LA-UR- 08-5522

Approved for public release;
distribution is unlimited.

Title:	PARALLEL PROCESSING OF DATA, METADATA, AND AGGREGATES WITHIN AN ARCHIVAL STORAGE SYSTEM USER INTERFACE
Author(s):	MARK A. ROSCHKE DANNY P. COOK BART J. PARLIMAN C. DAVID SHERRILL
Intended for:	I.E.E.E. SNAPI 08 WORKSHOP, SEPT 22, 2008, BALTIMORE, MD.

Los Alamos
NATIONAL LABORATORY
EST. 1943

Form 836 (7/06)

Parallel Processing of Data, Metadata, and Aggregates Within an Archival Storage System User Interface (Toward Archiving a Million Files and a Million Megabytes per Minute)

Mark A. Roschke
High Performance Computing Division,
Los Alamos National Laboratory
mar@lanl.gov

Bart J. Parliman
High Performance Computing Division,
Los Alamos National Laboratory
bartp@lanl.gov

Danny P. Cook
High Performance Computing Division,
Los Alamos National Laboratory
dpc@lanl.gov

C. David Sherrill
High Performance Computing Division,
Los Alamos National Laboratory
dsherril@lanl.gov

Abstract

Archiving large datasets requires parallel processing of both data and metadata for timely execution. This paper describes the work in progress to use various processing techniques, including multi-threading of data and metadata operations, distributed processing, aggregation, and conditional processing to achieve increased archival performance for large datasets.

1. Introduction

Ever-increasing computing capabilities result in ever-increasing data sets to be archived. Such data sets can consist primarily of large files, many small files, or both. Archiving data sets with large files requires an emphasis on parallel file transfer, utilizing as much bandwidth as possible to transfer data to the archive. And it is in this area that the majority of parallel archival development has occurred. When a data set includes a large number of small files, the archiving process must also include an emphasis on processing large amounts of file system and archival system metadata, not only when storing into the archive, but also when retrieving from the archive and when browsing/mining/maintaining the archive. As archival data sets accumulate over time, multiplying the amount of metadata owned by a single user, the need for high performance metadata processing also increases. Thus, there is an increasing need for parallel capabilities in handling large amounts of both data and metadata.

2. Overview of PSI

The Parallel Storage Interface (PSI) is an archival system user interface designed to provide high speed archiving for large data sets, with a special emphasis on focusing as many resources as possible on a single user request. Developed by the authors, PSI is the main user interface to the High Performance Storage System (HPSS) at Los Alamos National Laboratory. This paper describes the efforts to utilize PSI to achieve archival rates of a million files and a million megabytes per minute. The emphasis is on single user/single command performance, as opposed to the overall capabilities of the file system or the archival system.

While there have been efforts in the past to provide parallel Unix commands, these efforts [1] have largely been based upon achieving parallelism by running existing serial Unix commands on more than one host at a time. In contrast, PSI uses aggressive multi-threading per host to control and execute the various aspects of the archiving process.

PSI is based upon a multi-node, light weight, passive client model, with the majority of the software residing on the archival server. PSI uses a parallel work flow model for processing both data and metadata. Work is parallelized and scheduled on available server and multi-node client resources automatically, using a priority and resource-based approach. Optimization is performed automatically, including areas such as parallelization, optimized tape transfer, load leveling, etc. PSI utilizes UNIX-like

syntax and semantics. Areas discussed include parallel techniques used within PSI, results obtained, and perceived impediments to further performance increases.

3. Need for Both Parallel Data and Parallel Metadata Processing

Archiving very large data sets places a variety of high performance requirements on an archival storage system. While past efforts have focused primarily on meeting bandwidth requirements for data sets with large amounts (terabytes) of data, this focus is not sufficient when dealing with data sets consisting of large numbers of files. Data sets requiring archiving frequently exceed 100,000 files, and one million file data sets are no longer rare. High performance metadata access is required for these large data sets, both for the file system containing the data as well as for the archival storage system receiving the data.

