Ruprecht-Karls-Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Bachelor's Thesis

Performance Analysis of the PVFS2 Persistency Layer

Name: Julian Martin Kunkel
Matrikelnummer: 2233273
Betreuer: Prof. Thomas Ludwig
Abgabe Datum: 15.02.06

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

.....

Datum: February 14, 2006

Acknowledgments

My grateful thanks to Prof. Dr. Thomas Ludwig for his guidance and support during the whole project period. I also wish to thank Stephan Krempel, who set up the GNU-arch repository and provided guides for arch and Dulip Withanage, who provided a latex template for the Bachelor's thesis. My thanks also go to the whole PVFS2-development team for their help in many occasions.

Last but not least, I want to appreciate the valuable support of my parents Ursula and Paul Kunkel and my sister Simone.

Abstract

In this thesis the persistency layer of the Parallel Virtual File System 2 (PVFS2) is analyzed in order to argue about the limitations in the architecture. Therefore, a new module is introduced that replaces the current module responsible for the persistency layer. This module operates in memory and provides an upper bound for the achievable performance of the persistency layer.

The basics of the persistency layer and the internal request processing are documented to help the reader to understand the servers' workflow. Some simple theoretical thoughts lead to estimated upper bounds for the performance of various operations. These estimated upper bounds are later compared with measured performance values.

Different benchmarks are run to determine the I/O and the metadata performance of several configurations. The measured data is evaluated and discussed in order to improve the implementation. Within the framework of the thesis some bottlenecks could already be found. It turns out that there is much room for further improvement in the metadata handling of PVFS2.

Contents

1 General Goals of the Thesis

Performance of complex software with a layered architecture is strongly limited by the capabilities of each single layer. An analysis of a single layer can be done by implementing a simple stub which has a well known complexity for the functions provided by the layer. Furthermore, the performance of the remaining layers and even the whole software architecture can be measured with an efficient stub to find bottlenecks.

As the Parallel Virtual Filesystem Version 2 (PVFS2) has such a layered architecture, an examination of the different layers can help to improve performance. The goal of this thesis is the analysis of the PVFS2 persistency layer [1]. In addition, concepts for further improvement of the layers are derived from this analysis.

The thesis is structured as follows: Chapter 2 gives an overview of PVFS2 and its environment. Chapter 3 describes the persistency layer and the workflow of common file system operations in detail. Some simple considerations lead to estimated upper bounds for the performance of the file system operations in chapter 4. Chapter 5 introduces and discusses the source code modifications made during this project, especially the new persistency modules. Chapter 6 describes the operating modes of the benchmark programs which are used to evaluate the performance of PVFS2 in chapter 7. Chapter 8 summarizes the proceeding and main results of the project. Possible future works are shown in chapter 9.

[1] In terms of PVFS2 the persistency storage layer is referred to as TROVE

2 System Overview

This chapter gives a general overview of the PVFS2 server and client components.

2.1 The Parallel Virtual File System PVFS2

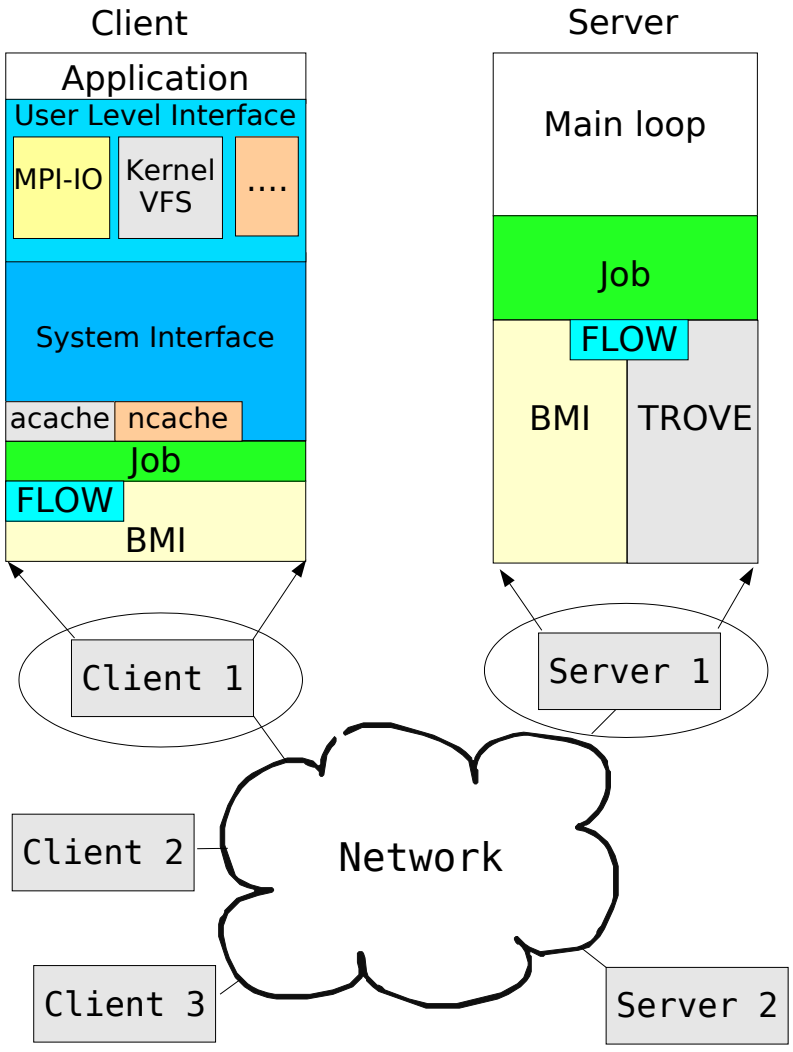

Figure 2.1: PVFS2 software architecture

Large computer clusters are now widely used for scientific computation. In these clusters, I/O subsystems consist of many disks located in many different nodes. The software that organizes these disks into a coherent file system is called a "parallel file system". Using the parallel file system software, applications can access files that are physically distributed among different nodes in the cluster.

The Parallel Virtual File System PVFS2 is one of the popular open source parallel file systems

developed for efficient reading and writing of large amounts of data across the nodes. To achieve this, PVFS2 is designed as a client-server architecture as shown in figure 2.1.

In the context of the file system the following terms are important:

Collection: A PVFS2 collection corresponds to a logical file system which is distributed among the available PVFS2 servers.

PVFS2 server A PVFS2 server holds exactly one so-called Storage-space, which may contain a part of several collections. According to the type of storage provided for a file system, servers can be categorized into data servers and metadata servers. data servers store data in a round robin manner, typically striped over multiple nodes using the UNIX file system. Metadata servers store object **attributes**. This is all the information about files in the UNIX sense, i.e. object type, ownership, permissions, timestamps and filesize. Additional information like extended attributes and the directory hierarchy, is stored on metadata servers, too. A compute node can be configured as either a metadata server, a data server, or both at once.

PVFS2 client Clients are nodes that access the virtual file systems provided by the PVFS2 servers. Applications can use one of the available userlevel-Interfaces to interact with the file system.

Filesystem objects Objects which can be stored in a PVFS2 file system are files, directories and symbolic links. Internally PVFS2 knows additional system level objects: metafiles, which contain metadata for a file system object, datafiles which contain a part of a UNIX file data and directory data objects, which store the mapping of a filename to a handle. PVFS2 stores a file system object as one or multiple system level objects.
As a concrete example a logical file is stored as a metafile and the file data is split into one or more datafiles, which can be distributed over multiple or even all available data servers. The metadatafile containing the object's attributes and other metadata for a single PVFS2 file system object is located on exactly one of the available metadata servers. The internal file system organization is further described in section 3.1.1.

handle A handle is a number identifying a specific internal object and is similar to the ext2 inode number.

2.2 Software Architecture

PVFS2 uses the layer model illustrated in figure 2.1. Interfaces for the layers use a non-blocking semantics. Desired operations are first posted and then their completion status is tested. A unique identifier is created for every post, which is used as an input for all test calls. In case the operation is not very complex and time consuming, it is possible that the operation immediately completes during the post call. This has to be checked by the caller. For a more detailed description see [8].

2.2.1 Userlevel-interface

The userlevel-interface provides a higher abstraction to the PVFS2 file systems. There are currently two userlevel-interfaces available: a kernel interface and an MPI interface. The kernel interface is realized by a kernel-module integrating PVFS into the kernel Virtual Filesystem Switch (VFS) and a user-space daemon which does communicate with the servers. PVFS2 is specially designed to provide an efficient integration into any implementation of the Message-Passing Interface specification MPI-2,

which is an interface standard for high performance computing. An ADIO device for ROMIO enables MPICH2[1] to access PVFS2 file systems.

2.2.2 System interface

The system interface API provides functions for the direct manipulation of file system objects and hides internal details from the user. Applications can use the `libpvfs` functions to access the PVFS2 file systems. A program using this interface is considered as client. Invoking a system interface function starts a statemachine which processes the operation in small steps.

Statemachines In the PVFS2 context, statemachines run a specified function in each of their states. The return value of this function determines which transition of the state should be taken. Modeling of a complex request is possible, therefore a complex operation is represented as a sequence of several states. State functions require two qualities: they should need little time and must not influence the results of different statemachines. Using time division multiplexing, statemachines enable a degree of parallelism, simply running the active state's functions for each statemachine in a round robin fashion. PVFS2 statemachines can be nested to model and simplify the handling of common subprocesses. Statemachines are used on clients and on servers.

Client-side caches Several caches are part of the system interface and try to minimize the number of requests to the server processes. The attribute cache (acache) manages metadata, like timestamps and handle number.
The name cache (ncache) stores a file system object's filename and related handle number. To prevent the caches from storing invalid information, data is set invalid when a defined time is over [2] or when the server signals the client that the object does not exist.

2.2.3 Job

The job layer consolidates the lower layers BMI, Flow and Trove into one interface. It also maintains threads and callback functions, which will be given as input to called functions. On completion of an operation the lower layers can simply run the callback function, which knows the next working step necessary to finish the operation. This can be used in the persistency layer, for example, to initiate a transmission when data was read. Furthermore, a request scheduler is part of the Job layer, which manages complex client requests as statemachines.

2.2.4 Flow

A flow is a data stream between two endpoints. An endpoint is one of memory, BMI or Trove. The user can choose between different flow protocols defining the behavior of the data transmission. For example, buffer strategy and number of parallel transferred messages may be different for two flow protocols. To initiate a data transfer, flow has to know the data definition (size, position in memory, ...) and the endpoints. Flow then takes care of the data transmission. Complex memory and file datatypes are automatically converted to a simpler data format, convenient for the lower level I/O interfaces.

[1] MPI Chameleon 2, an MPI-2 implementation
[2] Timeout does correspond to the storage hint `HandleRecycleTimeoutSecs`

2.2.5 BMI

The Buffered Message Interface provides a network independent interface. Clients communicate with the servers by using the request protocol, which defines the layout of the messages for every operation. BMI can use different communication methods, currently TCP, Myricom's GM and Infiniband. Similar to MPI, BMI requires to announce the receiving of a message before the message is expected to arrive. Any announcement includes the sender of the message, expected size of the message and identification tag. The transmitted message fits into the buffer, so no memory copy is necessary. This improves overall performance, which is only one of the incorporated optimizations.

Sometimes it is not possible to know the origin of the message. Then, it is called unexpected message. A client starts a file system operation by sending an operation specific request message to the server. However, the server cannot know that there will be a request or cannot even be aware of the client's existence. So the server buffers a pool of unexpected messages, which have the maximum initial request size possible.

2.2.6 Trove

Trove provides and administrates the persistent storage space for system level objects. Data is either stored as a keyword/value pair or as a bytestream. Keyword value pairs are used to store arbitrary metadata information while bytestreams store a logical file's data. Bytestream data is accessed using a size from an offset, while keyval data can be accessed by resolving the key.

Like BMI and Flow, Trove can switch between different methods, which are actually different implementations of the whole interface.

2.2.7 Server main loop

The server's main loop checks the completion of a statemachine processed by threads. If a statemachine's current state is finished, the next state is assigned for work and the state function is called. This either completes the current states operation or enqueues the operation for the threads. In case the function does complete immediately, the next state of the statemachine is called directly. There is a BMI and a Trove thread, which takes care of the unfinished operations depending on the operations' type.

In addition, unexpected messages from BMI are decoded. For each message the appropriate statemachine is started and a buffer for a new unexpected message is provided.

Summary: This chapter describes the components of the PVFS2 server and client. Furthermore, it outlines the tasks of the different layers, due to the access to the file system.
In the following the PVFS2 persistency layer Trove is reviewed more detailed.

3 An In-Depth Look at Trove

Documentation about the internals of Trove are rare - hence it is my concern in this section to document internal file system representation and processing of several common requests. This is important to understand to estimate PVFS2's performance.

3.1 System Level File Ssystem Representation

This section shows how system level objects form a logical file system - remember the term collection is used in the PVFS context. Filesystem objects are internally represented by using one or multiple system level objects, which are maintained by Trove. A server can maintain multiple collections. Therefore, a collection has a unique id, the collection id and a name which is mapped into the id by a server. In order to access a collection a client needs a configuration file which has a similar syntax than the `/etc/fstab`. This so-called pvfs2tab which can be incorporated into the common `fstab` specifies for each file system a responsible server and collection name. In case the linux kernel module is used the file system can be mounted on the defined mount point. If the application uses the MPI interface or directly the system interface, the defined mount point is virtual. An access to a logical file system object within a virtual path specified in the `pvfs2tab` is mapped to the corresponding collection.

Each system level object stores either keyval or bytestream data and owns common attributes, i.e. timestamps, object type, collection id, handle, keyval count and bytestream size. A stored object is called **dataspace** and has a unique identifier within a collection, the handle. The handle together with the collection id form the **object reference** which is used to identify the object. On the higher level system interface, the object reference of a file system object consists of the object's file system id and its metadatafile handle.

Internally, the client decides which server is responsible for an object, by inspecting the object's reference. Within a file system every server is responsible for a disjunct handle range. The ranges are set in the file system configuration. Therefore, the client can map the collection id and handle number directly to the entity's responsible server. The available handle numbers (8 Byte) are also shared between meta- and data servers.

Filesystem objects and internal representation:

- **Directory**
  attributes are stored in a directory object, while a separate directory data (dirdata) object holds the directory entries. The value of a directory's dir_ent key is the handle of the corresponding dirdata object.

- **File**
  A logical file is stored as a metafile, and the file's data is split into one or several datafiles which can be distributed over multiple or even all available data servers. How the data is split

over the data servers is controlled by the file's distribution function, for example splitting the file in 64KByte chunks and placing them on the data servers in a round robin fashion. Other distribution functions are possible and can be set for a file during creation. Information about the distribution and datafile handles is held by the metadatafile as value of the keys meta_dist and datafile_handles, which is an array of handles. Datafiles are the only objects located on a data server, other objects are kept on a metadata server.

- **Symbolic link**
  A symbolic link is similar to the UNIX pendant and internally represented by one object. Full access to the link object is allowed for everybody and cannot be changed. Instead of checking the links permissions, the permissions of the referenced object are checked. This is the same behavior as in UNIX. The link target's name is stored as a keyval pair.

