0, 10, 36, 6).

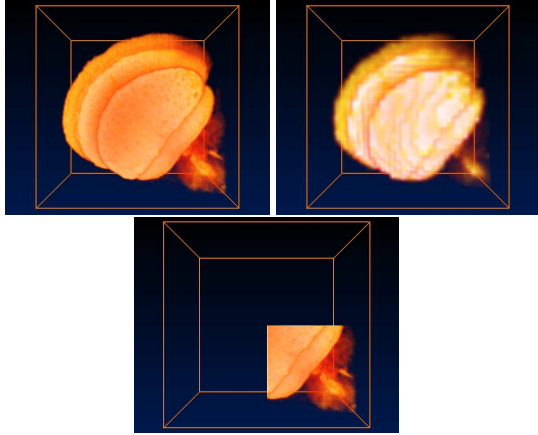
## 3. Format Independent Remote Partial File Access

The presented method to describe parts of a file is file format independent. It can be applied to any file format, but is beneficial only for binary files with parts of structured data. To use the patterns for remote partial file access, the client constructs a pattern description of the sections of interest, sends this description to the remote server (GridFTP in our implementation) and receives the requested data in one contiguous block.

In the following we describe some file formats for which the usage of regular patterns is useful. The described methods are applicable to other file formats similarly.

### 3.1. Accessing Multi-dimensional Image Data

Files storing multi-dimensional uncompressed image data are usually well suited for the application of pattern reads. A large number of file formats use 2D and/or 3D arrays to store data, like TIFF [13], AVS Field [1], AmiraMesh binary [15], BMP, PNG [8]. These file formats have an internal structure that allows the usage of patterns to subsample or to crop the data.



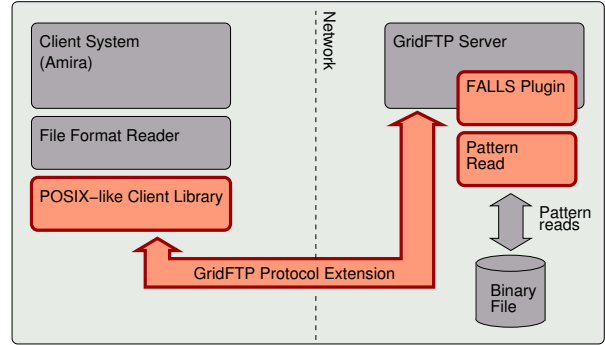
**Figure 3. Example: (1) full-resolution 3D image of the brain of a honey bee, (2) same data at 1/4 resolution, and (3) lower-right corner**

Most of these formats follow a common file layout. They start with a header containing meta data like dimensions, layout, format, and storage information, which is followed by, or contains pointer(s) to, data areas that contain the actual image data as a contiguous sequence of bytes.

Sub-sampled or cropped versions of this data can be read using patterns to generate low-resolution images for progressive visualization, or exploratory analysis of large data sets, and images. Similar operations can also be applied to 3D volume data. Figure 3 depicts the result of these operations applied on a 3D volume stored in the AmiraMesh binary format. The original data, a sub-sampled version and a 3D selection of the data were volume-rendered using the visualization software Amira [17]. Both the sub-sampled and the cropped volumes have one fourth of the size of the complete volume, the selections can be easily expressed using a *nested FALLS* containing three *FALLS*.

### 3.2. Accessing Gravitational Wave Data

As an example for complex scientific data, we re-implemented a file reader for the FrameLib file format used by the Geo600 Gravitational Wave Search experiment [14]. This format is structured similar to a file system. A table of contents (TOC) refers to other TOCs which refer to meta data and parts of binary data (Channels). Our implementation provides a read ahead cache to allow efficient parsing of the file structure of remote files. Using this cache, three remote reads are sufficient to find any particular channel in the TOCs and to read the meta data on that channel. The actual channel data itself can then be efficiently accessed using pattern reads.