The requirements for high performance vary with the file sizes being archived as well as with the various stages of archiving activity. For datasets dominated by large files, the initial archiving process is dominated by bandwidth-related activity. Once the initial archiving process is over, any subsequent incremental archiving can introduce the need for high performance metadata access on both the file system and the archival system to determine which files need to be archived. (This type of incremental archiving can either be the result of an interrupted initial archiving attempt, or the result of the application adding more data to the original data set residing in the local file system, and which now needs to be archived). Once the data has been fully archived, the need for performance becomes either the need to query the archive for file names and attributes, or the need to retrieve data from the archive (requiring both archival metadata performance plus file transfer performance).

For data sets dominated by small files, bandwidth plays a reduced role, with metadata access and I/O latency dominating all of the various archiving activities; high metadata performance is required from both the file system and from the archival system.

When such data sets are accumulated over time, scanning these combined data sets requires even greater archival metadata performance. For example, archiving one million files per hour requires "stat" rates and I/O rates of less than 300 files per second. However, to scan 60 existing data sets this size in an hour requires almost 17,000 file attribute calls per second, beyond what most file systems and archival systems are capable of providing, especially in response to a single user request.

4. Techniques Used for Performance Increases

The general approach chosen involves the use of parallel data and metadata processing, automatic optimized file aggregation and de-aggregation, and conditional operations when feasible. Combining these three features provides a variety of performance increases. For example, multi-threading to a degree of 40 threads might increase performance by a factor of 30 or so, while operating on a file aggregate of 1000 files can provide a performance boost of up to 300. Conditional operations can provide a factor of 20 or so. By combining these three features, performance gains of over 1,000 have been observed, as outlined below.

4.1. Multiphase Parallel Work Flow

To facilitate efficient control of the various steps required to execute user requests in a parallel fashion, tasks are organized into three phases. Each phase can consist of many threads, each requiring different resources. Achieving high performance in processing metadata requires a reasonably high degree of parallelism; typical thread counts for all three phases is 50 to 150, depending upon the mix of metadata and file transfer operations being performed.

4.1.1. Phase 1 – Parallel Tree Traversal. Virtually all user requests require attributes of files in order to be processed. These files may reside on either the archival system or the file system. For example, a command to store files would begin by obtaining attributes of files on the file system, while a command to retrieve or list files would begin by obtaining attributes of files on the archival system.

During phase one, each thread is assigned to one directory. This encourages scalability and avoids the loss of efficiency whenever threads are serialized by the kernel on per-directory operations, such as occurs on Linux when more than one thread attempts to "stat" in the same directory. During parallel tree traversal, whenever a directory is encountered, it is put onto the work list for phase one. This triggers the spawning of another thread to process that directory, provided sufficient resources are available. The thread count for phase one typically is in the range of 20 to 50.

4.1.2. Phase 2 – Parallel Processing By Directory. Any processing other than that described for phase one, occurs in phase two, with the exception of file transfers. Phase two parallel tasks includes such operations as finding files, changing permissions, showing storage used, listing file attributes, etc. Other

than file transfers, all file operations are performed on a directory basis. Each directory is processed in a separate thread in order to obtain more predictable multi-threading performance, given that some systems tend to lock directories when updating, etc. The thread count for phase two typically is in the range of 10 to 50.

4.1.3 Phase 3 – Parallel Processing By File. Phase three consists of all file transfer operations. The resources consumed by threads in phase three are normally different from, but much greater than, resources needed by threads in the first two phases. Priority is given to threads in later phases to avoid resource starvation by threads in earlier phases, thereby allowing work to flow with less interruption. The thread count for phase three typically is in the range of 10 to 50.