Additional extended attributes can be stored as keyval pairs in a files metadata and in the directory object for directories. The PVFS2 linux VFS interface, for example, maintains access control lists using this mechanism.

The data distribution of a logical file The selected distribution function for a file defines the way data is distributed among the different datafiles which are located on different servers typically. In this paper the default distribution simple stripe is used with a block size of 64 KByte. Simple stripe divides the logical file data into chunks with the length of the block size. These chunks then are mapped into the datafiles in a round robin manner, which means chunk 1 is mapped into the first datafile, chunk 2 into the second datafile and so on, until one chunk is mapped to each datafile. The next chunk then gets assigned to the first datafile. This process continues until all chunks are assigned. A mapping of a logical file with a size of 416 KByte into 5 datafiles is shown in figure 3.1. If the logical file grows slowly, then, at first, datafile 2 is enlarged up to 128 KByte before datafile 3 is increased.

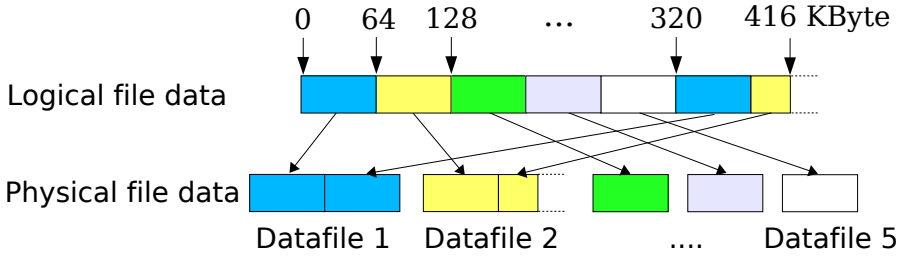

Figure 3.1: File distribution for 5 datafiles using the default distribution function which stripes data over the datafiles in 64 KByte chunks in a round robin fashion

Additional collection attributes There are several collection hints, which can be set in the file system configuration file. Most options are given to Trove while starting the server, before requests are dealt and cannot be modified during runtime. Configuration parameters can be used to control internal cache mechanisms, i.e. keyval pairs which could be cached and a server side attribute cache. Attributes are currently cached in an array of lists by hashing the object reference to the number of a linked list. The size of the hash array and the maximum number of elements of the attribute cache can be set.

Also, there is a synchronous mode for meta and filedata, which forces data to be written to disk during a modifying operation. The server sends the success of an operation after the data is written,

otherwise the server may acknowledge before data is saved and the operation can be deferred to a later time. Sync modes can be set during runtime using the user-space tool `pvfs2-set-sync`.

An important parameter is the collection handle range, which sets the handle range for metadata and datafiles if the current server should maintain both. Otherwise only the range is set, which the server should take care of.

There is also a handle timeout. This sets the time necessary to pass before a handle can be reused, after the object has been deleted. This setting is transferred to the client during client initialization and sets cache timeouts to avoid operating on an invalid (non-existent) object.

3.1.1 Example collection

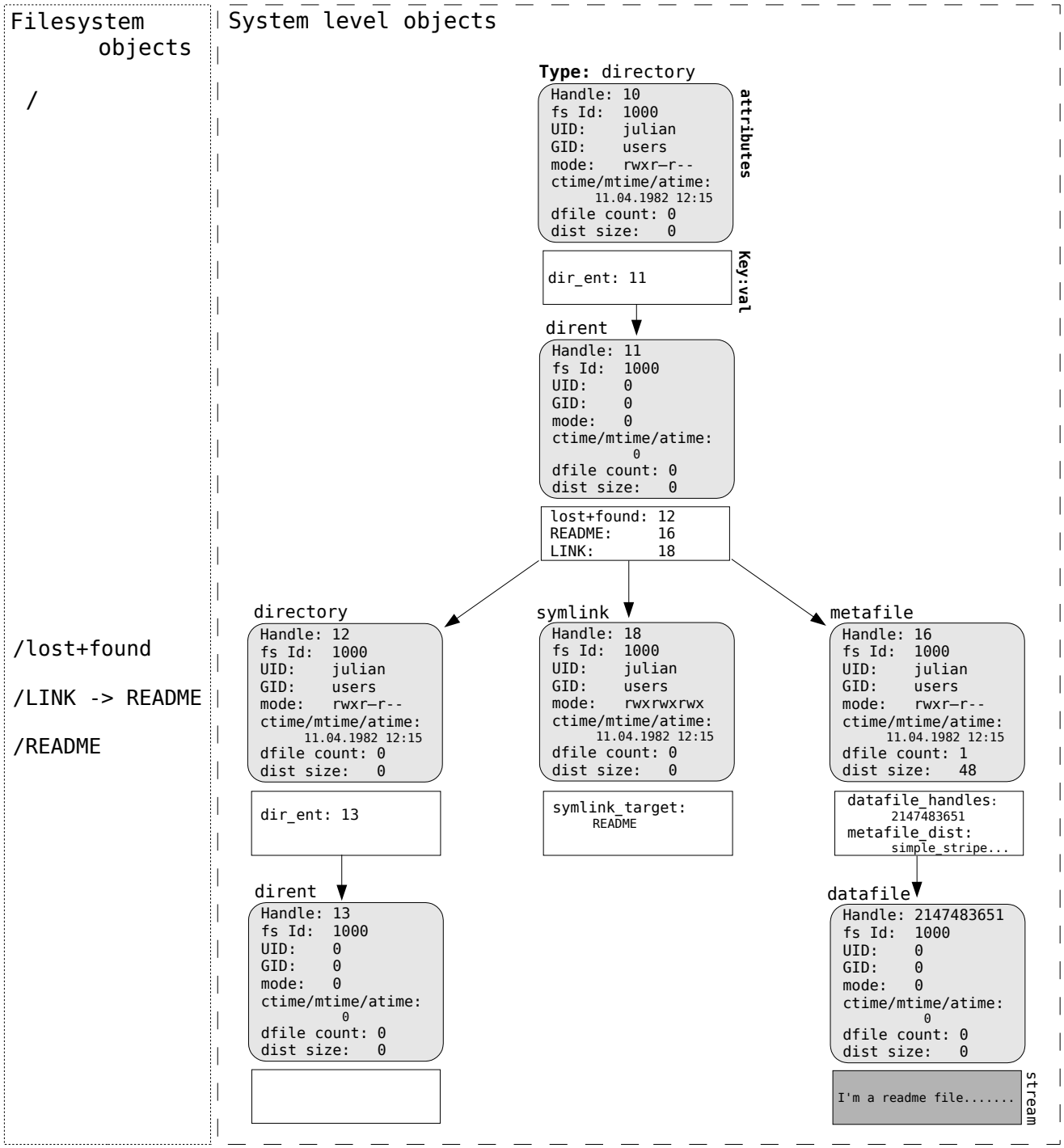

Figure 3.2: Example collection

On the left you can see some logical PVFS2 file system objects, accessible through the system interface and on the right the internal representation and mapping into low-level objects. Let us assume 1111 as collection id and `MYCOLL` as collection name. The example collection uses the range 4 to 2147483650 for metadata handles and 2147483651 to 4294967297 for datafiles. Depending on the configuration, objects can be located on different servers or on the same server. A higher dfile_count for the README file's metafile would have created more datafiles, which could be spreaded over miscellaneous servers to increase parallelism.
It is important to distinguish between the collection and its persistent representation on the par-

ticipating servers, which depends on the selected Trove module. As an illustrating example for the current implementation, the directory /tmp/pvfs2-server is displayed in figure 3.3. In this example the directory keeps the persistent representation of the system level objects of one server. Each file system object has a dedicated file for bytestreams and keyval pairs. Note that the names of the files and directories are based on the handle number. As an example the directory with the number 00000457 is the hexadecimal number of 1111. This is the collection id. A detailed discussion of the mapping is spared in this paper.

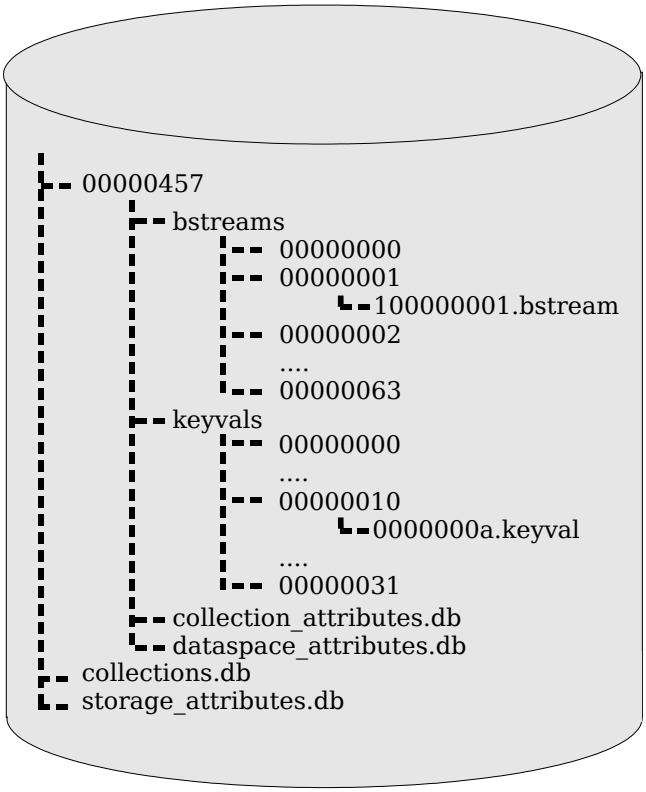

Figure 3.3: Example directory storing a part of a collection

The following `pvfs2tab` file specifies that the (virtual) mount point /pvfs2 is used to access the collection MYCOLL. The machine master2 runs a PVFS2 server participating on a collection with the name MYCOLL and the server is configured for TCP on the port 3334. A client requests from this server the vital file system configuration.

Example pvfs2tab:

```
tcp://master2:3334/MYCOLL    /pvfs2    pvfs2    default,noauto    0 0
```

3.2 Interface

The functions provided by the Trove-interface can be grouped into the following categories:

- Trove management
  Initialization and finalization of Trove.

- Storage management
  Creation and removal of the storage space.

- Collection management
  There are functions to create or remove a collection, to lookup the collection id by the name of the collection, iterate through all existent collections, get/set additional configuration parameters for a collection (i.e. cache size and handle range) and getting or setting extended attributes.

- Dataspace management
  Dataspaces can be created and removed. The existence and type of an object can be verified. It can be iterated over all existent handles and handle attributes (metadata) can be get/set. Also, there are some completion test functions to test for one, for multiple or for all unfinished jobs within a context and a function to cancel posted operations.

- Context functions
  Create or delete a context which restricts a test function to operate only on the unfinished operations of a context.

- Key/value access
  Possible operations are: to read/write one or a list of key/value pairs assigned to an object, the removal of a pair, to iterate over the key/value pairs or only the keys, or to flush the modified key/value pairs to disk.

- Bytestream access
  Data can be read/written at a position or multiple positions at once, a bytestream can be resized and the bytestream's data can be flushed to disk.

Most Trove operations take the collection id and handle number as input parameters to identify a system level file object. See [10] for detailed explanation of various functions.

3.3 Modularity

Trove is designed to support multiple methods, which are actually implementations of the interface. Most trove functions are wrappers, which determine the method to use calling the function map_coll_id_to_method, which maps the collection id to the method id. There are only a few management functions that do additional work. The correct function can be resolved from method tables by knowing the method id. Method tables store function pointers to the method's functions. Multiple methods are not fully supported. There is only one implementation yet, called Database plus File (DBPF). DBPF uses multiple Berkeley databases to store keyval data and collection information. Normal UNIX files are used to store bytestream data. By now the interface intends the selection of only one active method handling all the requests. The selection is made during the trove initialization

on server startup. The mapping function is only a dummy function returning zero all the time, which is the DBPF's method id.

3.4 Explanation of Some Client Operations

These explanations point out the sequence of operations triggered by calling a specific system interface function. Also, the client-server communication pattern and the called functions can be used to analyze the theoretic operation throughput. The illustrated requests reflect the state of the source code. PVFS2 is designed to retain consistency of the file system in case that a client crashes during an operation. Therefore, underlying sub operations of more complex requests are processed in a special order.

Normally, a system interface functions starts a request-specific statemachine on the client. Until the request is completed the system interface blocks. During the processing of the statemachine, several states communicate with the pvfs2-servers by using the request protocol. Message data is normally filled by a macro. Every message contains the current operation number, user credentials (user and group id) and parameters depending on the operation.

Messages sent to a server initiate a statemachine, corresponding to the operation number. The message content gets assigned to the statemachine. This allows each state to access the message data needed.

Many operations start the nested statemachine `prelude`, which posts the job to the request scheduler and checks the permissions depending on the operation type. If the request is processed, the nested statemachine `final_response` creates the response message, fills in the return code and sends it back to the client. This process is not mentioned explicitly.

The next subsections show the processing of some requests. This includes the required input parameters, client and server operations triggered by the request. Input and output of the operations are given when needed to illustrate the data flow. Client states are labeled with a C and the state number. Server statemachines are shown if a client state starts a server operation. Some actions done by the layers are mentioned, especially trove operations. If there are functions mentioned which are invoked during the request, they will normally have the prefix `job_trove_` like `job_trove_keyval_read`. These functions are part of the Job layer and invoke the Trove operation suggested by the name. Sometimes processing will be simplified, states will be merged together or not shown when they are not instructional for the reader. A reader eager to know the detailed internal steps should look at the source code.
Statemachines are referred to by their name. The naming scheme is straightforward and is indicated for a few requests.

3.4.1 Get the attributes of a file system object

This request is processed by the statemachine pvfs2_client_getattr_sm. It may be invoked by the system interface function `PVFS_sys_getattr` or as a part of some other complex functions.
Input: credentials, **Object reference** and **attribute mask** (specifies the requested metadata information of the object)

C1 Check whether the object reference is in the acache and the bytestream size of the object is not requested. If that is the case, return the acache data and the request is finished.

C2 Request from the metadata server the object's attributes, specified in the attribute mask, by sending a request of the type PVFS_SERV_GETATTR

S1 A PVFS_SERV_GETATTR request starts a get_attr statemachine. This starts the nested prelude sm, which does the next two steps.

S2 Get the object's attributes using `dspace_getattr`

S3 Permission check (for getattr access is always allowed)

S4 Start a nested get_attr_work statemachine

S5 If the object is a symlink, its target is read using `keyval_read`.

S6 In case the object is a metafile, read the datafile handles and the distribution with `keyval_read`

S7 If the object is a directory, read the handle of the dirdata object with `keyval_read` and the key number from the dirdata object with `dspace_getattr`. Note: For a directory data object the key-size does correspond to the number of directory entries.

C3 Store attributes in the acache

C4 If the logical size of a file is not required, we are done. Otherwise request the datafile size from each data server holding a part of the file, by sending a PVFS_SERV_GETATTR message with PVFS_ATTR_DATA_SIZE as a parameter. With the help of the distribution function the logical size of the file can be calculated. This step consists actually of several states processed by another statemachine. Note: Determination of the filesize is expensive, but the requests to multiple data servers are processed in parallel.

