Data access performance through parallelization and vectored access. Some results.

Fabrizio Furano[1], Andrew Hanushevsky[2]

Abstract. High Energy Physics data processing and analysis applications typically deal with the problem of accessing and processing data at high speed. Recent study, development and test work has shown that the latencies due to data access can often be hidden by parallelizing them with the data processing, thus giving the ability to have applications which process remote data with a high level of efficiency. Techniques and algorithms able to reach this result have been implemented in the client side of the Scalla/xrootd system, and in this contribution we describe the results of some tests done in order to compare their performance and characteristics. These techniques, if used together with multiple streams data access, can also be effective in making possible to efficiently and transparently deal with data repositories accessible via a Wide Area Network.

1. Introduction

In this work we address the problem related to the performance of data analysis jobs which access a data repository, which is not necessarily connected to the same local network as the machines where the computing is performed. An easy to realize rule of thumb says that the data access performance gets higher if the computing elements are 'close to the data', where the shortest distance may be identified with the CE local disks, and the longest distance may be identified with a storage facility connected through a Wide Area Network (WAN). Given enough throughput, even a WAN seems capable of giving data to a running process, however, the achievable performance has historically been very low for applications in the High Energy Physics domain. This is generally due to:

- the characteristics of WAN data streams (high latency, low achievable bandwidth per TCP connection);
- the usual structure of HEP data analysis applications, which, in their deeper simplification, consists in a loop of get chunk - compute chunk instructions.

In [5] the characteristics of WAN networks and of some HEP requirements are discussed, in order to introduce some methods to enhance their data throughput. Such techniques, schematized in Figure 3 have been implemented in the Scalla/xrootd system [1] [3], and the results of some performance tests through WANs will be discussed.

The typical situation of data centers offering computing services for HEP experiments usually comprehends computing facilities and storage facilities. In this environment, a simple application which has to analyze the content of some files, typically will:

[1] Present address: INFN sez. di Padova, via Marzolo 8, 35131 Padova-Italy (furano@pd.infn.it)

[2] Present address: SLAC - Stanford Linear Accelerator Center, 2565 Sand Hill Road, 94025 CA (abh@slac.stanford.edu)

- open the files it has to access
- cycle through the stored data structures, performing calculations and updating the results
- output in some way the final results

From here, we assume that those data access phases are executed sequentially, as it is in most data analysis applications. These considerations refer to the concept of file-based data store, which is a much used paradigm in HEP computing; however, other data access methods can be affected by the same performance issues.

One of the key issues we try to address is the fact that the computing phase of a HEP application is typically composed by a huge number of interactions with the data store. Hence, a computing application must deal with the fact that even a very short mean latency (e.g. 0.1 milliseconds) would be multiplied by the huge number of interactions (e.g. 10^7).

This argument has historically been considered a serious issue which makes it impossible for data analysis applications to access remote repositories with a high degree of efficiency. However, there are situations where this is not true. A trivial example of such a situation is when the application does not need to read all the content of the files it opens [5]. More complicated scenarios, instead, involve complex data analysis applications which can predict in some way which data chunks they are going to access. This kind of applications, if supported by adequate data access technologies, can enhance their performance of an order of magniture, reaching levels comparable to those achievable through local access.

2. Read requests and latency

Figure 1 shows that, in a sequence of read requests from a client to a server, the transmission latency affects the requests twice, and is located before and after the server side computation. If we suppose that the latency is due to the network, and that the repository is not local to the computing client, the value of the latency can be even greater than 70-80 milliseconds. With a latency of 80 milliseconds, an application in Padova requesting data from a server at SLAC, for example, will have to wait 160 seconds just to issue 1000 data requests. However, the work done at INFN Padova, SLAC and CERN addresses the problem in a different way. If we can get some knowledge about the pattern (or the full sequence) of the data request issued by the application, the client side communication library can, in general, request in advance the data for the future requests, and store portions of it in memory, while other portions of it are still in the network. Moreover, the future data requests can be issued in parallel with respect to the present ones, getting the advantage of having, in any phase of the computation, outstanding data which has a very high probability of containing the responses of the future data reads.