**Figure 4. Our system builds on the GridFTP infrastructure - the marked elements have been implemented.**

## 4. Implementation

Our implementation of the presented remote access method with *nested FALLS* uses the GridFTP infrastructure [3], and consists of three elements, as depicted in Fig. 4:

- a library providing pattern read capabilities for local files,
- an extension to the GridFTP protocol and a plugin to support *nested FALLS* on the server side, using the pattern read library, and
- a client library providing a POSIX like file access API to perform traditional remote file reads as well as remote pattern reads.

### 4.1. Local Pattern Reads

We implemented a library in C++ for local file access based on *nested FALLS*. The central function applies a *nested FALLS* pattern to a file and returns the read bytes in a buffer provided by the user.

```
int falls_pread_buf(
    const char *file_name,
    const char *pattern,
    char *buffer,
    unsigned int buffer_size);
```

To parse the *nested FALLS* the parser generator from the boost<sup>1</sup> libraries is used. After the parsing we perform a semantic check on the pattern, e.g. the covered area is compared with the file size.

In the second step the file is memory mapped (`mmap(2)`) and the *nested FALLS* can be processed using memory to

<sup>1</sup> <http://www.boost.org/>

memory copies. This is a standard method and our tests proved that it is more efficient than direct buffered disk reads.

In Linux, memory mapped files are accessed in blocks of the page size. The page size for x86 systems, which we used, is 4 kB. In the worst case every byte read by the library invokes a read of one page from the disk. Hence, the local read performance is limited by a combination of disk speed, page size and pattern characteristics (reads per page). For sparse patterns it can happen that for each byte send across the network up to a full page has to be read from the local disk.

## 4.2. GridFTP Plugin and Protocol Extension

GridFTP [3] is an extension to the standard FTP protocol, which supports Grid authentication mechanisms using the Grid Security Infrastructure (GSI) [9], remote partial file access, multiple concurrent point-to-point data channels, gathering data from several servers [2], a generic method for protocol extensions, and custom server side data processing using a plugin mechanism.

We use the latter to extent GridFTP for pattern access on the server side. The GridFTP protocol allows to specify custom server-side processing commands through the ERET (extended retrieve) and ESTO (extended store) commands. The general pattern for ERET and ESTO is the following:

```
ESTO <plugin_name>=<plugin_parameters>" <file>
ERET <plugin_name>=<plugin_parameters>" <file>
```

`plugin_name` is a unique string identifying the plugin. The second string (`plugin_parameters`) is plugin specific and contains an operation description. The last parameter (`filename`) specifies the file to be processed.

We use the ERET mechanism to define an operation that handles remote pattern reads. This operation is implemented by a plugin for the GridFTP server that uses the library for local pattern reads. The command for reading a remote pattern is:

```
ERET FALLSPLUGIN="<nested falls>" filename
```

## 4.3. POSIX-like Client Library

On the client side, the Globus Toolkit provides a client library for GridFTP, including support for custom ERET or ESTO operations. It is cumbersome to use this library directly on the application level, because complex Globus API calls would have to be mixed with the standard read operations. Another drawback is that local and remote file accesses would be handled separately.

For these reasons we implemented a client library which resembles the standard Unix POSIX interface for file access comprising `open`, `close`, `read`, `write`, `seek` and their file stream equivalents `fopen`, `fclose`, `fread`, `fwrite`, `fseek`, `fscanf`, ..., and extended that interface with calls for pattern reads:

```
int pread      (int      fd,
               const char *pattern,
               char      **buf,
               size_t    *nbytes);
int pread_buf  (int      fd,
               const char *pattern,
               char      *buf,
               size_t    nbytes);
```

and their file stream equivalents. The two methods differ only in their memory allocation policy. The second method leaves the allocation of a sufficiently large memory area for the user whereas the first method returns the data in a newly allocated buffer. The client library itself can handle multiple I/O plugins. It currently supports local Unix I/O, remote data access via pattern reads, and remote access via GridFTP partial file access.