S1 Every data server runs the same machine as invoked in step C2. To determine the size of a datafile the `fstat` system call is invoked.

3.4.2 Create a file

System interface function: PVFS_sys_create
Input: credentials, **parent directory reference**, **object name**, **attributes**, **distribution** (can be set, otherwise default distribution is used. The distribution has to be resolved using the distributions name)

C1 Get the parent directory's attributes

C2 Inspection. Set the object's group id (GID) if the parent directory has SetGID mode enabled

C3 Send an object creation request (PVFS_SERV_CREATE) with the metadata handle range to a random metaserver to initiate creation of a metafile

S1 Start prelude sm to insert the object into the request scheduler

S2 Create a metafile, which has it's handle within the metadata handle range, by using dspace_create,

C4 Request the creation of the datafiles (PVFS_SERV_CREATE). Therefore start with a random data server number. Send a create request to each data server with the collection's datafile handle range. All create requests are processed parallelly.

S1 Start prelude sm to insert the object into the request scheduler

S2 Create a datafile with dspace_create and use the datafile handle range as a parameter

C5 Set the metafile attributes

S1 Start the prelude statemachine to fetch attributes and verify permissions.

S2 Write the datafile_handle keyval pair in conjunction with the array of created datafiles

S3 Add distribution keyval

S4 Set the attributes using dspace_setattr

C6 Insert the attributes into the acache

C7 Initiate creation of a directory entry, which points to the metafile's handle (PVFS_SERV_CRDIRENT)

S1 Read the value for the dir_ent key. If the dir_ent handle does not exist, create one (This does not happen in the current implementation).

S2 Try to add the new directory to the dirdata object's keyval pairs. If there is already an entry, abort

S3 Update the timestamps of the parent object using dspace_setattr

S4 Check whether there exists a key for the new directory. If not, abort

A request is not successful if the directory entry could not be added to the parent directory's dirdata object, or when it already exists. In this case the client tries to remove the created objects. Client failure or crash before step C4 may leave unusable objects. However, the objects are not visible for the user, thus consistency of the file system is guaranteed. Inaccessible objects can be detected during a file system check. The number of datafiles is limited to 1024 (PVFS_REQ_LIMIT_DFILE_COUNT).

3.4.3 Create a directory

The system interface function `PVFS_sys_mkdir` starts the statemachine pvfs2_client_mkdir_sm.
Input: credentials, **parent directory object reference**, **directory name**, **attributes**

C1 Fetch the parent directory's attributes (see 3.4.1)

C2 Inspect attributes. This step modifies the GID of the object if the parent directory's SetGID mode is enabled

C3 Choose a random metadata server and send a PVFS_SERV_MKDIR request with the input parameters and the collections metadata handle range

S1 Create the directory object with dspace_create, which chooses a free handle within the metadata handle range

S2 Set the attributes with dspace_setattr

S3 Create the dirdata object

S4 Add the dir_ent keyval entry to the directory

C4 Request addition of the directory entry to the parent directory (operation type PVFS_SERV_CRDIRENT)

S1 This request is the same as for the file creation, see 3.4.2 step C4.

The notes for directory creation are the same as for file creation.

3.4.4 Read content of a directory

Multiple directory entries are read by calling `PVFS_sys_readdir`. This operation returns the directory mtime as a version to figure out if a modification was made to the directory while the content was read. Clients can compare the returned versions while iterating over the directory content. If the version is different, another client has manipulated the directory. For example, a client application can decide to re-read the directory to ensure that all entries were read.

Input: credentials, **directory reference**, **read position** (number of already read directory entries), **incount** (signals the maximum amount of entries, which can be processed per function call)

C1 Get the directory attributes

C2 Request the directory read (PVFS_SERV_READDIR)

S1 Start prelude sm to fetch object's attributes

S2 Verify that the object is a directory and set the response directory version

S3 Read dirdata handle

S4 Use the function keyval_iterate to get the requested number of entries and their corresponding handles

S5 Set the directory version and send the response

C3 Put the received handles in conjunction with their names in the ncache

3.4.5 File open/get object reference

PVFS2 does not explicitly open and close a file. To initiate I/O a client only needs the object reference. The reference can be acquired with `PVFS_sys_lookup (absolute filename)` or `PVFS_sys_ref_lookup (relative filename)`. Starting with the root-handle, clients try to resolve the reference of the remaining path component from the responsible metadata server. The root handle

is part of the file system configuration, which is requested by every client on initialization. Therefore, it is known by every client.

Metadata servers are capable to resolve multiple path components at once. This happens when the server is responsible for the directory handles of a subsequent path. However, if a server does not maintain a handle, the client has to request the path resolving from the metadata server responsible for the next unresolved path component. Symbolic links are resolved by the client too. Lookup can be expensive if every directory within the path is owned by a different metadata server. Thus a server request for every directory is required.

C1 In case the ncache holds the filename we are done, else proceed

C2 Lookup the remaining path in the ncache. If it does not exist, request the lookup of the remaining segments:

S1 Start prelude sm to fetch attributes of the current unresolved path segment

S2 Allocate memory to store handles and attributes of all resolved segments during the lookup

S3 Get the attributes of the current path segment

S4 If the object is not a directory, return all resolved segments and attributes to the client and cleanup, else continue

S5 The directories permissions are checked against the credentials

S6 Read the dirdata object handle

S7 Get the next segments handle in case the directory entry exists

S8 Try to lookup the next segment in, go to step S3. If all segments are resolved, send the response to the client

C3 Memorize all resolved handles and attributes

C4 If the last resolved segment is a symlink, we request the attributes to get the target. In case the attributes are not available, this indicates we have to ask a different metadata server.

C5 In case we did not finish resolving, go to C2

C6 Insert the filename with the parent into the ncache

Notes: The resolved attributes in step C3 could be inserted into the acache. Furthermore, sub-paths could be inserted into the ncache to improve lookup for multiple objects within a common directory.

3.4.6 Delete a file system object

System interface call: PVFS_sys_remove
Input: credentials, object name, parent reference

C1 Request object removal from the parent directory

S1 Start prelude sm to fetch directory's attributes and check permissions

S2 Read dir_ent value from directory object

S3 Get the handle of the object to remove from the dirdata object

S4 Remove the keyval using keyval_remove

S5 Update parent directory's timestamps especially mtime

S6 Return the removed object's handle to the client

C2 Get object attributes

C3 Check object type. Depending on the type do the following:

- metafile: remove datafiles parallelly using PVFS_SERV_REMOVE requests

  S1 start prelude sm

  S2 remove datafile dspace

- directory: if the directory is not empty create the directory entry again and abort. Note: it is not allowed to remove a non-empty directory.

C4 Remove the object with a PVFS_SERV_REMOVE request

S1 Start the pelude sm

S2 Check object type. In case the object is a metafile or symlink simply remove the object with dspace_remove and the work is done, else we have a directory and proceed.

S3 Read dir_ent value

S4 Read dirdata attributes

S5 If the dirdata object has keys inside, we abort the removal because the directory is not empty. The client then creates the object's directory entry again

S6 Remove the directory

Notes: Client failures during a directory removal after step C1 and before the processing of C3 result in a directory which is not accessible. Therefore, all the objects inside are lost. A better way to delete an object would be to check the directory state first. This should be done to prevent non-empty directories to be inaccessible. After the creation, check it again to handle races between multiple clients.

3.4.7 Flushing of a file

The `PVFS_sys_flush` call forces the server to synchronize metadata and bytestreams belonging to a file with the persistent storage.

Input: credentials, object reference

C1 Get the attributes. This includes distribution and handles of the datafiles

C2 Send a parallel request to the metadata server, responsible for the object's metafile, and all data servers holding file's data.

S1 In case the object is a metafile run keyval_flush, else invoke bstream_flush

3.4.8 Do I/O

Depending on the access pattern, an I/O operation will usually only require a subset of the datafiles. The server's I/O statemachine is interlocked with the client's I/O statemachine. There is a performance counter that can be used to analyze the bandwidth of transferred data within a period. For example, by the karma tool. The value of the performance counter is processed and stored by a different state machine.

This operation can be triggered trough PVFS_sys_io
Input: credentials, file reference, request (can be a simple or complex data type), **offset, buffer, memory request, IO type** (read or write)

C1 Get the attributes (see 3.4.1)

C2 Find the datafiles needed for the I/O using the distribution function

C3 Send a message to every data server holding a datafile participating during IO to initiate flow

S1 Run prelude sm

S2 Send a positive acknowledgement to the client, if the object exists, else a negative

C4 Start a flow to a server if we get a positive acknowledge, else abort

S3 setup a flow calling the function job_flow which does the following:
Post the flow, probe for the flow protocol which does handle the specified transfer type and call flowproto_post for that protocol.
In our case the protocol is flowproto-mulitqueue, which initializes several buffers (currently 8). It makes a setup depending on the flow's endpoints and runs the appropriate callback function to start the flow.
Now a flow will be established between client and server, transferring a maximum of 256KBytes data per message.
Depending on the endpoints two callback functions are selected, one for Trove, responsible for reading or writing, and one for BMI, sending or receiving the data. In case a function's operation is finished, it invokes the partner function until the I/O request ends.

Concrete processing:
For a write, the source is BMI and the target is Trove. The callback trove_write_callback_fn announces the client's connection to BMI. It is called again when a trove_write is done and updates the performance counter. When BMI receives data, bmi_recv_callback_fn is called, which calls trove_bstream_write_list and the process starts from the beginning. Currently only one buffer is used at a time.

A read operation has Trove as source and BMI as target endpoint. Here for every buffer bmi_send_callback_fn is called, which initiates a communication and updates the performance counter. This also runs trove_bstream_read_list to read data. The trove_read_callback_fn is executed when a trove read has completed. It starts a BMI send operation for the data read. When the data is sent the BMI send callback is invoked again until all data is transferred.

S4 If the flow completes and we did a write operation, acknowledge the success to the client.

C5 The client sticks in this state until the transmission has completed or a transfer error occurred during the flow. Then we try do to the unprocessed I/O again: go to C3

C6 Analyze the success of the I/O operation and the amount of data transferred by using the distribution function.

C7 For a read operation it can be necessary to get the sizes of all datafiles to detect the correct file size read. This happens when a hole is within the requested file area. Hole means our read request touches a logical offset which is not within a datafile but within the file. It has to be detected if the read request is beyond the end of the file or does hit a hole. Logical filesize can be calculated using the size of each datafile and the distribution function.
Imagine to stripe data in 50 KByte blocks between two datafiles. One datafile is 100 KByte, holding the file data between 0-50 KByte and 100-150 KByte. The other is currently 1 KByte, storing the logical data of 50-100 KByte. If you try to read 2 KByte from the offset 50 KByte, you hit only the second datafile, which contains 1 KByte data by now. In order to determine that you have read 2 KByte data (with a lot of zeros) you have to verify that the first datafile stores data beyond the logical offset of 52 KByte.

Permissions are currently not checked during I/O. The data server does not know the file's permissions because they are only stored in the metafile.

There were recent changes of the I/O handling for small requests. In case the data can be piggybacked to the I/O initiating unexpected message for writes, or to the servers response for reads, this is done (this type of request is called eager read or write). Therefore, flows are not necessary for small requests. The maximum size of the unexpected message depends on the BMI method used, which is 16 KByte for TCP. This modification was made only recently, so is not considered in this paper. However, due to the reduced overhead, this should improve small IO requests significantly.

3.4.9 Test the availability of a file system

The availability test has a special role. It is not triggered trough a system interface call, but instead a sequence of operations executed by the user-space tool `pvfs2-ping`. The test is mentioned to show some management functions and the current way of file system verification.

C1 Check the `pvfs2tab` configuration file and request the file system configuration from the server mentioned in the pvfs2tab.

C2 Send a noop request (PVFS_SERV_MGMT_NOOP) to every server in order to check the server's availability from the client

S1 Responds

C3 Send a PVFS_SERV_PARAM_FSID_CHECK message to verify that all servers responsible for the file system know the file system id.

S1 Start setparam sm

S2 Use trove_collection_iterate to get the names of collections and their corresponding id. Verify that a collection exists with that id.

C4 Verify that root handle is owned by exactly one server. Therefore, send a management setparam message to all servers with the root-handle as parameter and check the result

S1 Start setparam sm

S2 Run dspace_verify(root handle)

There are a few more important operations: handling of extended attributes and renaming of objects. Also, there are several more management operations, but these operations are not discussed in this paper.

Summary: In this chapter details about the persistency layer Trove are given. The internal representation of logical objects is pointed out and common requests accessing these objects are illustrated. In particular, triggered network communication and I/O operations are mentioned. The knowledge about the behavior is used in the next chapter to estimate the expected performance.
In the next chapter some theoretical reflections are made. The theoretic results are compared with practical experience later.

4 Theoretic Performance Evaluation

In this chapter some simple considerations lead to estimated upper bounds for the aggregated throughput, which is the sum of all connected clients' throughput. Estimated upper bounds are determined for metadata operations, small contiguous I/O requests and large contiguous I/O requests. Therefore, the influence of the I/O subsystem, network and CPU are discussed. These resources limit strongly the performance of a parallel file system. However, it is not possible to get the resources' concrete performance behavior. As an illustrating example the performance of the I/O subsystem varies depending on the access pattern and position of the hard disk's head. As a consequence, average values are used for the performance of the hardware to determine a non-strict upper bound. The term estimated upper bound is used for this bound in order to clarify that we expect it to limit the throughput of the parallel file system and that it may be surpassed by practical results. The term throughput is used for either the I/O throughput or for the number of operations of a specific type which can be done per second (operation throughput or rate). The performance evaluation in chapter 7 shows that these simple estimated upper bounds are close to PVFS2 real-world throughput.

4.1 Resources Limiting the Performance

4.1.1 I/O subsystem

A hard disk needs a few milliseconds to move the hard disk heads to the correct position (seek time) and to wait until the sought block rotates by. The time needed to place the head over the correct block is the **access time**. The **track-to-track seek time** is the time a disk drive's head needs to move to the adjacent track. Once the head is placed, subsequent blocks can be read or modified very fast, which results in a higher **transfer rate**. Depending on the file system used, the throughput is less than the possible transfer rate. In order to improve performance, a disk typically prefetches and caches blocks.

Additionally, the operating system buffers a fair amount of an I/O operation. The buffer size depends on the server memory. A write operation can be buffered efficiently so it can complete before data is actually written to disk. A read operation can completely omit an I/O operation if the data is in memory. Otherwise, the operation has to wait.

4.1.2 Network

A message sent over a network interconnection needs some time until it arrives. Also, some time is needed for the message content to be given to the operating system by the network interface card. This is the **latency**. Twice the latency is the **round-trip-time** (RTT), which is the shortest amount of time expected to receive a response to a request. **Bandwidth** is the number of bits which can be transferred in a specific time. This is a real upper bound to the throughput. Because protocols like TCP have some overhead, the **throughput** is smaller than the bandwidth.

Latency and bandwidth depend on the used network technology. Latency is also influenced by the distance, network topology and the connection type (copper-cable, fibre, wireless, ...).