3. Read requests and throughput

The other parameter which is very important when dealing with data access is the data rate at which the computing process requests new data chunks to process, and its comparison with the data throughput that the data network can sustain. One of the characteristics which make WANs difficult to deal with in the case of data access is the fact that, even if the throughput of the network could be sufficient to feed the data processing, a single TCP stream through a WAN is typically able to tranfer data only at a fraction of the available bandwidth. This can be related to the characteristics of the commonly used TCP stacks, and also to the common configurations of the network devices, and led to the development of tools able to transfer data files at high speed by using a large number of parallel TCP connections (BBCP, GRIDFTP, XRDCP et al.). The approach of the Scalla/xrootd system with respect to this aspect is to apply a similar mechanism to the transfer of an analysis application-requested data chunks. The main difference from a copy-like application is that, generally, a data analysis application generates a stream of requests for data chunks whose sizes are an order of magnitude smaller (some KBs versus

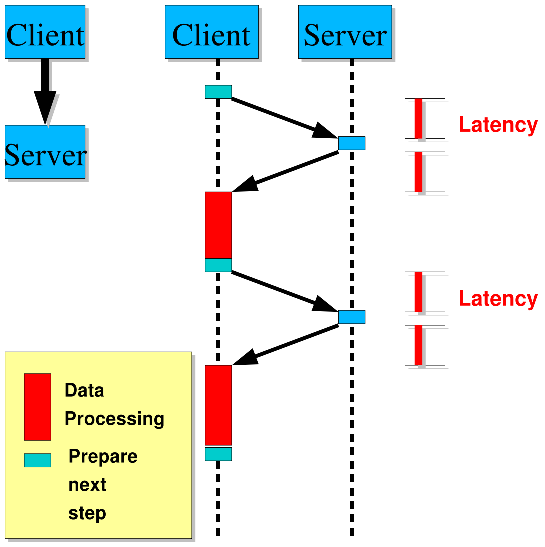


Figure 1. Role of the latency in read operations.

512-1024KB) than the chunk sizes used for copy-like applications. This is supposed to have some impact on the achievable data rate in the case of a data analysis application. An evaluation of this impact is the purpose of the measurements described later in this work. In general, a desirable feature would be the ability of the communication library to:

- split a request for a big data chunk into smaller chunks to be distributed through multiple parallel TCP streams
- join multiple requests for small chunks into a bigger one, to achieve a higher efficiency in the WAN data transfer through a single TCP stream.

4. Where we are

In the Scalla/xrootd system, to lower the total impact of latency on the data access, in the client side communication library (*XrdClient* or *TXNetFile*) we exploited these techniques:

- a memory cache of the read data blocks;
- the client is able to issue asynchronous requests for data blocks, which will be carried out in parallel;
- the client exposes an interface for the application to be able to inform it about its future data requests;
- the client keeps track of the outstanding requests, in order not to request twice data chunks which are in transit and to make the application wait only if it tries to access data which is currently outstanding.

Figure 2 shows a simple example of how such a mechanism works. The client side communication library can speculate about the future data needs, and *prefetch* some data in advance at any moment. In alternative, the API it exposes can be used by external mechanisms (like the ROOT's *TTreeCache*) to inform it about the future data chunks to request. The informed prefetching mechanism drastically optimizes the number of requests which have to be forwarded to the server, thus reducing the overall data transfer latency, and is used as a buffer to coordinate the decisions about the outstanding data chunks to be waited for. Moreover, for the intrinsecally asynchronous architecture involved, the client side communication library is able to receive

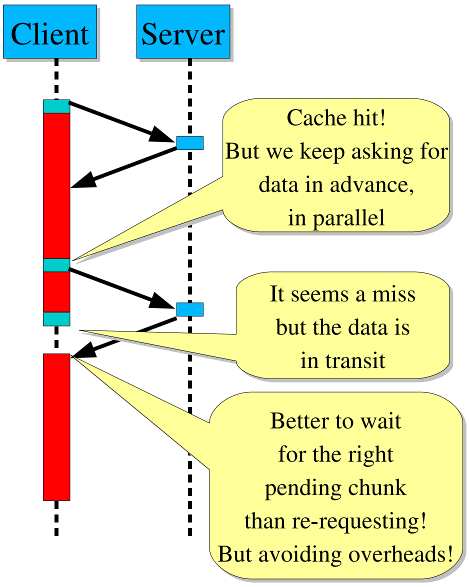

Client
Server
Cache hit! But we keep asking for data in advance, in parallel
It seems a miss but the data is in transit
Better to wait for the right pending chunk than re-requesting! But avoiding overheads!