4.1.4. Heterogeneous Tree Traversal. Many user requests can be performed entirely in a parallel mode, such as those involving file transfers, change of permissions, and others. For those user requests that require that the final activity is performed in a particular order, such as "du", "ls", or "rm", the work list selection mechanism for each phase allows for selection of work by serial tree traversal order, rather than by the normal parallel directory order. Thus, parallel performance can still be obtained for the first one or two parallel phases, while still providing the desired serial order to the final phase of a user command. If the final phase can be performed as fast as work becomes available for it, then the overall command can execute a parallel speed, even though the last phase may be serial. For example, an "ls" command can execute at parallel "stat" rates, even though the final output might be in some specific serial tree order.

4.2. Resource-Based Scheduling

As threads execute various types of tasks, they consume several types of resources, including server memory, client memory, client CPU bandwidth, client disk bandwidth, and network bandwidth. When many threads are executed in parallel, they can quickly exhaust available resources on the server or on any of the available clients, causing either performance degradation or the crash of some component of the user interface. To keep parallel activities from consuming too many resources, each thread is controlled according to the amount of resources it uses, and is prioritized according to its execution phase, described above.

The overall scheduling goal is to approach saturation of the most limiting resources without oversubscribing them. To keep track of all of the resources being consumed at any given moment, a detailed resource estimate is maintained for each thread. This estimate is based upon a detailed description of the various resource components involved in various thread activities. For example, for each file transfer performed, configuration information is utilized that describes disk speed, CPU speed, network interface speed, overall network speed, archival device speed, and maximum archival devices allowed. For each thread that is obtaining file attributes, per-client memory consumption and CPU bandwidth are estimated. This performance information allows for very fast and reasonably accurate task dispatching, and for load leveling across client machines. (Interface processes are automatically started as necessary on available client machines for purposes of load-leveling).

5. Many Small Files

Large numbers of small files result in problems with both archival performance and with the amount of metadata that must be maintained within the archival system database. For example, HPSS can archive approximately 100 files per second. At this rate, nearly three hours are required to archive one million small files. In addition, since roughly 2000 bytes of metadata are required in the database for each file, one million files results in roughly 2 GB of database metadata that must be backed up, etc. Since our site has an estimated 5 billion files archived, these files would currently require an archival database of approximately 10 TB of data if all of these files were stored individually.

6. Small File Aggregation

To alleviate both performance and metadata problems, various forms of aggregation are often utilized by archival systems. While aggregation can be performed on the client or on the archive, client aggregation was chosen for use within PSI. Client-side aggregation facilitates scalability, leveraging client file system bandwidth while reducing the transaction load and data transfer load on the archival system (1000 client files/sec, 1000 bytes/client file => 1 archival file/sec, 1MB/sec).

6.1. HTAR

A "tar"-like utility named HTAR (from Gleicher Enterprises) is used to aggregate and de-aggregate files on the client. HTAR is multi-threaded, and generates a

standard "tar" file directly on HPSS, reducing the load on the local file system. HTAR archives an index file with each tar file. This allows the determination of what files are in the tar file without having to read the actual tar file. HTAR also supports the effective removal of individual files from a tar file by modifying the index file.

The main drawbacks of using HTAR (or any client-side aggregation approach) are over-aggregation and an invisible name space. Over-aggregation results from placing too many files and too much data into a single file, and therefore onto only a small number of archival devices. The invisible name space occurs simply because the files in the aggregate files are not a part of the archival system name space, and thus are not directly accessible via archival system API function calls.

6.2. Avoiding Over-aggregation

PSI addresses the issue of over-aggregation by aggregating (at most) one directory of files into each tar file, i.e. no subdirectories are placed within an aggregate. This approach allows many directories to be operated on (aggregated, queried, retrieved) simultaneously. For scalability, PSI supports breaking up a directory into more than one tar file, allowing single large directories to be operated on in parallel via multiple aggregates per directory. The advantages of aggregation by directory are shown in the performance results below. Another aspect of avoiding over-aggregation is that only small files are placed into aggregates; large files are stored as regular files, allowing for more bandwidth to be applied to a given user request.