4.1.3 CPU

CPU speed and architecture defines the time needed to process instructions. PVFS2 mostly uses efficient data structures like huge hash tables. In comparison to the network latency and storage subsystem's access time, the CPU is the fastest component. Especially if we connect various clients to one server, CPU is not expected to limit operation throughput. Assume that each request needs the same time for processing. Then the number of requests which can be processed in a time interval is determined by the CPU's capabilities.

4.2 Scalability and Estimated Upper Bounds

Scalability in terms of a parallel file system's capabilities means:

1. the required resources and provided performance grow proportional to the number of clients
2. equal increase in the number of clients and in the number of servers preserves per client throughput
3. extra clients should increase overall performance if possible, but never decrease throughput

Requests from additional clients can saturate the network or I/O system, on the other hand a server's CPU can be kept busy. In the next paragraphs we will calculate estimated upper bound for the throughput of some operations. Therefore, concrete hardware data of our cluster is used. The analysis is kept very simple, only concerning the three bounds known from the former section.

4.2.1 Example hardware specification

The calculations use the following facts which are taken from the hardware we have on our cluster:

- **Configuration:** One metadata server and five data servers.
- **Network architecture:** Gigabit Ethernet, interconnected with copper cable linked to a switch. Between two machines the round-trip time is 0.52 ms and a throughput of 107 MByte/sec can be achieved. Performance evaluation was done using `netperf` and its TCP stream and TCP request/response tests. Netperf's tests use a single TCP connection. A PVFS2 client-server interconnection uses one persistent TCP connection as well. Therefore, PVFS2's behavior is expected to be comparable with netperf's results. An ICMP echo reply takes on average 0.2 ms which is much slower than netperf's round-trip time. And the maximum bandwidth for 1 GBit/sec Ethernet is 125 MByte/sec.
- **Hard disk:** The specification from [2]: the disk's transfer rate is between 29 and 56 MByte/sec, average access time is 8.5 ms, maximum access time is 15 ms and the track-to-track seek is 1.1 ms.
  The effective read throughput of 39.8 MByte/sec respective 38.4 MByte/sec for write were

measured using `dd` and a 5 gigabyte file to eliminate kernel buffering effects. For simplicity, 40 MByte/sec is assumed.
Kernel cache behavior is tested with a 250 MByte file and results in 473 MByte/sec for an in-memory read and 171 MByte/sec for a write.
The underlying file system is ext3 mounted with the options `commit=60` and `data=writeback`.

- **Machines:** The machines are equipped with two Intel Xeon 2000 Mhz processors which have the Hyperthreading option deactivated by the kernel's boot parameters.

4.2.2 Metadata operations

The expected metadata performance depends on the internal processing of the request. Remember, an operation is internally divided into multiple small steps. Normally, the steps have to be executed in serial because later steps depend on data of the former steps.
Messages transferred and data written during a metadata request are only a few bytes, so we ignore them. Also, we ignore instruction processing time. The network latency and the disk's access time have the major influence on the performance of metadata requests. Thus, only the communication behavior and number of I/O requests necessary for an operation are considered in this paper.

The analysis of different metadata operations is quite similar, as a consequence we analyse only one operation as representative for metadata operations which is the create operation. First, we analyse the processing of a create operation to determine the number of I/O operations and network messages. The realization of the file creation is described in section 3.4.2.
Client side states and expensive operations are as follows:

- C1: Assume the client already has the parent directories attributes in the acache, thus no request is necessary.

- C3: One network message between client and server to create metafile. This initiates one I/O operation on the server.

- C4: Datafile creation: for each data server, another network message and one I/O operation per datafile.

- C5: Set metafile attributes and keyvals. This requires one network message resulting in 4 I/O operations. Three operations modify objects' states.

- C7: Creation of the directory entry requires a request over network and triggers up to 4 I/O operations. The parent directory's attributes are updated and a new keyval is added. Thus, two modifying operations occur.

Regarding the internal operation processing a sequential file creation needs 4 times network RTT and in the worst case up to 10 I/O operations. In case the modifying operations were synchronized and read operations were buffered, we could save only 3 operations.

With the knowledge of the number of I/O operations and network messages an estimated upper bound for the creation rate can be calculated based on the hardware assumptions. This will be done for a given example in the next paragraphs.
In the worst case a file creation needs 10 I/O operations and 4 network messages. Therefore, the total time for an operation is $10 \cdot 8.5\,ms + 4 \cdot 0.52\,ms = 86.08\,ms$, resulting in an average of 11.6 create operations per second. In the worst case of the disk, $10 \cdot 15\,ms + 4 \cdot 0.52\,ms = 152.08\,ms$ are needed,

so 6.6 creates are possible. Of course, the access time may be smaller, especially if the actuator has to move the disk's heads only a bit. Therefore, we use the track-to-track seek as an optimistic bound. This results in $10 \cdot 1.1\,ms + 4 \cdot 0.52\,ms = 13.08\,ms$ respective a rate of 76.5 creates per second.

Normally, the caching mechanisms implemented in the hard disk and operation system reduce the number of I/O operations significantly. Let us assume we don't need I/O operations at all. Then we would only have the network latency for 4 messages which is $2.08\,ms$. This would result in 480.8 operations per second.

Clearly it is necessary to reduce the number of I/O operations, due to domination of the I/O access time. However, sometimes we cannot prevent I/O. In particular, a read request for uncached data is always necessary. For writes and modifying operations a non-syncing mode is expected to be much faster.

Additional clients, hosted on different machines, should be able to create at the same rate until the servers are busy or the network is saturated, which is expected to happen later. In syncing mode, one client ends up with an average rate of 11.6 and an optimistic rate of 76.5. Multiple clients increase the overall rate only a bit, by eliminating the network latency. Another effect which might improve performance for multiple clients is the disk elevator algorithm, which can reorder the requests to reduce disk access time. These effects are not considered in this paper. Because of the PVFS2 metadata server randomization, servers share the clients requests. Thus multiple servers reduce the load to every server. For non-syncing configuration, increasing the servers improves their disk throughput i.e. one server average rate 11.6, two server rate 23.2 and so on.

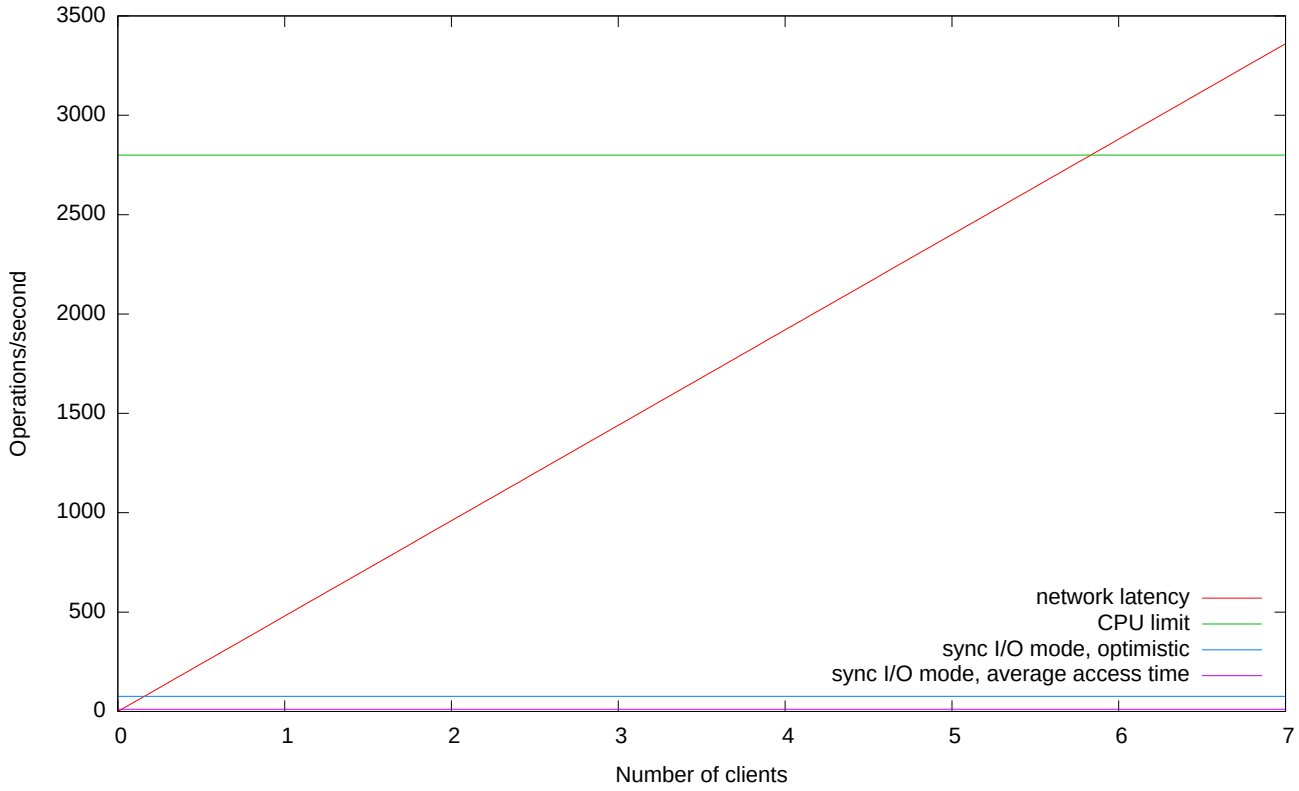

Figure 4.1: Estimated upper bounds for the create operation

Figure 4.1 is a diagram that shows the aggregated estimated performance for a variable number of clients. For the CPU, a limit of 2800 creates/sec is used as an example. This is the creation rate that is measured on our cluster. Then the CPU is busy processing the clients' requests. The diagram shows that it is vital to use caching mechanisms if possible.

For the other meta operations the graph looks similar. However, the amount of network and I/O requests may be different. This modifies the non-synchronizing gradient and alters the y-coordinate of the synchronizing variants.

4.2.3 Large I/O requests

I/O of larger amounts of data, requested by one interface call should be handled efficiently by the file system. Due to the continuous data transfer we can ignore network latency and disk access time. The throughput is now the limiting factor. If the server-side buffer is too small to store the amount of data, the I/O subsystem defines the maximum measured throughput. Otherwise the network throughput limits performance. The diagram 4.2 shows the estimated upper bounds for 5 data servers aggregated throughput. Thus, the approximate network throughput for the servers is 535 MByte/sec and for the servers' I/O system 200 MByte/sec. The theoretically achievable network bandwidth is 700 MByte/sec. As long as the servers have unused network capacity, additional clients increase the overall throughput. It is assumed that data is distributed equally between the different data servers.

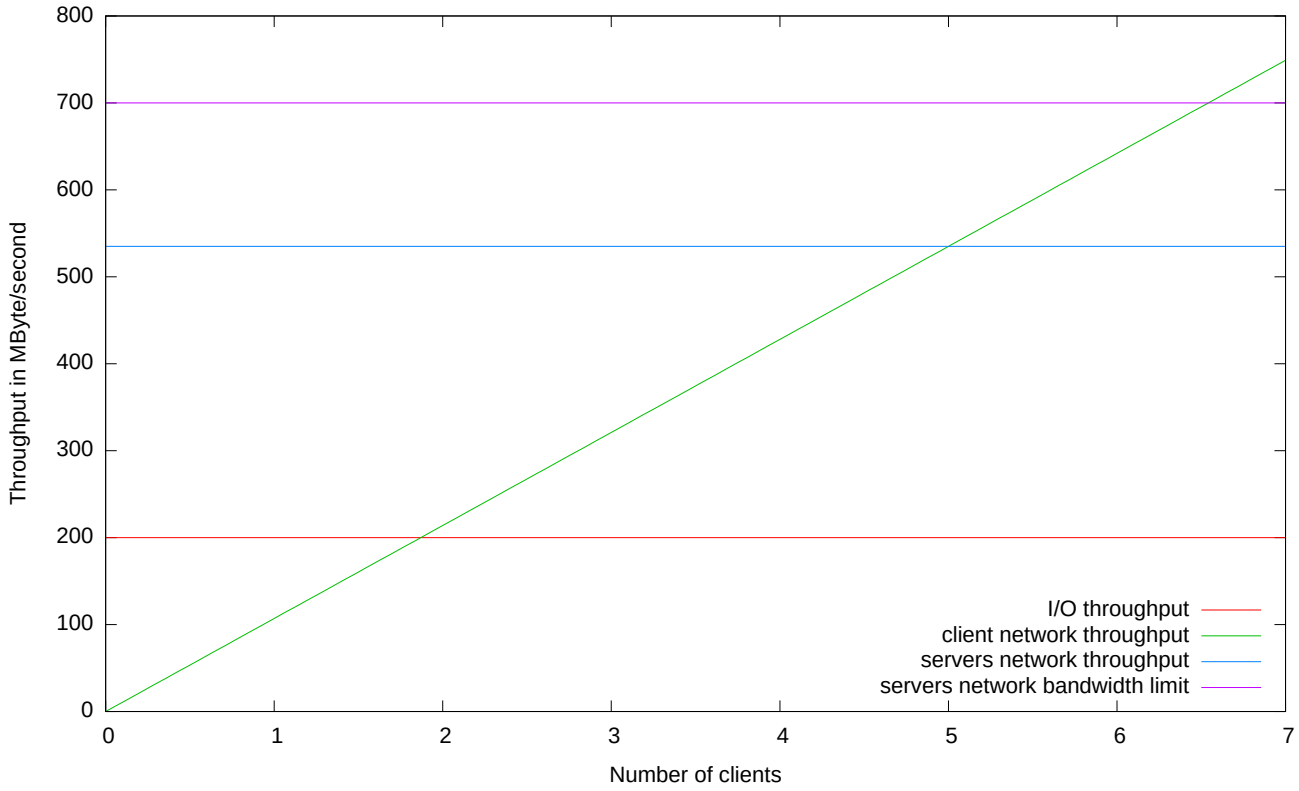

Figure 4.2: Estimated upper bounds for large I/O requests

4.2.4 Small I/O requests

This request type represents a mixed form, influenced by latency as well as by the components throughput. In difference to the large I/O requests this request type invokes a system interface function for a small amount of contiguous data. The amount of data requested is called **block size**. If we do a subsequent request to the same file, PVFS2's caching mechanisms are expected to reduce the metadata communication. However, setup of an I/O request takes some time for the client as

well as for the server. The whole process needs at least round-trip time in order to piggyback the data with the initiating message and to acknowledge a successful write. The analysis is split into two parts. First, the estimate performance of one client is analysed and then performance of multiple clients.

For one client The parallel file system's achievable I/O throughput can be calculated using the following formula, which takes the network latency as well as the network throughput into account.

$$t(s, l, n) = \frac{s}{s/n + l}$$

The variable s is the block size, l is the latency and n is the network throughput.
Remember the 0.52 ms network latency and assume we transfer 1 KByte per request. Therefore, by using the formula we get a maximum throughput of 1.9 MByte/sec per client! This shows that it is very inefficient to access files of a parallel filesystem with small block sizes directly. In order to achieve a better throughput in this case additional concepts like client-side buffering have to be used. For 1 MByte of data the network interface needs about 9.35 ms to set it on the wire. Thus, about 101.3 single messages can be sent in a second, resulting in a throughput of 101.3 MBytes/sec. However, network protocol overhead and behavior decrease the rate.