Figure 2. Prefetching and outstanding data requests.

the data while the application code performs its processing on the already received data, thus reducing the impact of the data access on the data processing. For more information about caching and prefetching, the reader is encouraged to see [2].

Figure 3 shows some typical scenarios for remote data access, sorted in the way they have (or will be) implemented and evaluated. The most obvious one is the first one, where the needed files are transferred locally before starting the computation. This solution, much used for historical reasons, forces the application to wait for the successful completion of the data transfer before being able to start the computation, even if the accessed data volume is a fraction of the transferred one. The second scenario refers to the application paying for the network latency for every data request. This makes the computation very inefficient, as discussed. The third one, instead, shows that, if the data caching/prefetching is able to keep the cache miss ratio at a reasonably low level (or the application is able to inform the communication library about the future data requests), the achievable result is to keep the data transfer go in parallel with the computation, but a little in advance with respect to the actual data needs. This method is more efficient than the previous discussed ones, and in principle could represent a desirable solution to the problem. However, if the data demanding application generates requests for very small data chunks, the overhead of transferring and keeping track of a large number of small outstanding chunks could have a measurable impact, which we try to evaluate in the following section.

If the client side concern moves to the fact that individually transferring small chunks (although in a parallel fashion) can degrade the data transfer efficiency, one obvious solution is to collect a number of data requests and issue a single request containing all of them. This kind of policy, known as *vectored reads* involves also the fact that the response itself is constituted by a unique chunk containing all the requested ones. This kind of solution can give a good advantage over the policy of transferring individual small chunks, but, in principle, has the disantvantage of requesting both client, network and server to serialize the composite data request, in order to manage its response as a unique data block to transfer. This means that, for example, the application has to wait for the last subchunk to be arrived before being able to process the first one. An interesting idea is to try merge the advantages of individual asynchronous chunk

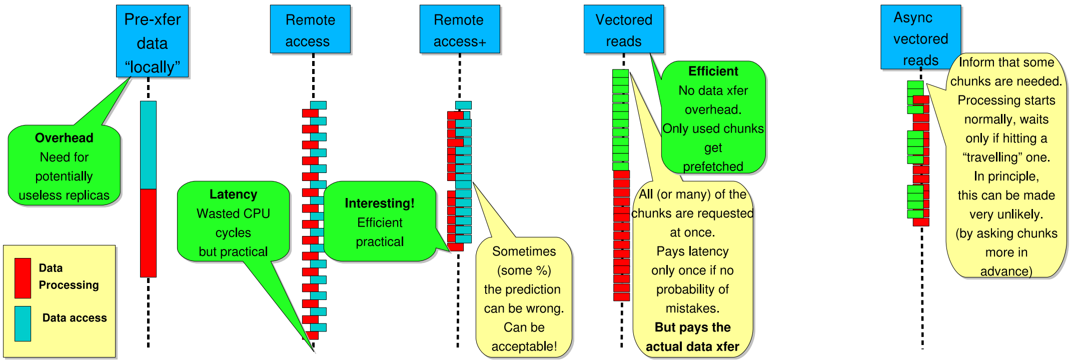

Pre-xfer data "locally"
Remote access
Remote access+
Vectored reads
Async vectored reads
Overhead Need for potentially useless replicas
Latency Wasted CPU cycles but practical
Interesting! Efficient practical
Sometimes (some %) the prediction can be wrong. Can be acceptable!
Efficient No data xfer overhead. Only used chunks get prefetched
All (or many) of the chunks are requested at once. Pays latency only once if no probability of mistakes. **But pays the actual data xfer**
Inform that some chunks are needed. Processing starts normally, waits only if hitting a "travelling" one. In principle, this can be made very unlikely. (by asking chunks more in advance)
🟥 **Data Processing**
🟦 **Data access**

Figure 3. Simple and advanced scenarios for remote data access technologies.

transfers with the vectored data transfers, leading us to the last scenario visible in Figure 3. The basic idea shown is to request data transfers through *vectored reads* which:

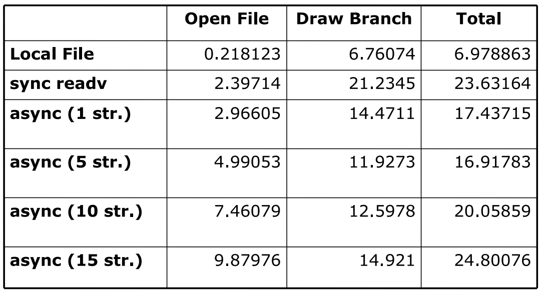
- contain enough subchunks to be able to reduce the protocol overhead linked to a client/server request;
- generate responses whose data chunk is sufficiently large to be efficiently transferred through the network pipe;
- are small enough to avoid an excessive serialization of the requests sequence.