6.3. Namespace Extension

Aggregation by directory also facilitates an extensible name space, which in turn addresses the issue of an invisible name space. PSI provides the capability to extend the name space of a given directory into the tar files located within that directory. This allows user interface commands, e.g. "ls", "chmod", "store", "cp", "get", etc to access the files within these tar files. As part of the name space extension, a user command is able to refer to files within these tar files by means of "globbing" (wild cards).

PSI is also responsible for maintenance of the contents of directories containing tar files, including removing regular files or files in tar files to avoid duplicate names in a directory name space. In addition, user interface commands can utilize aggregates intelligently, operating on whole tar files when executing commands such as "cp", "rm", "du", and "chmod", instead of operating on individual files within tar files, when appropriate, especially when operating on whole tree structures.

The result is an aggregation approach that is high speed, scalable, and provides the user with a reasonably transparent view of all files in a tree, whether they are regular files or are contained within an aggregate file. Also, the archival database is significantly decreased through the reduction of the number of regular files within the archival system.

7. Overview of Network Configuration

The HPSS archival storage system was running version 6.2 of HPSS on 12 AIX main servers and tape storage servers, including 32 StorageTek T10000 tape drives, attached in groups of 4 to Linux tape servers, each server connected to the network with a 10-GigE interface.

Each of the eight client nodes has four dual-core AMD Opteron processors @ 2.2 Ghz. running Redhat EL 4. The intra-client network fabric for the Linux clients is InfiniBand.

The client file system is a Panasas global parallel file system (version 3.0) with each user file configured across 7 to 10 storage blades.

8. Performance Results

In the following tests, whole trees were operated upon by a single user issuing a single command to the archival user interface. The "Threads" axis refers to the number of threads used in the dominant (limiting) work flow phase, described above. For example, in the "find" command, the dominant thread type is always the phase one "stat" threads. When transferring data, the dominat thread type is the phase three transfer thread. In the description of results below, the following terms are used.

client_tree	refers to a local file system tree of a million file tree of small regular files (1000 bytes per file, 1000 files per directory, 1000 directories per tree).
reg_tree	refers to a million file tree of small regular files (1000 bytes per file, 1000 files per directory, 1000 directories per tree).
htar_tree	refers to a million file tree of aggregated small regular files (1000 bytes per file, 1000 files per aggregate, one aggregate per

	directory, plus one aggregate index file (512,000 bytes) per directory, 1000 directories per tree.
large_file_tree	refers to a one terabyte tree of 61 files, each 16 gigabytes in size.
cond	refers to a conditional operation that is performed only if it is determined from the file attributes that the operation is necessary.

8.1. Results for Finding Files

This set of results involves the use of the "find" command. The UNIX "find" command and the PSI local "find" command were executed on the client file system, and the PSI archive "find" command was executed on archive trees of regular and HTAR files. This set of results primarily measures the performance of "stat" requests, since the per-file "find" logic is negligible.

The limiting factor in the client file system cases is the rate at which the client file system can provide "stat" information. The limiting factor in the archive reg_tree case is the HPSS DB2 query rate for files, including auxiliary metadata. The limiting factor in the archive htar_tree case is the number of file transfer requests (to read HTAR index files) per second that can be handled by a single API connection to HPSS.

The archival performance gain from multi-threading and aggregates was a factor of 99.