I/O subsystem's access time can be simply incorporated into the network latency. Assume a synchronizing operation and that the server has to move the hard disk's mechanic for every request exactly once. Throughput for the I/O subsystem is slower than for the network, thus the disk throughput limits overall performance. The calculated throughput is an upper bound for every parallel file system. The diagram 4.3 shows the expected throughput depending on the block size. The figures 4.4 and 4.5 are subdiagrams displaying the throughput for smaller block sizes.

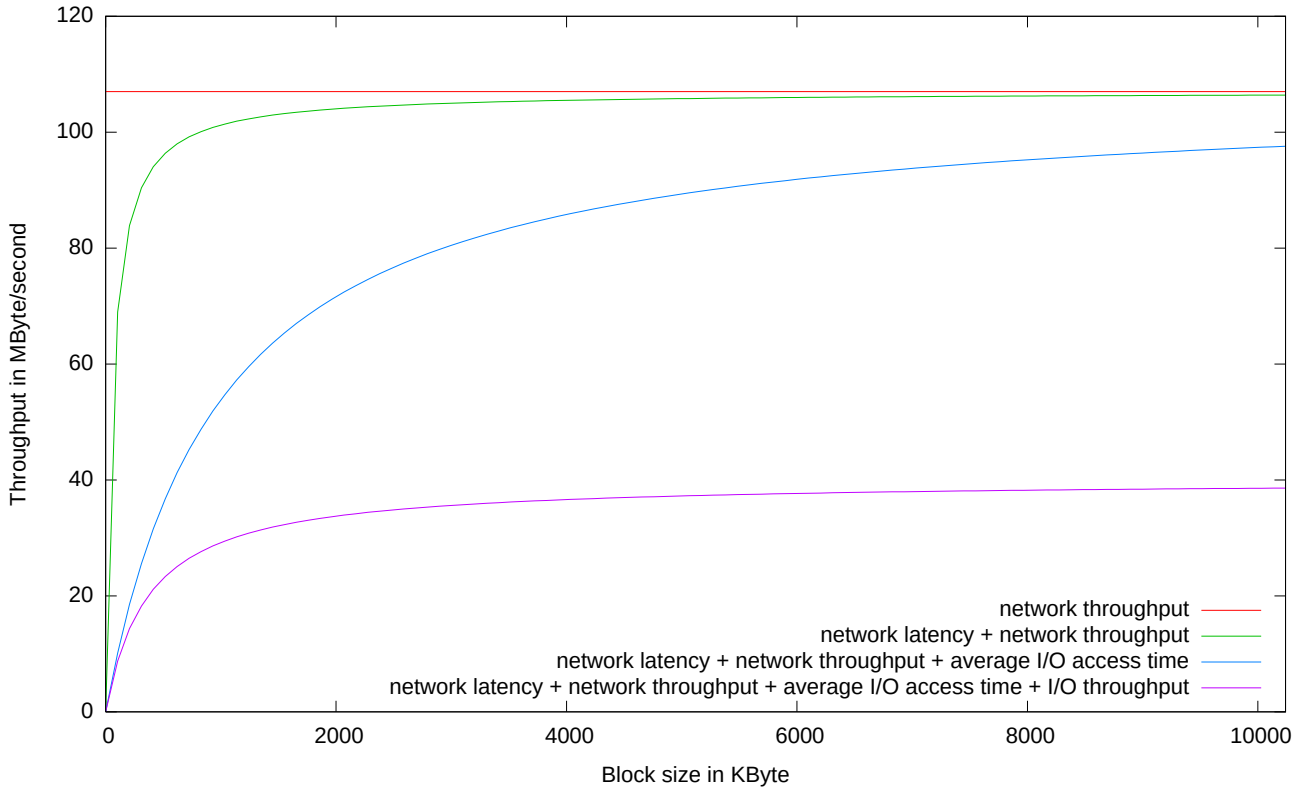

Figure 4.3: Estimated upper bounds for one client using small I/O requests with a varying block size between 0 and 10 MByte

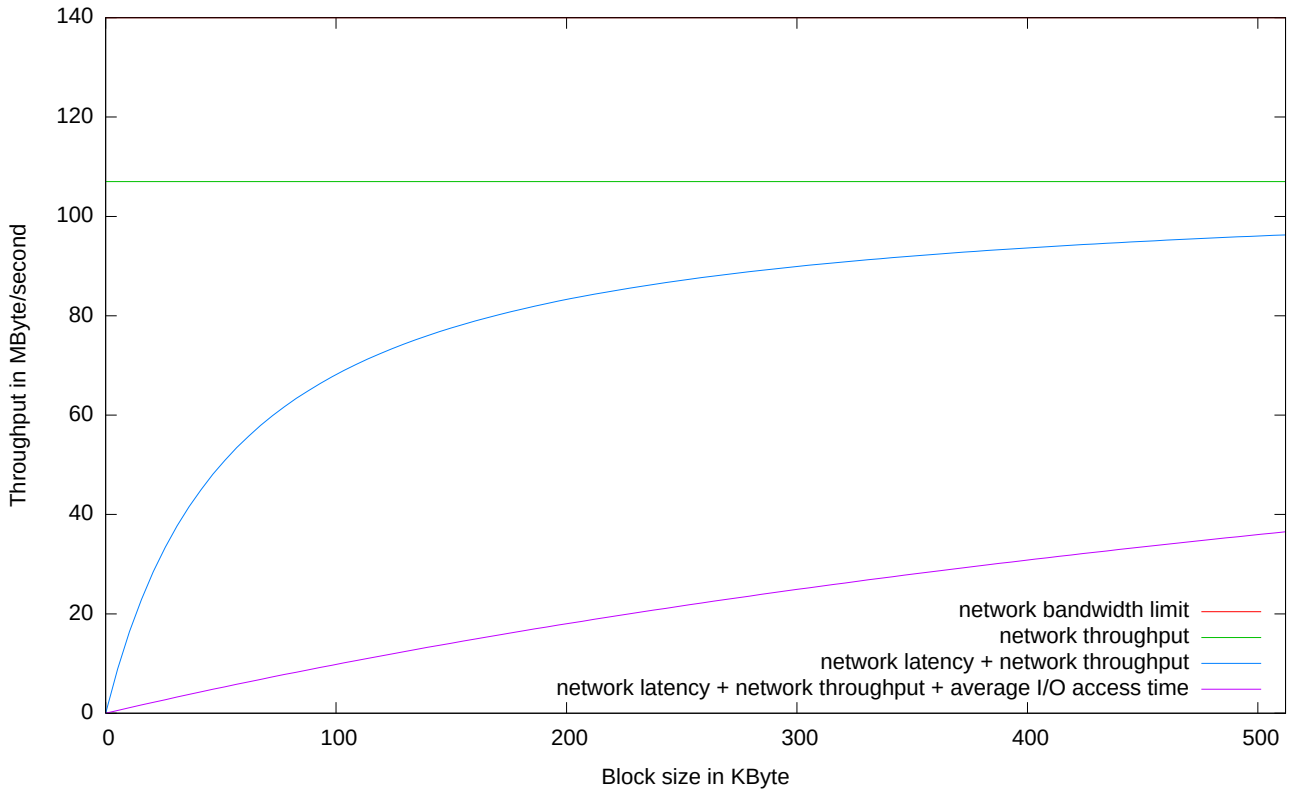

Figure 4.4: Estimated upper bounds for one client using small I/O requests with a varying block size between 0 and 512 KByte

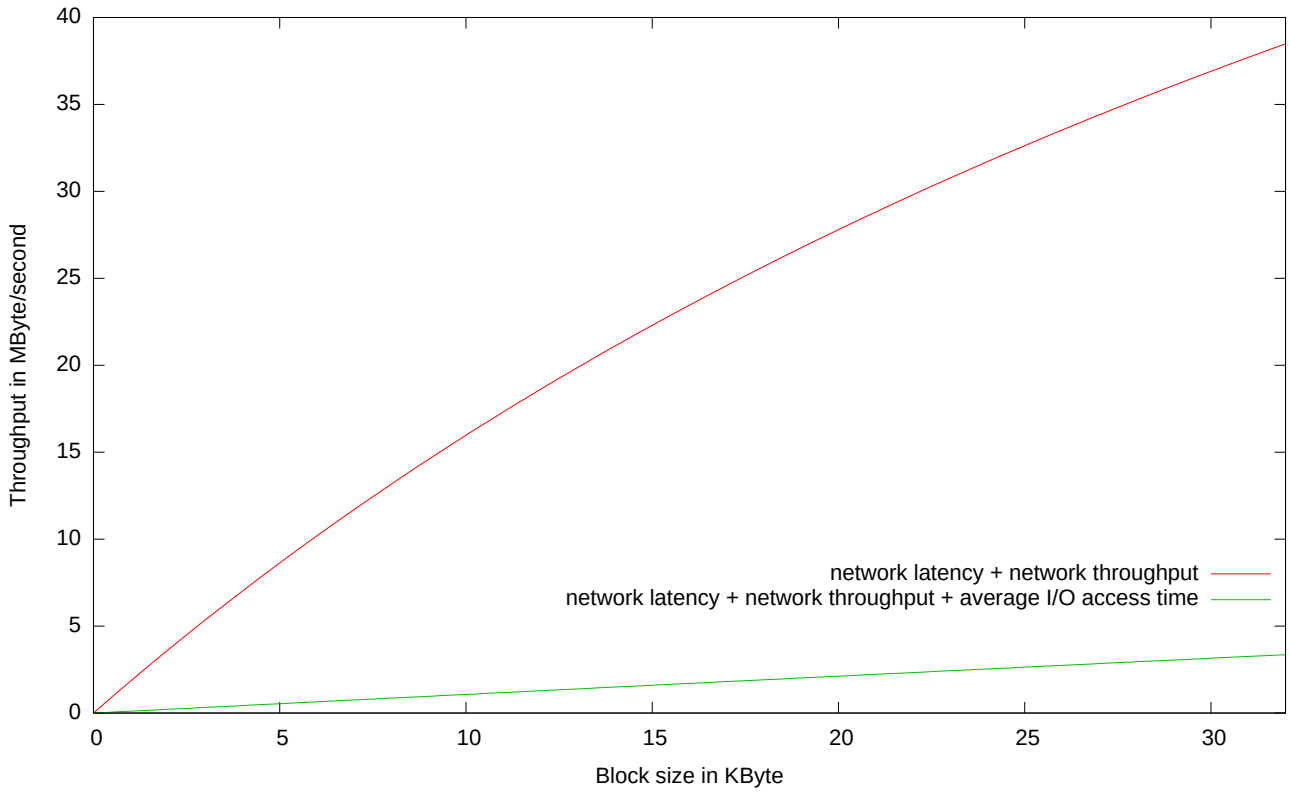

Figure 4.5: Estimated upper bounds for one client using small I/O requests with a varying block size between 0 and 32 KByte

For multiple clients In that case the calculation of the overall throughput is more complicated.

If the server caches data efficiently and due to the serial request processing by the client, the network latency remains the bottleneck. This effect can be seen for 1 KByte requests in figure 4.6. More clients increase the overall throughput, due to parallel execution on the server. However, it is hard to analyze the I/O access pattern in this case as it depends on the servers' workflow. Therefore, this analysis is omitted in this paper.

The servers I/O limits for doing only 1 KByte requests are also shown in the diagram. A clever non-syncing implementation can combine several write operations into one lowlevel I/O request by writing all data at once and also prefetch more data in the read case. This strategy is used for example in ROMIO and is known as data-sieving. Additionally, the linux kernel buffers files, thus the I/O limits shown won't yield in the real world, but show the necessity of such caching methods.

A larger request size using multiple clients leads to saturation of either the server's I/O subsystem, the network interface card or the CPU.

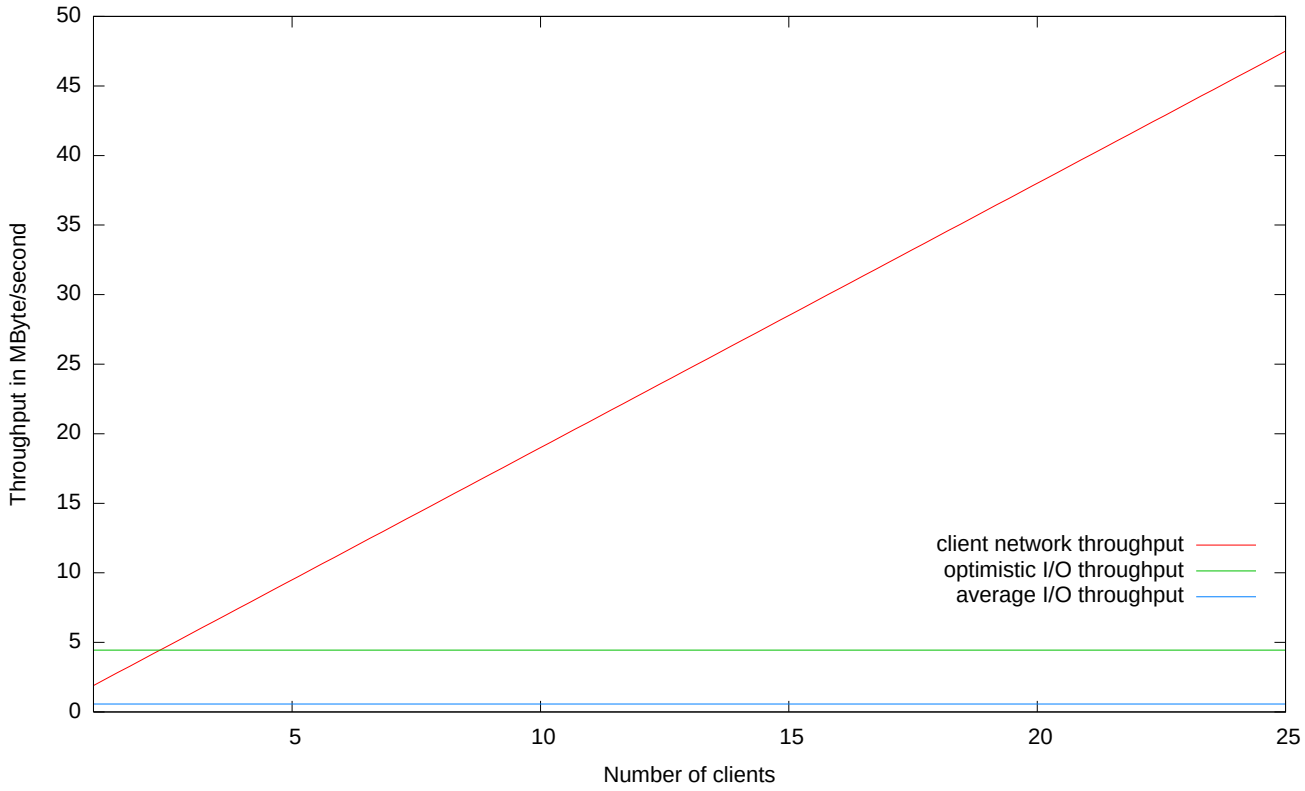

Figure 4.6: Estimated upper bounds for multiple clients using 1 KByte I/O requests

Summary: Network architecture, I/O subsystem and CPU provide limits for a parallel application, especially for a parallel file system. Network latency and disk latency restrict small operations, whereas throughput is the limiting factor for large I/O requests. A machine's CPU limits the number of requests processed in parallel. In order to facilitate a qualitative analysis of practical data, it is important to take these estimated upper bounds into account.
The next chapter introduces modifications to the PVFS2 source code that result in two new Trove modules.