GridFTP Partial File Access (GPFA) is a feature provided by GridFTP as an extension to standard FTP. In fact, GPFA is the first implementation using the more generic ERET command introduced by GridFTP. Our implementation, as explained later, also makes use of the ERET mechanism, and we compare it to GPFA also for this reason. In short, GPFA allows to retrieve a section of a file at offset  $O$  with length  $L$  with a single call, thus providing us with a mechanism to make non-sequential reads (or writes).

Most file formats store meta data in small parts at the beginning of the file, whereas the actual data is stored as a huge blob (binary large object). To speed up the latency-dominated meta data reads, with large spatial locality, each read operation actually transfers a chunk of several kB that is buffered in a local cache.

If meta data is distributed across the file (as for the FrameLib format), each cache miss induces one remote operation. For the pattern reads, the cache is disabled.

## 5. Benefits in a Data Management System

Because scientists no longer work isolated in local groups, but they collaborate on an international level, scientific computing today relies to a large extent on the ability to quickly access remote files. To avoid access bottlenecks for popular files and to improve data reliability in such distributed settings, replicas of the datasets are dispersed over the Grid.

Using our pattern access, remote reads on a single replica can be speed up significantly. Having a data management

system in mind, that knows about all the replicas and controls the access to them, further improvements become possible by gathering the data from several replica locations in parallel.

## 5.1. Accessing Replicated Files

The replica catalog of a distributed data management system can be used to find several copies of a file. Accessing these replicas in parallel can improve the access performance using pattern operations where only a fraction of the data is read from each replica. Pattern reads on local disks induce many disk head movements caused by the alternating read and seek operations. Disk performance in such scenarios is poor and the available network bandwidth can easily outperform the disks. We will employ several servers to be able to saturate the client network link.

If the user wants to read the pattern  $F$  from a file  $A$  and the replica system reports 5 locations  $A_0, A_1, A_2, A_3, A_4$  for this file,  $F$  can be split into 5 disjunct patterns  $F_0, F_1, F_2, F_3$  and  $F_4$  with  $\bigcup F_i = F$ .  $F_i$  will be processed by the host where  $A_i$  is stored. Each host sends its part back to the user who then combines the results of the subqueries.

With a proper splitting of  $F$ , we can assign different amounts of work to each node to take differing performance into account. The pattern can be constructed, so that all hosts complete their task in approximately the same amount of time. Thereby the given resources are used in an optimal way. The network topology, local disk speed, etc. have to be considered to minimize the execution time.

As a first approximation we assume that the execution time is proportional to the number of bytes covered by the pattern divided by the network bandwidth between the client and the replica location. The network latency plays only a minor role here.

Splitting a  $FALLS$   $(l, r, s, n)$  can be done either by dividing the area covered by  $l$  and  $r$  into several parts or by assigning complete line segments to different hosts.  $(l, r, s, n)$  can be split to:

1.  $(l, r - x, s, n)$  and  $(r - x, r, s, n)$  or
2.  $(l, r, s, x)$  and  $(l + x * s, r, s, n - x)$

Combinations of both methods are also possible. For *nested FALLS* we can even apply several splitting methods to the different  $FALLS$  on the various levels.

The assignment of complete line segments (2nd method) of the outer  $FALLS$  to different hosts is suited for an increase in the spatial locality of file accesses. Most I/O systems work more efficient with higher spatial locality, because in this way the number of operating system internal cache hits is increased.

## 5.2. Supporting Distributed Files

Another possible application is the support of partial replicas in a replica catalog. In distributed parallel numerical simulations, for example, where each host writes local results in a separate file, the local results often represent a regular subset of the solution. Using patterns, we can register these subsets  $(D_1, \dots, D_n)$  as partial replicas. The disjunct partial replicas from several simulation hosts form a complete dataset.

To access such a distributed file with a pattern  $F$ , we have to intersect  $F$  with each  $D_i$ . This selects for each node  $i$  the parts of  $F$  that are stored at host  $i$ . Note that we cannot split  $F$  according to performance considerations, because the distribution is already given by the partial replicas.