The result would be that the client application reads the data chunks it needs in small bursts, spread through the time line of the application, thus giving a more uniform load profile on the involved data server.

5. A real world evaluation

The first preliminary evaluation of the performance level achievable with such techniques was done in a testbed set up at CERN, and reading data from SLAC. The TCP round trip time between SLAC and CERN is about 160ms, hence a simple legacy synchronous application should not be able to issue more than 6-7 requests per second. At SLAC an xrootd server was set up in order to give access to a ROOT file containing data to histogram. The file was 250MB in size, and its histogramming involves about 10000 data request interactions. The table in Figure 4 shows the results of the first test. In the tests, all the techniques described in Figure 3 were evaluated, except the last one, since the current implementation of the data access tools does not yet support it in conjunction with the multistream data transfer. Also, the results of the test related to the first simple data access scheme (synchronous data reads) were not included in the table, since they were two orders of magnitude worse (more than 1800 seconds).

A closer analysis of the data visible in Figure 4 confirms the good performance improvement which is achievable using the right technique to avoid the impact of network latencies on performance. The usage of multiple parallel TCP streams in the processing also gave good results, reducing the processing time to the same order of magnitude measured when accessing the data file locally (12 seconds versus 6.7 seconds). Also, we have to remark that the nature of this kind of processing is quite particular, since the creation of a histogram is computationally very simple, hence there is almost no computing time under which the *XrdClient* communication library could hide the latencies under. For this reason, for applications doing more computations on the data they read we expect a lower difference between the local and remote data access performance. However, a deeper look at the shown information gave us some surprises, which suggested that there is some space for further improvements of the data access techniques under



	Open File	Draw Branch	Total
Local File	0.218123	6.76074	6.978863
sync readv	2.39714	21.2345	23.63164
async (1 str.)	2.96605	14.4711	17.43715
async (5 str.)	4.99053	11.9273	16.91783
async (10 str.)	7.46079	12.5978	20.05859
async (15 str.)	9.87976	14.921	24.80076

Figure 4. First evaluation of the available data access algorithms with an increasing number of parallel TCP streams between CERN and SLAC. Courtesy of Leandro Franco and the ROOT team.

evaluation. First of all, we can note that, for a single stream test, reading asynchronously the single small chunks (a few KB each) seemed faster than reading them as a unique vectored read. At the moment there is not a clear explanation of this behavior. The most probable reason could lie in the fact that a vectored read inherently serializes the data communication on the TCP stream, but this point of view needs further investigations. Another worth noting aspect was that 15 data streams sustained a data rate worse than with 10 streams. This might be due to difficulties in evenly filling the TCP streams with a huge number of requests of different sized small data chunks. Moreover, we can note that the application startup time was influenced by the time the *XrdClient* communication library takes to establish the multiple TCP streams over a high latency WAN. From the table, we can see that opening streams towards SLAC takes 0.6 secs per stream, and that the streams in the test were opened in a consecutive way, thus needing up to 10 seconds to initialize in the 15 data streams case. This issue will be addressed in the near future releases of the *XrdClient* communication library. However, the problem was not considered as dramatic, since that is a latency paid only once when a client application contacts a data server. Moreover, once established, the TCP connections are shared between multiple clients/files through the *XrdClient* connection multiplexing, thus reducing the overall impact on the computations.

To be able to decide where it is worth optimising, and what to do and what to expect from the point of view of the performance, a more comprehensive test was set up, involving:

- a data server in Padova;
- a client benchmark application running at SLAC (a noric machine. The TCP round trip time with Padova is about 160ms);
- a client benchmark application running at IN2P3 (a ccali machine. The TCP round trip time with Padova is about 23ms).

The benchmark workload consisted in a sequence of 50K requests for chunks of 10KB, for a total of $5*10^8$ bytes to read from a single remote file through an xrootd server. The size of the remote file is not important, since what is taken into account is the number of bytes requested/transferred.