5 Software Design

This chapter discusses at first the phases of the project and then introduces the new components that are designed, implemented and evaluated.

5.1 Project Phases

The idea to create a upper bound for PVFS2's performance by using a Trove module emerged. With such a bound bottlenecks of the architecture can be found. Furthermore, it allows to evaluate the efficiency of the persistency layer.
The new module should coexist with the current Trove module. Therefore, the Troves module support is enhanced. This divides the trove layer into storage management method and collection method. The storage method manages the collections including their attributes. The collection method takes care of the persistent representation of the file system objects and may be selected on the collection's creation.

A new method is implemented, the Trove Analyzation Stub (TAS), which uses queues as basic data structure. This method should provide a upper bound for specific metadata operations. A benchmarking of create operation with MPI points out that creation slows down. This happened because the object's directory entry is checked before the object is created, thus all keyval elements were compared. In order to improve overall throughput for the test cases several configuration options were added. However, this breaks the correct handling of metadata.

TAS gets adapted to use red-black-trees as underlying data structure. This implementation stores metadata correct. A performance comparison between the two basic data structures shows that the red-black-tree is not worse than the optimized queue implementation which does not handle all metadata correct. Hence, the red-black-tree version is used for further analysis.

A benchmarking of the file creation points out that the creation of more than 10240 files per clients dramatically decreases the performance. In order to eliminate the influence of PVFS2's ADIO driver and ROMIO `pvfs2-bench` was programmed. This is a program which uses the system interface to benchmark the creation, deletion and I/O throughput. Further debugging shows a problem with the clients acache implementation. The acache tries to free entries during creation if it is full. This is not possible because the cached object's timeout is not reached. Due to the slowdown a new acache implementation was done. It uses red-black-trees and least-recently-used as crowding out strategy. Also, this should figure out whether the red-black-tree overhead does impact the performance. This was not the case.
A reduction of the cache-timeout in the file system configuration also prevents the bad results.

The I/O throughput is benchmarked for some test cases. It shows that the achieved I/O throughput is close to the TAS results. On the other hand the metadata throughput of DBPF on tmpfs is only one tenth of TAS throughput and decreases with additional files in the same directory.

A DBPF code inspection point out multiple possible reasons:

- Each system level object has a own database for keyval entries, which has to be opened and closed.

- Operations are enqueued and processed from another thread.

- There is additional overhead of the handle management. A handle manager maintains the handles of the removed objects to guarantee the handle is not reused until the recycle time passed.

Currently, the C files of DBPF are 8000 lines of code and share some code fragments with other PVFS2 components. This makes it hard to figure out the reasons of the slow down.
In order to analyse different strategies the alternative implementation ALT is created, which stores metadata persistent like DBPF. The new module is a simple research prototype, which is used for evaluation of different approaches.

During the implementation some bugs are revealed and feedback is given to the developers concerning several issues.

The performance evaluation of the different modules is presented in the next chapter.

5.2 Decisions

The next few paragraphs summarize the crucial decisions that were made to push the project.

- **Division of Trove into storage and collection method**
  It is not possible for different methods to manage the same storage space. Collection management does not influence the servers run-time efficiency, but requires to implement some functions. Separation makes it possible to implement a new collection method, by using the existing DBPF storage management. This reduces the programmers effort.

- **Implementation of a correct metadata handling**
  Section 3.4 shows that most operations either check the result of a request afterwards or operate on manipulated objects. Thus, it is necessary to memorize the manipulation of the dataspace for some time. Also complex benchmarking application may access the metadata randomly. However, the method's internal data structures should handle requests efficiently.

- **Focus to analyze MPI**
  The analyzation is focused on the MPI layer. The VFS integration is not examined, because PVFS2 is especially developed for MPI use. Also performance is important for parallel scientific applications and there are less layers in between.

- **Benchmarking**
  The efficiencies of the new approaches are compared with different DBPF configurations, namely syncing and non-syncing option and DBPF using tmpfs as storage, which is a linux file system for main memory. Tmpfs efficiency is a good bound for every underlying I/O subsystem, hence shows bottlenecks of the DBPF implementation.

5.3 Enhancement of Trove Module Support

In order to allow the selection of a Trove method for each collection the following modifications are made:

- A new file system configuration keyword is introduced: `TroveModule`, which specifies the name of the method to use for the collection. Internally, the method name is converted to the method id.

- During the initialization with trove_initialization, the method id sets the storage module which a global variable. Instead of using the method for all requests, this module will be used only to manage collection information.

- The storage method support is unfinished. The modules implemented during this work are not able to administrate collections. The storage administration is done by DBPF.

- The collection method can be selected during creation (trove_collection_create) and has to be saved by the storage method. DBPF is adapted to save the collection's method id in the collection attribute database.

- Extension of the trove interface with a new function get_collection_method_id which is expected to return the id of the responsible method from the storage method.

- Implementation of the function map_coll_id_to_method:
  This function invokes the storage method's get_collection_method_id function and caches the result in a linked list to reduce overhead. This reflects that normally only a few file systems are hosted, which is enough for benchmarking.

5.4 Trove Analyzation Stub (TAS)

5.4.1 Overview

The Trove Analyzation Stub is written to provide a fast in-memory file system representation. Several compile-time options have been added to control the internal behavior in order to design a fast realization. Basic data structures are double linked lists and red-black-trees. During compile time one of the data structures can be chosen to administrate the keyval pairs and dataspace objects.
During the server shutdown metadata is saved on disk in a simple text based file format to allow offline inspection and manipulation. The metadata is loaded on server startup to preserve the state of the file system. Furthermore, the approximate memory usage is printed on shutdown.
Only the interface methods needed to handle requests used during the benchmarking are implemented. These routines provide a basic file system support for most operations. TAS fakes bytestream I/O operations and does no real I/O. Instead for reads the buffer is returned unmodified and for writes the transmitted data is discarded.

For one or the other reason the following functions were not implemented:

- **bstream_read_at, bstream_write_at, keyval_iterate_keys...**
  PVFS2 upper layers do not use all low-level functions specified. These unused routines are not

implemented.

- **Collection management**
  Collection management is done by DBPF.

- **Context and test completion functions**
  All TAS operations complete immediately. Also the functions are not needed when the implementation has thread support.

- **Manipulation of extended attributes**
  Extended attributes are not supported by TAS. However, they could be easily integrated.

- **Flushing of files**
  There is no bytestream representation, thus flushing is not necessary.

5.4.2 Dataspace objects

Each dataspace object is stored in a structure holding the object's handle, collection id, bytestream size, number of keyval pairs, the objects attributes and a pointer to the objects key value pairs. The global variable handleCache points to the data structure holding all dataspace objects.
Depending on the selected data structure during comile time, keyval pairs and dataspace objects are linked in an unsorted queue or are maintained in a red-black-tree. Lookup of an object with its objects reference is the most common operation, hence should be fast. Also searching the value for an object key happens quite often.
If there exist n objects, the worst case lookup of a object is $O(n)$ for the queue and $O(log(n))$ for the red-black-tree. Clearly, using the queue is only useful for benchmarking if the tester knows that the worst case is avoided during the tests. Then, the tests can benefit from the lower administration overhead of the queue. Detailed implementation depends on the selected data structure.

Double linked list The double linked list has some optimizations that can be used to guarantee a worst case time, especially for the create operation.

- Least-recently used objects have an early position in the queue, to reduce lookup time for subsequent usage. This is realized by pushing the object to the lists head during a lookup. For example an object creation benchmark which does not access created elements later, benefits a lot from this strategy.

- Newly created keyvals are inserted at the head of the objects keyval pair list.

- Keys are first compared using a hash function to minimize the number of string comparisons.

- Modification of a keyval does not overwrite or lookup the existent keyval, instead a new keyval pair is added.

- The maximum number of search iterations during a keyval lookup can be set to guarantee the running time. However, keyvals which are at the queue's end would never found. This is a big drawback, for example files in larger directories get inaccessible, because their handle cannot be found.

An evaluation of the queue and the red-black-tree showed that even for the creation of over 50000 files the red-black-tree is very close to the queue's implementation. Thus, the red-black-tree is used for further benchmarking.

Red-black-tree A red-black-tree is a nearly-balanced tree that uses an extra color attribute per node to maintain balance. No leaf is more than twice as far from the root as any other. Insertion, lookup and removal of objects can be done with a complexity of $O(log(n))$.

Red-black-trees need a comparison function being able to decide which of two objects is smaller or if they are the same object. The function chosen for the dataspace objects defines that an object is smaller if the handle is smaller or the handles are equal but the collection id is smaller. Key/value pairs are sorted by length lexicographic order.

Keyval iteration is done using a additional double linked list. This operation is very expensive in TAS, so it should be avoided for benchmarking. Normally, the iteration is used only for directory listing. An interface modification can improve the situation; however it would be incompatible to the current handling of DBPF and would require modifications of some statemachines. This is described in detail, later.

Handle management Handle management ensures that handles are not reused for a specific time to prevent operations on wrong objects. It is mainly used during dataspace creation and removal. It can be used to verify an object's existence during other operations, as well.

TAS never reuses handles. It simply increments the handle number each time a new handle is requested and does not check an overflow. Once an overflow occurs the file system is invalid. But this should only happen for long running servers.

5.4.3 Detailed function description

The next paragraphs clarify the implementation of the functions used during benchmark. This should point out the methods' overhead.

tas_collection_setinfo This function is invoked on startup to set multiple parameters. When a collection's handle range is set, read all metadata from the persistent storage and set the values of the next metadata and the next datafile correct.

tas_bstream_read_list This function gets an array of memory regions and an array of file regions, which should be read into the memory regions. TAS only summarizes the input stream size and sets the number of bytes read to this value. This simulates reading and should be a real upper bound to all read operations. Complexity is $O(1)$.

tas_bstream_write_list Similar to the read function. Additionally, the datafile is searched in the red-black-tree and the object's stream size is adapted in case we wrote data to the end of the file. Operation complexity for the tree is $O(\log(n))$.

tas_dspace_create Take the next handle number for the new object depending on the type (metadata or datafile). In case a fixed handle is selected, set the next usable handle number to the fixed number + 1. Thus, slip handles in between. Normally, this is only used to enforce the number of the root handle. Insert the newly created object into the data structure (i.e. red-black-tree or queue). Therefore, the operation complexity is $O(\log(n))$.

tas_keyval_write First, the object is searched within the data structure by the object's reference. Then, the key is searched within the keyval pairs. If it does not exist, it is inserted, else it is modified. Overall complexity is $O(\log(n))$.

Other operations Most operations work in a similar fashion: lookup of the object in the handle cache data structure, search for the keyval object and modify data if necessary. The complexity for keyval_read, keyval_remove, dspace_remove and so on is $O(\log(n))$.

5.5 ALTernative Implementation (ALT)

5.5.1 Overview

The aim of the **ALT**ernative method is to provide a simple implementation for investigating the impact of different strategies. It utilizes Berkeley DB with binary trees to store metadata and UNIX files for bytestreams. Thus, it provides a persistent representation of the file system like DBPF. Metadata requests are processed immediately while stream I/O is executed by additional threads. Caching is not done explicitly, instead the mature Berkeley DB does this job.

5.5.2 Dataspace objects

A single database contains all objects and takes an object's reference as a key. A key-lookup returns the common object attributes, including the number of keyval pairs. Another database stores all keyval pairs and uses the object reference and the pair's key as database key.

Basic support for a three-keyval database version is implemented, which maintains separate databases for the metafile's keys datafile_handles and metafile_dist. Keyval names are defined by specific databases, thus they need not to be stored on disk. Hence, outsourcing can reduce the required disk space. However, a create request analysis shows that this decreases overall throughput a bit. On the other hand the source code gets complicated.

Keyval iteration has the same problem as in TAS. Therefore, the issue is discussed in more detail in the following section.

Handle management The free handle-ranges are maintained completely in memory in a red-black-tree. During startup used handles are removed from the available pool. There is no handle reuse timer, instead handles are free for reuse immediately. The memory usage of this solution depends on the amount of disjunct ranges. Thus, it is a drawback when there are many small ranges.

5.5.3 Detailed function description

alt_collection_setinfo If the function is invoked to set the collection handle range, the handle-management-range is initialized. An iteration over all persistent objects removes their handles from the available ones.

alt_bstream_resize, alt_bstream_read_list, alt_bstream_write_list Each function stores all operation parameters in a structure and enqueues the operation in a double linked list which is shared with the threads. A thread ready to process removes the first entry. The queue is protected by a mutex. Note that the thread does not need any metadata of the object it is operating on. Each bytestream is stored in one file which gets the name of the object handle. For each I/O access the file is opened again. Then, the I/O operation is performed and the file is closed. Thus, this realization is very simple.

keyval_write Lookup of the key. If it does not exist, read the object attribute, increment the keyval count and write the attributes. Store the keyval pair.
Alternatively, a key's existence could be checked by using the Berkeley DB's put operation flag DB_NOOVERWRITE.

keyval_remove First, try to remove the keyval pair. Then, read the object attributes, decrement the number of keyvals and write the attributes again.

dspace_create Get an unused handle number from the handle manager. This is supported in linear runtime, except in case the handle number is enforced. Put the object attributes into the database.

Other operations Perform the necessary metadata operations directly.

5.6 Keyval iteration

Currently, the Trove keyval iteration takes the position as a parameter. This is the number of already processed pairs. In case that for each object a new database holding the keyvals is created, this makes sense. Berkeley DB can maintain a database's record numbers automatically.
The records of one object are arranged in ALT's database in a sequential way to allow efficient keyval iteration once the current key is located. But, it is necessary to find the object's position. Therefore, the iteration starts from the first keyval pair and continues reading until the number of already processed entries is read. Hence, a pair might be read multiple times. This is expensive, of course.
Berkeley's binary tree implementation (BTree) can support record numbers. However, overall write performance suffers, thus this should be avoided.

Suggestion Lookup using the concrete key instead of the number is supported efficiently. It is possible to send the last processed key instead of the current position. However, the interface modification require a lot of code changes. Furthermore, the consistence of directory listing would be given, even if another client manipulates the directory.
Consider the following directory states and solutions for the new interface:

- **directory is deleted:** lookup aborts

- **another directory entry is added:** if the entry is inserted within the already processed entries, we can safely continue the iteration. The client can read the directory's timestamps after the iteration in order to ensure that the directory's entries did not change.
  In case the insertion takes place beyond the already read entries, we simply continue the iteration. This will hit the new entry during processing.