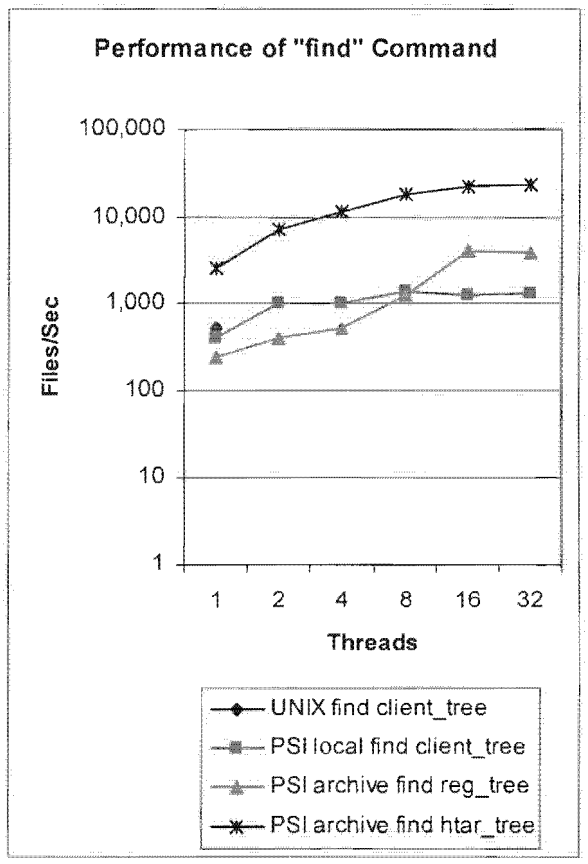

Figure 1. Find results

8.2. Results for Changing File Permissions

This set of results involves the use of the "chmod" command, and measures the performance of the "stat" and "chmod" operations. The UNIX "chmod" command and the PSI local "chmod' command were executed on the local file system. The PSI archive "chmod" command was executed on archive trees of regular files and HTAR files.

The conditional cases involve only changing file attributes when necessary. Also, note the performance advantage of modifying aggregate attributes instead of individual file attributes, as in the "htar_tree cond" tests, since only the HTAR file permissions required updating.

The archival performance gain from multi-threading and aggregates was a factor of 839. Adding the conditional "chmod" operation increased the gain to a factor of 4074.

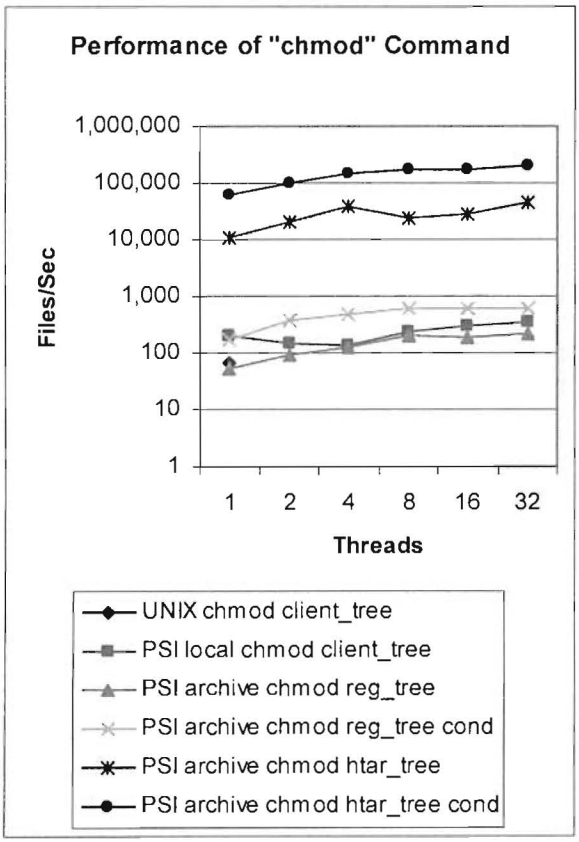

Figure 2. Chmod results

8.3. Results for Copying Small Files

The following tests demonstrate the performance of copying small files (1000 bytes each) from one tree to another tree within the same file system. The UNIX "cp" command and the PSI local "cp" command were executed on the client file system, and the PSI archive "cp" command was executed on archive trees of regular and HTAR files.

The limiting factor for the client executions was the rate at which files could be written to the local file system. These results illustrate the advantage of copying aggregated (HTAR) files instead of copying individual files. The limiting factor in these case is the rate at which aggregate files and index files can be copied within the archive.

The archival performance gain from multi-threading and aggregates was a factor of 2564.