**Example.** Figure 2 was already used to explain the  $FALLS$  concept in Sec 2. Now let us assume that the matrix is generated by a parallel application and distributed across 9 compute nodes. The task is again to read the highlighted elements. The necessary pattern is already known from Section 2 :  $(0, 10, 36, 6, (0, 0, 2, 6))$ . We can describe the distribution of the data using the following pattern  $(\lfloor \frac{i}{3} \rfloor \cdot 108 + (i \bmod 3) \cdot 6, \lfloor \frac{i}{3} \rfloor \cdot 108 + (i \bmod 3) \cdot 6 + 5, 18, 6)$  where  $i$  is the node index. Given that the data is distributed regularly across the nodes we could have used  $PITFALLS$  [16] (an extension to  $FALLS$ ) to describe the distribution, but we prefer to use  $FALLS$  to be open for more irregular patterns.

Each node  $i$  executes a different pattern  $(0, 10, 36, 6, (0, 0, 2, 6)) \cap (\lfloor \frac{i}{3} \rfloor \cdot 108 + (i \bmod 3) \cdot 6, \lfloor \frac{i}{3} \rfloor \cdot 108 + (i \bmod 3) \cdot 6 + 5, 18, 6)$ . Each node stores its subset of the data in a continuous byte array. Therefore the received data must be remapped on the client-side to merge the data from the different nodes. Algorithms for the intersection and the mapping are described in [16].

Integrated into a distributed data management system, a combination of both approaches, pattern access to replicated files and pattern distributed files, is most powerful. In such a system, arbitrary portions of files can be stored on individual nodes and registered in the replica catalog, including the registration of overlapping patterns. Then the overlapping area of two patterns represents two replicas for this area. In a system with distributed files this allows a performance driven splitting of the access pattern  $F$ , but is still more limited than the fully replicated scenario. In such an environment, achieving even load on each node by splitting of patterns is a non trivial task.

## 6. Tests and Benchmark Results

As described earlier, our implementation supports a number of file formats which make use of the pattern based remote access. Table 1 shows the results of measurements for the access to a file stored in the AmiraMesh for-

mat. For the tests we used a  $512^3$  data set of scalar float values (data size is 512 MB). The plain text header of the data file was cached locally by our library. The patterns used for the reads select sub-sampled versions of the data. The queried data is evenly distributed over the complete data set. For a  $32^3$  data selection we have a source line length  $ll$  of 512 floats, the float size  $fs$  of 4 bytes, a destination line length  $dl$  of 32 floats, and a header length  $hl$  of 339 bytes. The pattern is constructed as follows:

$$\begin{aligned}
 & (hl, \quad hl + ll^2 * fs - 1, \quad fs * ll^3 / dl, \quad dl, \\
 & (0, \quad ll * fs - 1, \quad fs * ll^2 / dl, \quad dl, \\
 & (0, \quad fs - 1, \quad fs * ll / dl, \quad dl))) \\
 = & \\
 & (339, \quad 1048914, \quad 16777216, \quad 32, \\
 & (0, \quad 2047, \quad 32768, \quad 32, \\
 & (0, \quad 3, \quad 64, \quad 32)))
 \end{aligned}$$

The inner-most *FALLS* selects every 16th float (4 bytes) from a line of 512 floats (2048 bytes). The *FALLS* in the middle works in the same way. It selects every 16th line (32768 bytes) of a plane of 512 lines. The outer-most *FALLS* selects every 16th plane of the cube, and has to skip the 339 bytes ASCII header.

We compare the performance of pattern access (*FALLS*, *RemFALLS*) with other access techniques, namely, local file access using POSIX system calls (*POSIX*), and GridFTP's remote partial file access (*GPFA*).