- **a directory entry is removed:** if the entry was not read yet, the listing is still valid. Otherwise, the client will detect the wrong entry either when it tries to access the non-existent object or by checking the directory's timestamp on iteration completion

- **the iteration's current object was deleted:** Berkeley DB supports a search flag for BTree access, that moves the cursor to the first database record whose key is greater or equal to the specified key. [7]

Summary: This section gives a detailed discussion of the new modules that are developed within the framework of this thesis. TAS is a module working without any I/O operation. It is especially constructed to support a fast access to common metadata operations.
The ALT module uses Berkeley DB and UNIX files to store all file system objects. This are the same underlying mechanisms as used by DBPF. I/O is handled in a straightforward way. The next chapter introduces the programs that will be used for benchmarking.

6 Benchmark Programs

There are many benchmarks for file systems, which were approved by the community, for example bonnie++. However, for parallel file systems no such common benchmark suite exists. There are several programs which attempt to fill the gap, i.e. the Effective I/O Bandwidth Benchmark (b_eff_io) or the NASA Parallel Benchmarks or Metabench[1]. These approaches are not suited to measure the whole range of interesting performance characteristics. Instead, the benchmarks make assumptions of access patterns used from the clients. Thus, a good result does not necessarily mean that a file system is a good choice for a given application.
Often, a scientific application is used directly to measure the I/O system's efficiency. However, the overall performance of an application depends much from the cluster configuration.

Therefore, I decided to use simple benchmarks suited for PVFS2 in order to measure points of special interests. These are I/O requests using a small data-amount (block-size) per access, large sequential I/O requests and the file creation operation as representative for metadata operations. For each data point the benchmarks run a program with a specific input. The next sections explain the programs' operating modes in order to assess the benchmarking results.

6.1 `mpi-io-test`

This small program is part of the PVFS2 CVS version and its purpose is to test the MPI-I/O interface. The program can be used for all ADIO modules, not only PVFS2.
In a single run each client accesses a a specified region in an MPI file. All clients use the same file and the regions are non-overlapping. However, clients can also be configured to use overlapping regions inside the file. Data is not synchronized during write.
Each processes of a program spawned with MPI gets a unique number assigned which is called the **rank**. The rank can be used in the program in order to select a processes behavior and tasks depending on its number. It is used in `mpi-io-test` to add an offset to the position of the I/O access to guarantee that each process accesses its correct file region.
There are two important input parameters: the **block size** defines the amount of data which is accessed per system interface function call and the **iteration count** which is the number of subsequent I/O operations.

Client program:

1. For i=0 to (iteration count - 1):
   Seek to the next write position which is calculated as follows
   $position = i * number\_of\_processes * block\_size + rank * block\_size$
   Write an amount of data per MPI I/O call equal to the selected block size

2. Wait until all processes finished writing

[1] A parallelized version of the postmark benchmark

3. For i=0 to (iteration count -1):
   Seek to the next read position which is calculated with the same formula as for writes
   Read an amount of data per MPI I/O call equal to the specified block size

4. Wait until all processes completed, calculate operations statistics including I/O throughput

6.2 mpi-md-more

This is a variant of `mpi-md-test`, which is also part of the PVFS2 CVS version. It is a simple program which measures the throughput of the create operation for a specified number of files. Additionally, the file resize operation can be benchmarked. The `mpi-md-test` is modified such that each processes works on a disjunct set of files.

Client program:

1. For i=1 to number of files:
   create a file in the test directory the name of which consists of the rank and i.

2. Wait for all processes to complete and calculate statistics

6.3 pvfs2-bench

The `pvfs2-bench` utility was developed in this thesis to measure operation throughput of the PVFS2 system interface on one client. Depending on the test selected, the system interface's file creation or deletion rate, as well as the contiguous read/write performance can be determined. In order to detect throughput changes during operations, the program can print the throughput every second. During an iteration only one system interface function is invoked. Hence, the client spends most time in processing requests without additional overhead.

Client program for I/O operation mode:

1. For i=1 to number of iterations:
   do I/O with the selected block size and increment next operation's file position

2. Calculate statistics

Client program for file creation mode:

1. For i=1 to number of files:
   create a file in the test directory and append the iteration number i to the filename

2. Calculate statistics

Client program for file deletion mode: Similar to the algorithm for file creation. However, the iteration decrements the current file number instead of increasing it. In conjunction with a earlier run of the creation mode this allows to benefit from the server's internal caching mechanisms.

The utility supports additional tests, which should point out the benefit of client and server caching mechanisms.
Therefore, in case of metadata in each iteration the file is first created and then deleted. For I/O operations, data of one block is written and then read again.

Summary: There exists no general purpose benchmark for parallel file systems. Three simple programs are introduced, which measure PVFS2's throughput of either data I/O or metadata. In the next chapter these programs are run multiple times with different configurations in order to form benchmarks for I/O and metadata.

7 Evaluation

In this chapter benchmarks of the different Trove modules are evaluated.

Mainly, the behavior of PVFS2 is analyzed for the following three cases: Small contiguous I/O requests, large contiguous I/O requests, and file creation throughput.

Most benchmarks are performed on the research group's cluster. For hardware details refer to section 4.2.1. Thanks to Rob Ross I got access to the Chiba City cluster hosted at the Argonne National Laboratory. The Chiba cluster is used to verify the results for a larger scale of nodes.

For each data point the appropriate benchmark program was run three times. The results are displayed in diagrams with error bars marking the minimum and maximum throughput of a run. If the benchmark uses multiple clients. Then, the throughput of all clients is aggregated.
To ensure a similar environment for all tests the PVFS2 servers got restarted and the PVFS2 storage space was recreated for each run.
The different lines of the diagrams refer to the Trove modules, i.e. TAS, ALT, DBPF and additional configuration options. A tmpfs extension in the name means the test was run on the linux kernel's in-memory file system, no-sync refers to the non-syncing configuration of DBPF. In diagrams comparing MPI and the system interface, sysint refers to a test done with pvfs2-bench.

Configurations Three different client-server configurations are used on the research group's cluster:

- **One metadata server and one data server** There is only one server, the clients are distributed over 7 different machines for MPI. Clients and server are disjunct.

- **One metadata server and five data servers** The clients are distributed on the server machines in a round robin fashion, beginning with the metadata server. One server acts as both, metadata and data server.

- **Five metadata servers and five data servers** Each machine is configured to act as metadata and data server. The clients are distributed on the same machines as the servers in a round robin fashion.

The detailed PVFS2 file system configuration can be found in the Appendix. For an I/O test the HandleRecycleTime is set to 5000, which is larger than the longest run. Hence, there are no additional metadata operations necessary. The simple stripe distribution function is used, which stripes file data over all data servers in 64 KByte chunks.

7.1 Small Contiguous I/O Requests

In this benchmark the clients access a 100 MByte file with different block sizes between 1 KByte and 10 MByte, which is the amount of data accessed per user-level-interface call. It is expected that the performance varies only a bit for a small difference in block size. Therefore, up to 2048 KByte the range is sampled with a block size equal to a power of 2 and then 5120 KByte and 10240 KByte are sampled. The `mpi-io-test` and `pvfs2-bench` are run for each block size with a appropriate iteration count.
The analysis of small contiguous I/O requests is structured as follows:

- **Results for one metadata server and one data server**
  First, the performance of the system interface for one client is determined with pvfs2-bench and then compared with the estimated upper bounds.
  Then, the I/O throughput of the MPI-I/O interface is compared with the throughput of the system interface in order to show the overhead of the MPI-I/O layer.
  Performance is measured for 5 clients using a variable block size, too. This performance can be used for comparison with later results. Also 5 clients are selected for this test because client and servers of the other configurations share the same machine and there are 5 machines in this case, each serving one client and one server.
  In addition, multiple clients are run with MPI to determine the server's behavior and maximum throughput under a high load.

- **Results for one metadata server and five data servers**
  With this configuration the same MPI tests are run as for only one data server. This allows a comparison of the results.

- **Comparison of the throughput for one and five data servers**
  The results for one and five data servers are compared.

- **Access of large files with small contiguous I/O requests**
  It turned out that the 100 MByte files are cached very well by the linux kernel. In order to reduce the influence of caching mechanisms this test case accesses a very large file to show the behavior of enforced I/O operations.

7.1.1 One metadata and one data server

At first, the results of the system interface are presented and then, the results achieved with MPI.

System interface

The I/O throughput of the system interface is measured for one client and displayed in three diagrams. The first diagram shows the throughput of the whole range and two others show the performance for small block sizes up to 500 KByte.
In the following the observations are presented. They are always placed before the figures.

Observations:

- The kernel caching mechanisms work well for this test set. Therefore the I/O is not limited by the hard disk's capabilities (see figures 7.1 - 7.3).
- There is a massive cut in the read performance for 128 KByte. The read throughput does not recover from this cut (see figure 7.2). All Trove modules perform similar, thus the I/O subsystem is not the bottleneck. This issue is discussed in detail later.
- Up to 64 KByte the read performance is better than the write performance (see figures 7.2 and 7.3).
- Read performance of different implementations are close together (see figures 7.1 - 7.3).
- For writes TAS is a bit faster than the other modules (see figures 7.1 - 7.3), hence in the architecture is room for improvement in this case. This might be achieved for example by combining several requests into one larger request. Performance of read requests is similar for all modules.
- The straightforward ALT I/O implementation is not worse than DBPF (see figures 7.1 - 7.3).

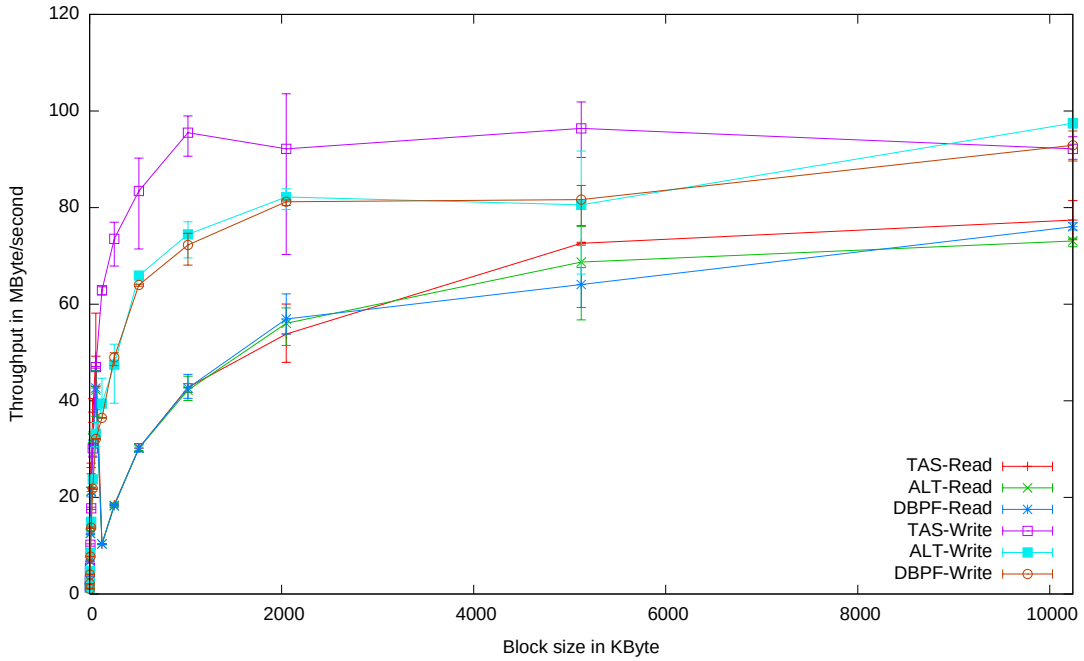

Figure 7.1: System interface I/O throughput for read and write using different block sizes

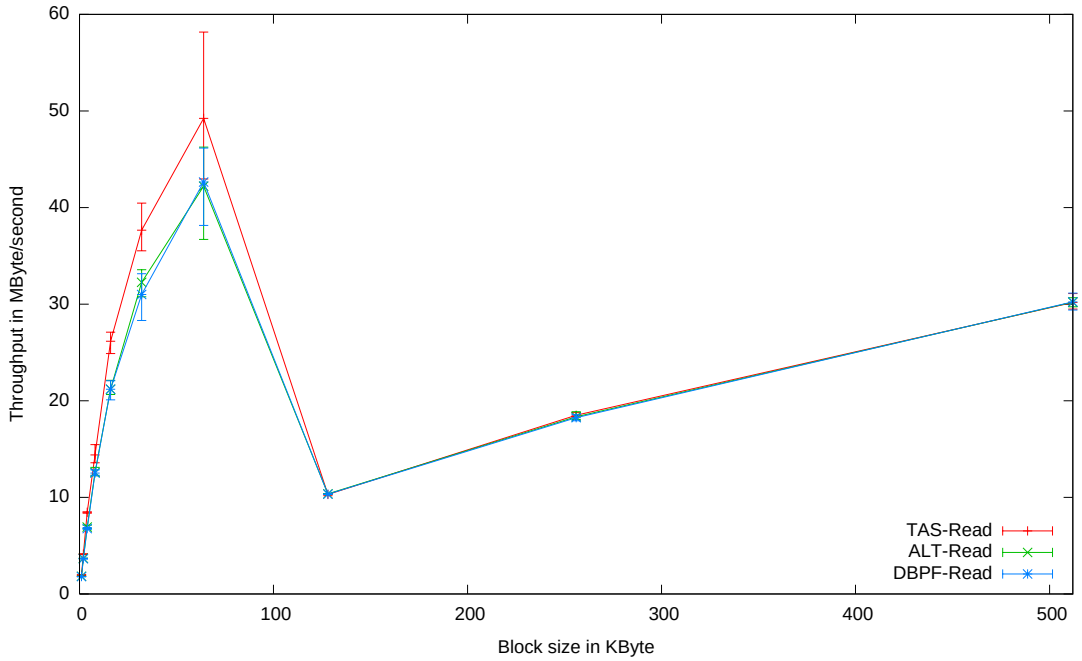

Figure 7.2: System interface read throughput using different small block sizes (data extracted from figure 7.1)

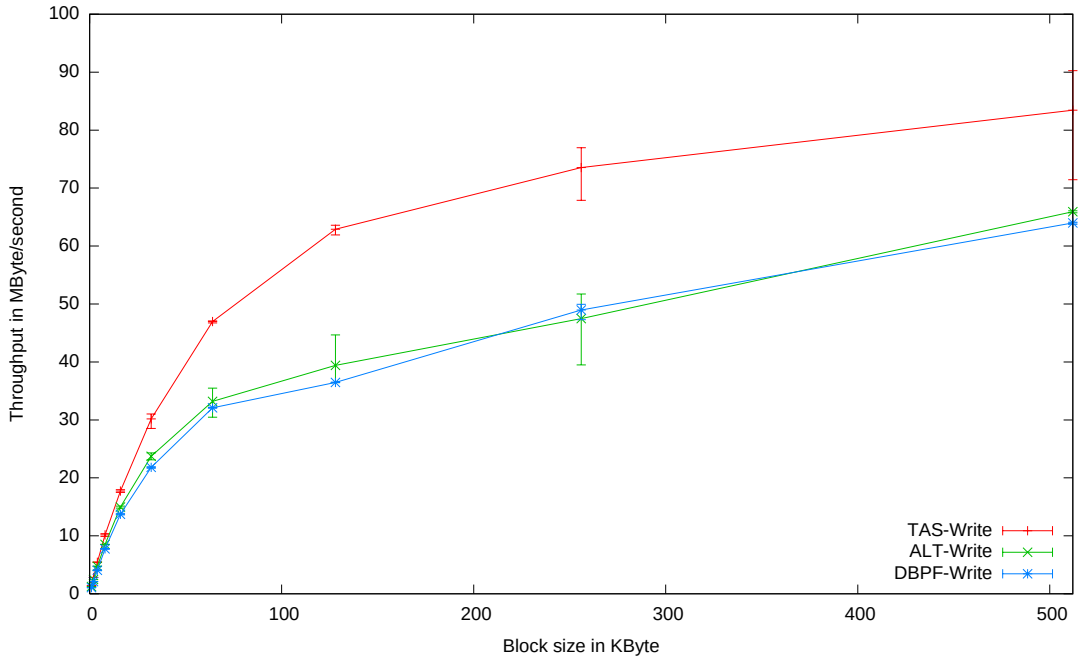

Figure 7.3: System interface write throughput using different block sizes (data extracted from figure 7.1)

Comparison of the measured throughput and the estimated upper bounds

The write performance is close to the network bound provided by netperf's results for TCP-latency and throughput. In the later metadata section is shown that the network latency derived from netperf's test is a bit too high. This can be seen for small block sizes and TAS read, too. However, this bound is expected to be close to the real throughput.