Since the main purpose of the benchmark was to better understand the characteristics of the performance achievable through different data access techniques, the following ones have been considered:

- synchronous reads
- asynchronous reads in advance. The number of outstanding requests was fixed to 1024.
- huge readaheads. The block size was fixed to 10MB, managed by the *XrdClient* library.

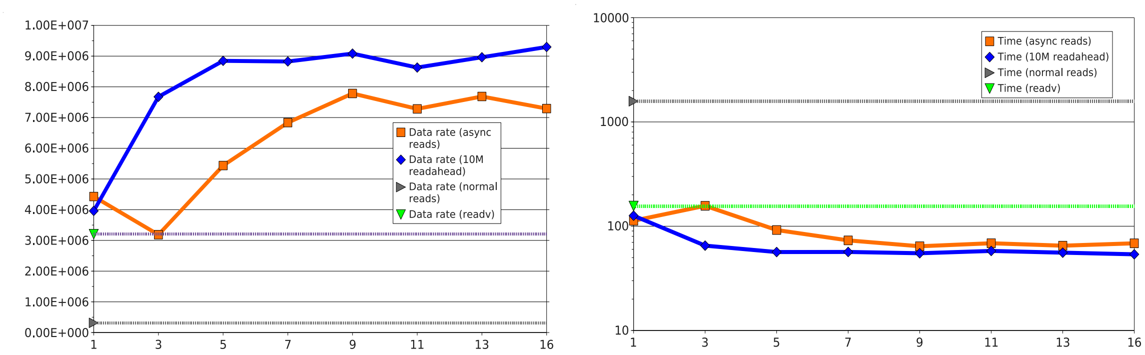


Figure 5. Data rate and processing time between Padova and IN2P3 (ccali machines) with various data access techniques and parallel stream counts.

- synchronous vectored read. The number of requested subchunks was fixed to 20480.

Some things to point out about this benchmark are that:

- the performance of the disks at the server side are of secondary importance, if compared to the network latency and throughput;
- for the same reason, the order in which the requests are sent to the server is not important from the disk and network point of view;
- only the data transfer technique based on large block transfers in advance needs (by definition) that the requests are sequential with respect to the offset they specify inside the file.

The results of the benchmark executed between Padova and the ccali machines in Lyon (IN2P3) show some particularities. Some of them are to be exploited in order to come up with more efficient data transfer schemes.

- With no additional streams (1 stream) the asynchronous chunk tranfers look slightly faster than the synchronous readv. This was a surprise, and could be related to the additional parallelism that the asynchronous requests exploit both at the client and server side. Moreover, a delayed TCP packet in the case of the vectored reads can put in stall the whole huge composite chunk which is being transferred, while in the case of the asynchronous reads, other data chunks can be processed by the client side communication library.
- In this case, the used network bandwidth tends to be saturated even with a few parallel streams.
- 2 data streams (in the 3 streams case, since one stream is used as a coordination stream in the client/server architecture used) seem to perform worse than 1 data stream. This effect needs to be better understood, but it could be related to how the server side reacts to a very large number of requests coming almost at the same time and queued through a few parallel streams.
- The data rates obtained with the large read ahead chunks are comparable with the data reads obtained with the asynchronous reads (9MB/s versus 7.5MB/s), but some space for improvements remains for the latter technique.
- The maximum performance increase obtained with the asynchronous chunk transfers versus the simple synchronous data transfers is about a factor of 24.
- The maximum performance increase which we believe should be obtainable with more optimized data transfer schemes is about a factor of 28.

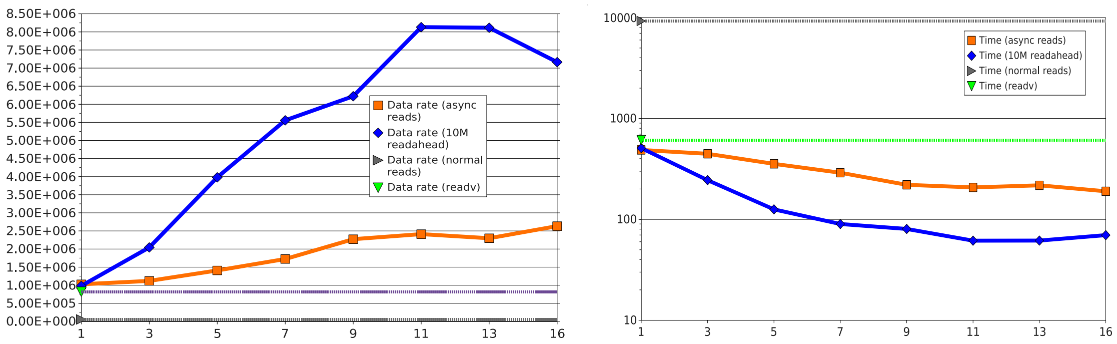


Figure 6. Data rate and processing time between Padova and SLAC with various data access techniques and parallel stream counts.