Local measurements were obtained on a Linux system (Pentium III 1.0 GHz, Kernel 2.4.19, 512 MB RAM). For the remote measurements the server was running on a dual Pentium 4 Linux server (Kernel 2.4.19, 1 GB RAM) and the client was the machine mentioned above. For remote file access we used a WAN connection between the Vrije Universiteit Amsterdam, Netherlands, and the Zuse Institute Berlin, Germany. The theoretical link speed between both systems is 100 Mbits and the latency about 15 ms.

**Results.** The performance as listed in Table 1 shows that local I/O is similarly efficient in both cases, *POSIX* and *FALLS*. For large cache sizes, *POSIX* performs even worse than *FALLS*, but we suspect our implementation of caching for local I/O to interfere with the systems disk cache.

The *RemFALLS* has similar execution times for all tests. That is not surprising: for the comparatively small amounts of data, the single round trip time dominates the overall execution time.

The results for *GPFA* show a significant dependency on both, the number of read items, as expected, and on the local cache. The transfer time for the largest pattern is much larger than the time needed to transfer the whole file via scp. In fact, due to the caching reads, the complete data set is transferred here, but inefficiently in multiple chunks. The results are promising, so we will apply this scheme for remote partial file access to other applications and scenarios in the near future.

Driver	Cache Size	Transferred Data	#Remote Ops	Time / s
SCP	-	512 MB	?	256.00
CP	-	512 MB	-	57.60
POSIX	-	4 B	-	0.04
POSIX	-	256 B	-	0.19
POSIX	-	16 kB	-	1.06
POSIX	-	128 kB	-	1.04
POSIX	1 kB	4 B	-	0.03
POSIX	1 kB	256 B	-	0.63
POSIX	1 kB	16 kB	-	1.10
POSIX	1 kB	128 kB	-	0.14
POSIX	1 MB	4 B	-	0.39
POSIX	1 MB	256 B	-	1.37
POSIX	1 MB	16 kB	-	2.52
POSIX	1 MB	128 kB	-	1.98
FALLS	-	4 B	-	0.29
FALLS	-	256 B	-	1.33
FALLS	-	16 kB	-	0.50
FALLS	-	128 kB	-	0.91
GPFA	-	4 B	1	0.83
GPFA	-	256 B	64	18.09
GPFA	-	16 kB	4,096	1,202.64
GPFA	-	128 kB	32,768	11,119.82
GPFA	1 kB	4 B	1	0.95
GPFA	1 kB	256 B	64	17.84
GPFA	1 kB	16 kB	2,048	70.40
GPFA	1 kB	128 kB	8,192	278.91
GPFA	1 MB	4 B	1	1.45
GPFA	1 MB	256 B	2	3.87
GPFA	1 MB	16 kB	8	7.14
GPFA	1 MB	128 kB	8	7.38
RemFALLS	-	4 B	1	1.14
RemFALLS	-	256 B	1	1.20
RemFALLS	-	16 kB	1	1.14
RemFALLS	-	128 kB	1	1.25

**Table 1. Results of the performance evaluation for local and remote accesses to a large binary data set. The various transfer sizes for partial access correspond to different resolutions of the resulting data selection ( $1^3$ ,  $4^3$ ,  $16^3$  and  $32^3$ , ...).**

## 7. Related Work

The DataCutter Framework [6, 7] also provides flexible indexing and filtering of distributed datasets. The indexing service provides access to subsets of the complete dataset. The filtering service can execute multiple filters during access. Filters can be connected with pipes and make a stream processing possible. For the indexed access additional index files are stored in the system. Filters are C++ objects which are derived from a common filter class. Filters can be placed and executed on any host in the system. DataCutter can be used in conjunction with the Storage Resource Broker SRB [5].

*FALLS* and *PITFALLS* are already used to design a parallel file system named clusterfile [12, 11]. Clusterfile files

can be seen as a very flexible implementation of RAID 0. On RAID level 0 the filesystem is striped over several disks. Traditionally the stripe size is a fixed parameter. For several applications disk access performance can be enhanced by modifying the stripe size. In clusterfile the stripe pattern can be selected per file using a *PITFALLS* pattern. In Section 5 we described how this idea can be extended to Grid-like environments.