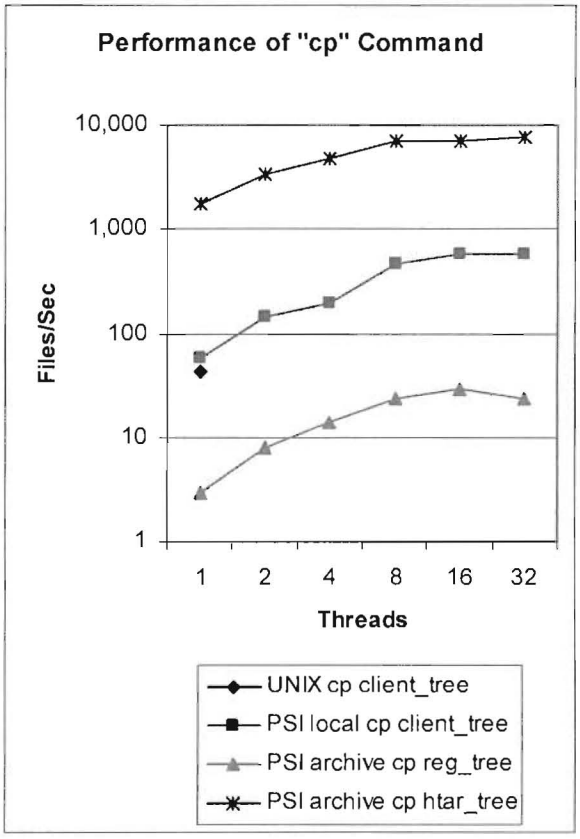

Figure 3. Cp performance

8.4. Results for Storing Small Files

The following tests demonstrate the performance of storing small files (1000 bytes) into the archival storage system. These tests stored files from the local file system into either regular or HTAR trees in the archive.

The apparent limit for the unconditional HTAR case is the rate at which the client file system could perform a sequence of "stat, open, read, close" system calls on small files for a single user. The apparent limit for the conditional transfer cases consists of the combination of the limit on "stat" of the local file system and the archival file system, and the rate at which these two sets of file attributes could be compared.

The archival performance gain from multi-threading and aggregates was a factor of 267. Adding the conditional "store" operation increased the gain to a factor of 1066.

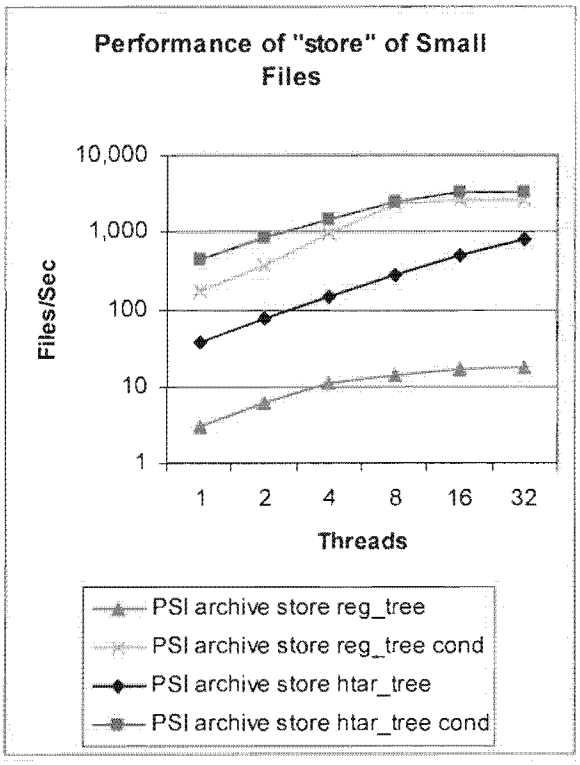

Figure 4. Small file store performance

8.5. Results for Storing Large Files

The large file tests demonstrate the fundamental ability to utilize multiple client nodes to transfer from the client file system to archival system tape drives, with multiple file transfers occurring on each node. In this test, the file transfer threads were spread "evenly" across 8 client nodes. The apparent limiting factor is the client file system performance.