Similar, but not equal, particularities can be seen in the results of the benchmark execution between Padova and SLAC:

- with no additional streams (1 stream) the asynchronous chunk tranfers still look slightly faster than the synchronous readv;
- in the asynchronous transfer case, using up to 15 more TCP streams does not generate a performance increase greater than 3 times. This could be related to the fact that the single transferred chunks are quite small (10KB), and the client and the server (due to the high network latency) are less efficient in managing the queues of the outstanding ones;
- the data rates obtained with the large read ahead chunks are of 9-10MB/s, roughly 150 times the performance of the simple synchronous reads schema;
- with an increasing number of streams there is the suspect that the performance may decrease in the large read ahead case. This can be due to the fact that the large read ahead block is splitted through the parallel streams, hence, adding more streams reduces the size of the sub-chunk (differently from what happens in copy-like applications);
- in total, the maximum performance increase we can see of the asynchronous data transfer technique versus the simple synchronous requests is about 44 times. For sure there is space for optimizations, since the theoretically achievable performance increase is about 150 times, if we consider the performance of the large read ahead case as the maximum achievable.

6. Conclusion

In general we can say that the results of the various benchmarks are quite promising, and that some of the evaluated techniques already perform at a level more than sufficient to be used to actually analyze and process remote data in several application domains, depending on the available network bandwidth. The fact that the maximum performance achievable with small chunk transfer is lower than the performance achievable with large chunks suggest the idea that gluing together a number of small chunks and transferring them together through multiple streams could fill a big part of the performance gap detected. However, at this time, the used Scalla/xrootd system implements almost all the seen techniques, but is unable to asynchronously transfer vectors of chunks through multiple streams. This feature will be for sure available in the next releases of the platform in the near future. We recall also that all these techniques rely on some knowledge (from the perspective of the application or of its software framework) about the subsequent data accesses. Applications which perform sequential accesses which cover the major part of the file they are reading can already use a simple read ahead scheme, which gives very good performance. Applications or frameworks using the features of TTrees in the ROOT-I/O packages instead can rely on the TTreeCache features, which, in the upcoming releases, will exploit the more general asynchronous API exposed by the Scalla/xrootd client XrdClient. Our point, in general, is that the level of performance achieved and achievable, together with features like the fault tolerant architecture and communication and the Parallel Open feature, are together characteristics which are going to make it possible to give alternative choices for data or replica placement. Doing this way, the systems designers could choose why their systems should need replicas (more reliability, more performance, willing to 'own' a copy of the data, and other reasons), but without forcing every subsystem to get and keep track of potentially useless ones.

References

[1] A. Hanushevsky, A. Dorigo, P. Elmer, and F. Furano. The next generation ROOT file server. In CHEP'04 - Computing for High Energy Physics. CERN, 2004.

[2] Patterson R. H., Gibson G. A., E. Ginting, Stodolsky D., and Zelenka J. Informed prefetching and caching. In Proceedings of the 15th ACM Symposium on Operating Systems Principles., 1995.

[3] Andrew Hanushevsky. Hyper-scaling data access: understanding XROOTD data clusters. In ROOT Workshop 2005. CERN, September 2005.

[4] Fabrizio Furano. Large Scale Data Access: Architectures and Performance. Ph.D. thesis TD-2006-1, Universita' Ca' Foscari Venezia, 2006.

[5] Fabrizio Furano, Andrew Hanushevsky, Peter Elmer, Gerardo Ganis. Latencies and Data Access. Boosting the performance of distributed applications. In Proceedings of Computing for High Energy Physics 2006, 2006.

[6] A Conversation with Michael Stonebraker and Margo Seltzer. Relating to databases. ACM Queue vol. 5, no. 4 - May/June 2007.

[7] Bell, G.; Gray, J.; Szalay, A.; Petascale computational systems, IEEE Computer, vol. 39, Issue 1, Jan. 2006 Page(s):110 - 112