[10] used the concept of plugins to GridFTP servers for implementing file format specific data access capabilities. These are, from an architectural point of view, very similar to the solution presented in this article. They offer substantially better performance for unstructured data, and comparable performance for structured data. The main penalty for a format specific plugin is the overhead for development and deployment in the target environment. The FALLS based access patterns presented here, span a broader range of potential use cases with a single implementation. [10] presented a plugin for the HDF5<sup>2</sup> file format. This is a very powerful file format, but it has a complex structure. We are currently investigating whether it is feasible to write an HDF5 reader based on the *FALLS* library. It would be interesting to measure the overhead of the generic approach compared to the file format specific plugin.

## 8. Conclusion

We presented a new approach for remote partial file access based on a compact pattern description language. The major advantage of this approach is its file format independence. E.g. a file-format specific plugin for the GridFTP server would have a comparable or better performance, but it would be limited to one file format. Especially in Grid environments it is difficult to install software for every used file format on all servers.

While we showed that FALLS based patterns are very useful for regularly structured binary data formats, their range of use is limited: these patterns are not applicable for ASCII data, and for unstructured binary data, such as JPEG, wavelet or RLE compressed data, or data stored on unstructured grids. The worst case performance to be expected from the presented framework is similar to the performance of cached remote Partial File Access via GridFTP. On the other hand, the FALLS patterns are trivially exploitable for simple file formats, as described above. Our POSIX-like IO-Wrapper library further simplifies the implementation of file readers based on *nested FALLS*.

We use the described approach in the GriKSL and the GridLab [4] project. Currently we are trying to improve our model by including predictions of the execution time for patterns, whereas on the implementation side we will en-

hance an existing replica catalog/data management system to support distributed files using patterns.

## References

- [1] Advanced Visual Systems Inc. *AVS/Express Field data type*, 2001.
- [2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, May 2002.
- [3] W. Allcock, J. Bester, J. Bresnahan, S. Meder, P. Plaszczak, and S. Tuecke. GridFTP: Protocol extensions to FTP for the Grid. *GWD-R (Recommendation)*, April 2002.
- [4] G. Allen et al. Enabling applications on the Grid - a GridLab overview. *Intl. Journal on High Performance Computing Applications*, 17(4):449–466, 2003.
- [5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. *Proceedings of CASCON'98*, 1998.
- [6] M. D. Beynon et al. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.
- [7] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the 2000 Mass Storage Systems Conference*, pages 119–133, College Park, MD, Mar. 2000. IEEE Computer Society Press.
- [8] T. Boutell. RFC 2083: PNG (portable network graphics) specification version 1.0, Jan. 1997.
- [9] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational Grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, 1998.
- [10] H.-C. Hege, A. Hutanu, R. Kähler, A. Merzky, T. Radke, E. Seidel, and B. Ullmer. Progressive retrieval and hierarchical visualization of large remote data. *Proceedings of the Workshop on Adaptive Grid Middleware*, September 2003.
- [11] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. *Proceedings of IEEE Cluster Computing Conference*, October 2001.
- [12] F. Isaila and W. Tichy. Mapping functions and data redistribution for parallel files. *Proceedings of IPDPS 2002 Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, April 2002.
- [13] A. Katz and D. Cohen. RFC 1314: A file format for the exchange of images in the Internet, Apr. 1992.
- [14] LIGO Data Group and VIRGO Data Acquisition Group. Specification of a common data frame format for interferometric gravitational wave detectors (IGWD), August 1997.
- [15] Amira 3.0 user's guide and reference manual, 2002. <http://www.amiravis.com/Amira30-manual.pdf>.
- [16] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory

---

2 <http://hdf.ncsa.uiuc.edu/HDF5/>

multicomputers. In *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

- [17] D. Stalling, M. Westerhoff, and H.-C. Hege. *Visualization Handbook*, chapter Amira - a Highly Interactive System for Visual Data Analysis. Academic Press, 2004.