The archival performance gain from multi-threading and multiple nodes was a factor of 5.

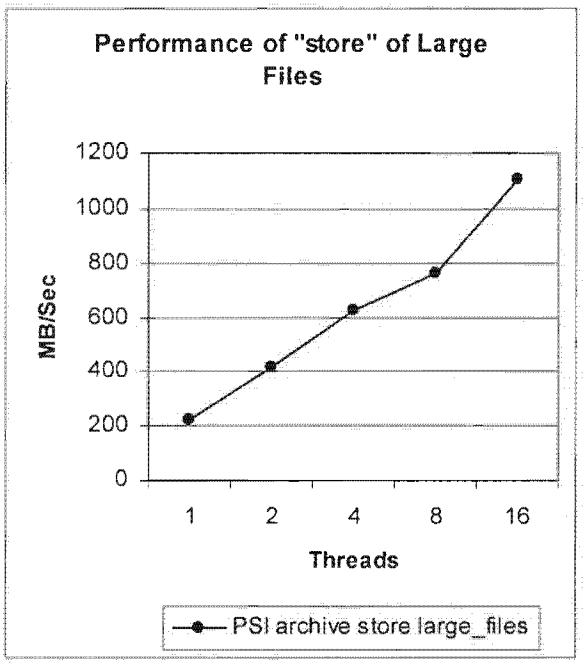

Figure 5. Store performance

8.6. Results for Grep of Archive Files

These tests ran the PSI archive "grep" command on the regular and HTAR trees in HPSS. The increase in performance when using aggregate (HTAR) files illustrates the value of utilizing the HTAR co-location of data and metadata, as seen by the nearly two orders of magnitude gain in speed. Virtually any data mining operation could feasibly make use of aggregates in this fashion to obtain significant performance improvement.

The archival performance gain from multi-threading and aggregates was a factor of 616.

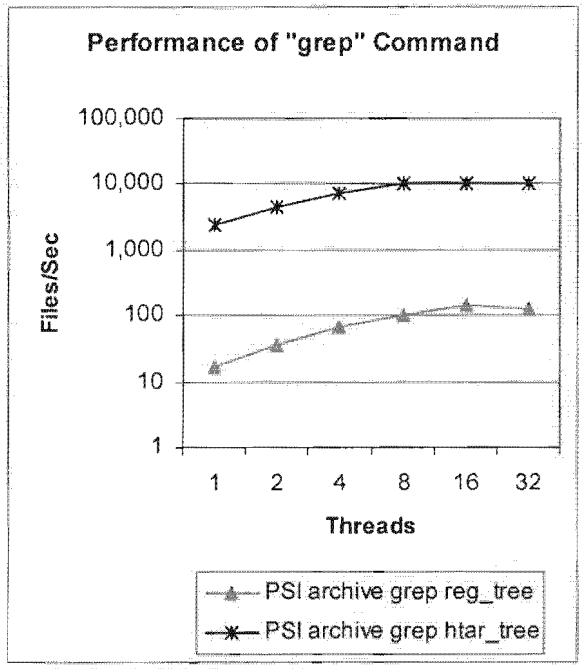

Figure 6. Grep performace

9. Conclusion

Combining the techniques of multi-threaded processing of data and metadata with the concept of small file aggregation can result in significant performance increases for archival storage systems. These increases can be further improved by adding techniques such as conditional updates or conditional file transfers. Performance increases above factors of 1000 have been observed. In addition, using user-generated aggregates can result in significant decreases in archival system metadata.

10. References

[1] E. Ong, E. Lusk, and W. Gropp, *Scalable Unix Commands For Parallel Processors: A High-Performance Implementation*, Y. Cotronis and J. Dongarra (Eds): Euro PVM/MPI 2001, LNCS 2131, pp 410-418 (Springer-Verlag, Berlin Heidelberg 2001)