Up to 64 KByte read is even better. However, the performance drops for 128 KByte. Here is still investigation necessary, the reason might be the the flow-protocol or the handling of holes, which is only necessary for read requests. These are the main differences for read and write. Refer to 3.4.8 for a description about the internal processing of I/O requests.

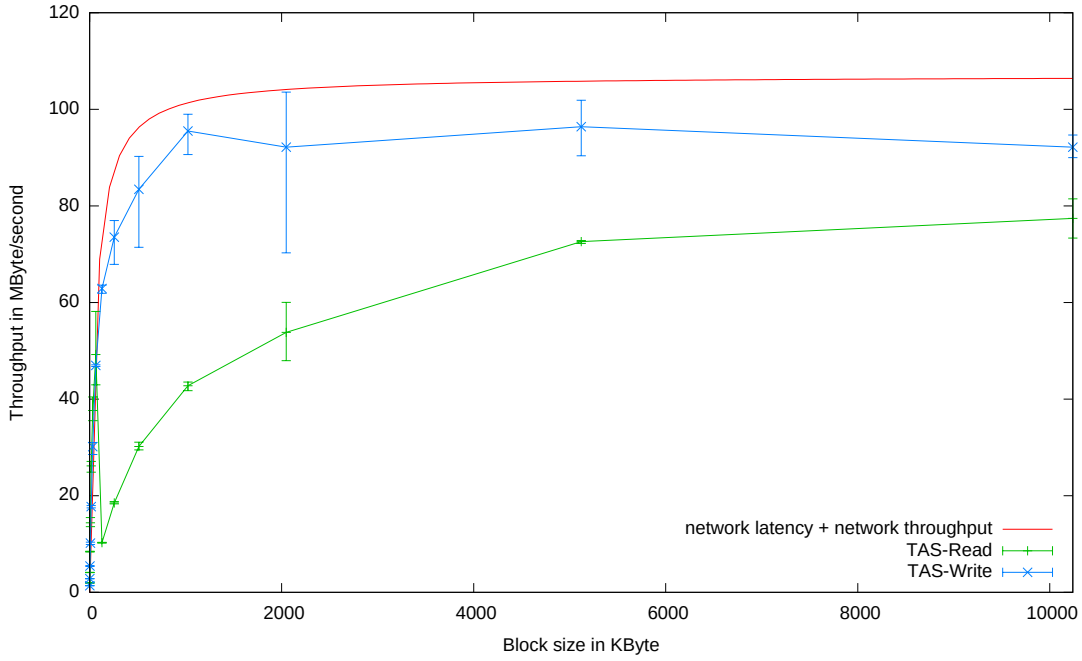

Figure 7.4: Comparison of the system interface I/O throughput for read and write and the estimated upper bounds using different block sizes

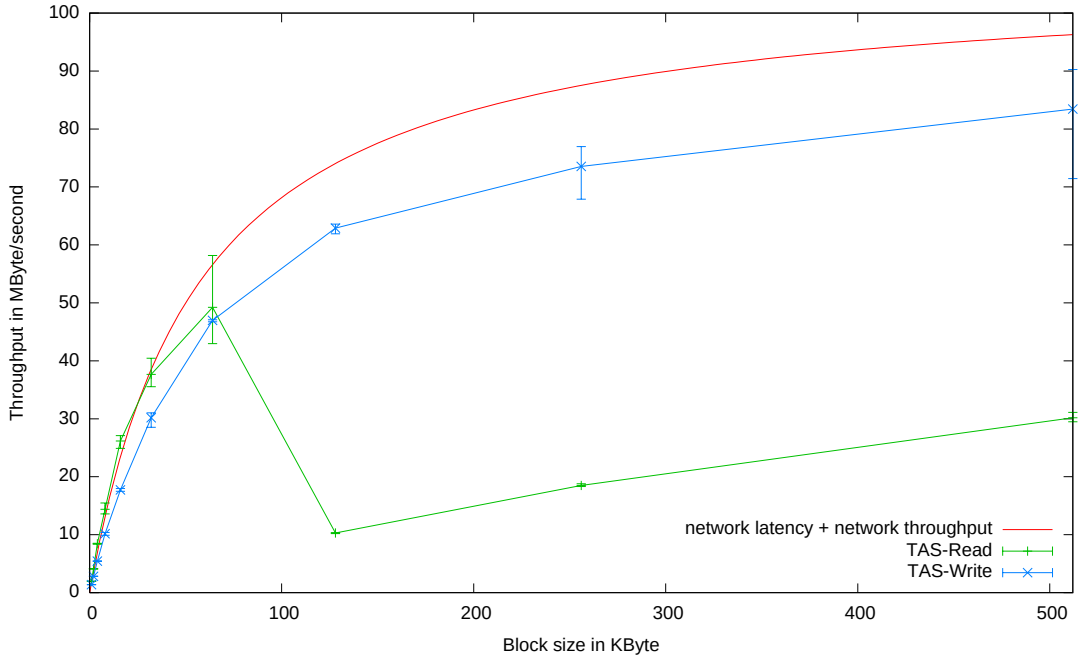

Figure 7.5: Comparison of the system interface I/O throughput for read and write and the estimated upper bounds using different small block sizes (data extracted from figure 7.4)

MPI-I/O interface

First, the read and write performance for one client using either the system interface or the MPI interface is compared. Then, the aggregated throughput for 5 clients accessing a file is presented. This points out the servers behavior dealing with multiple clients, each hosted on a dedicated machine. Furthermore, this figure is used for comparison with 5 data servers later. In addition, the I/O throughput for a variable client number between 1 and 50 is measured for a block size of 32 KBytes. The block size is chosen larger than 16 KBytes, which is the maximum amount of data able to be piggy packed to the initial request message for TCP. It is chosen smaller than 128 KBytes on which the performance drops for reads.

Observations:

- Due to the additional layer MPI lose a lot of performance for write operations (see figure 7.7). This does only influence the client. Hence, overall throughput for multiple clients is expected to be similar for both userlevel-interfaces. However, the system interface and MPI interface stick close together for read operations (see figure 7.6).

- The read performance cut for a block size of 128 KByte is still visible for 5 clients (see figures 7.8 and 7.9).

- For small block sizes and 5 clients the read throughput grows faster than the write throughput (see figures 7.9 and 7.11). For example the throughput of 100 MByte/sec is achieved for reads with a block size of 64 KByte and for writes with 128 KBytes.

- Starting with a block size of 1024 KByte for I/O access about 120 MByte/sec of throughput is measured for TAS and 5 clients (see figures 7.8 and 7.10). This throughput is close to the servers network bandwidth limit of 125 MByte/sec. The block size is very similar for read and write operations.

- Throughput using a variable client count stabilizes for writes between 80 and 90 MByte/sec while it oscillates for reads (see figures 7.13 and 7.14). Measuring the server's load during the test shows that the server is busy processing the requests.

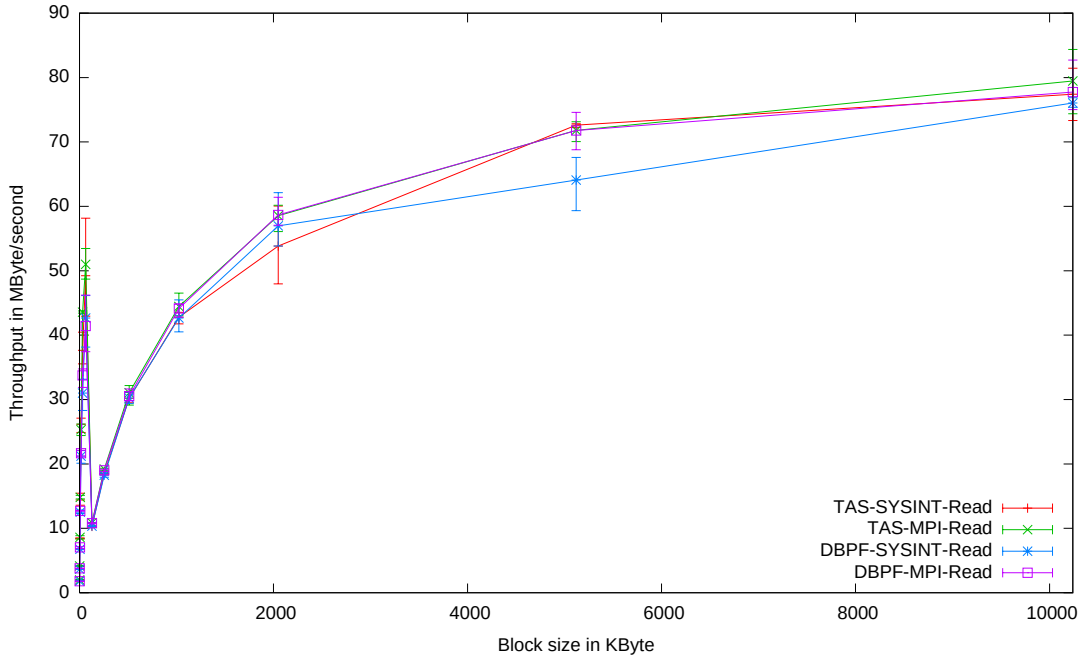

Figure 7.6: Comparison of MPI and System interface read throughput for 1 client using different block sizes

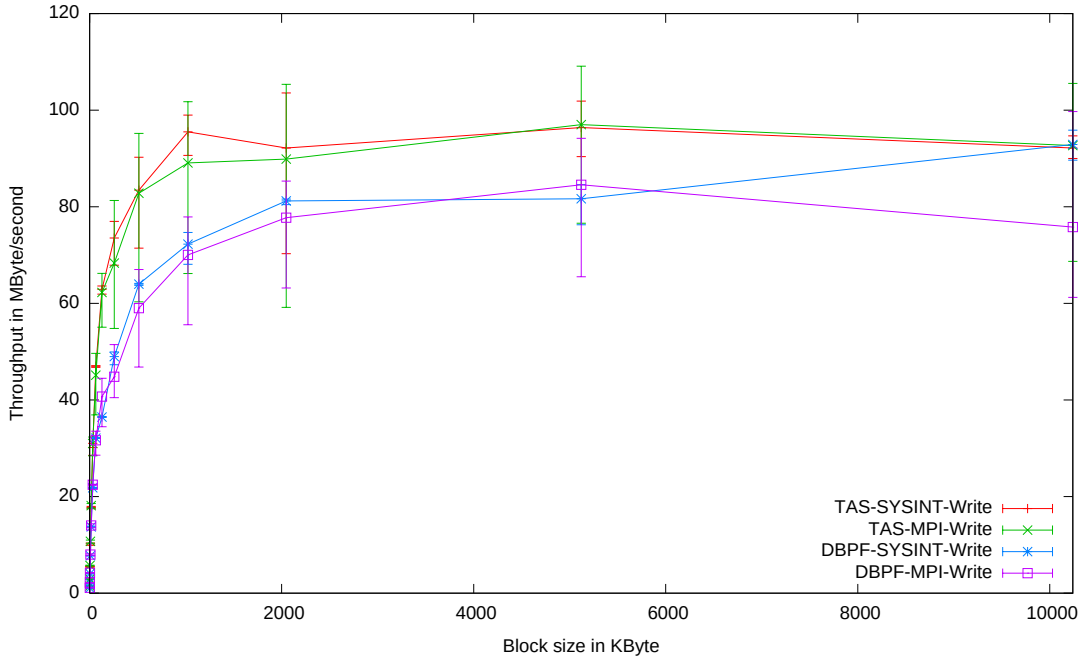

Figure 7.7: Comparison of MPI and System interface write throughput for 1 client using different block sizes

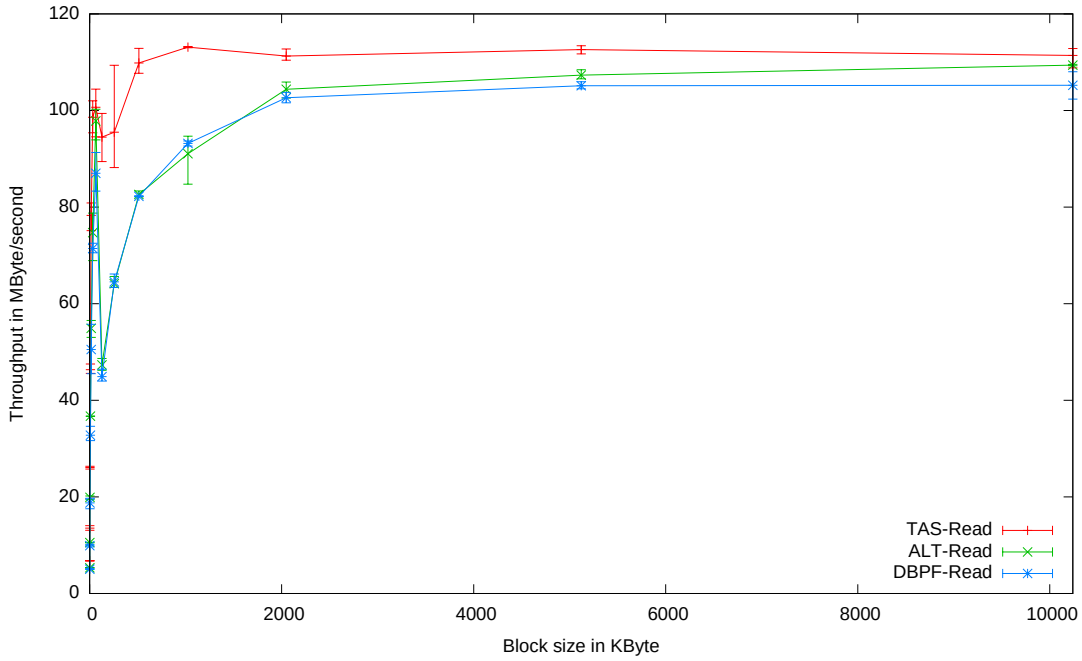

Figure 7.8: Read throughput for 5 clients using different block sizes

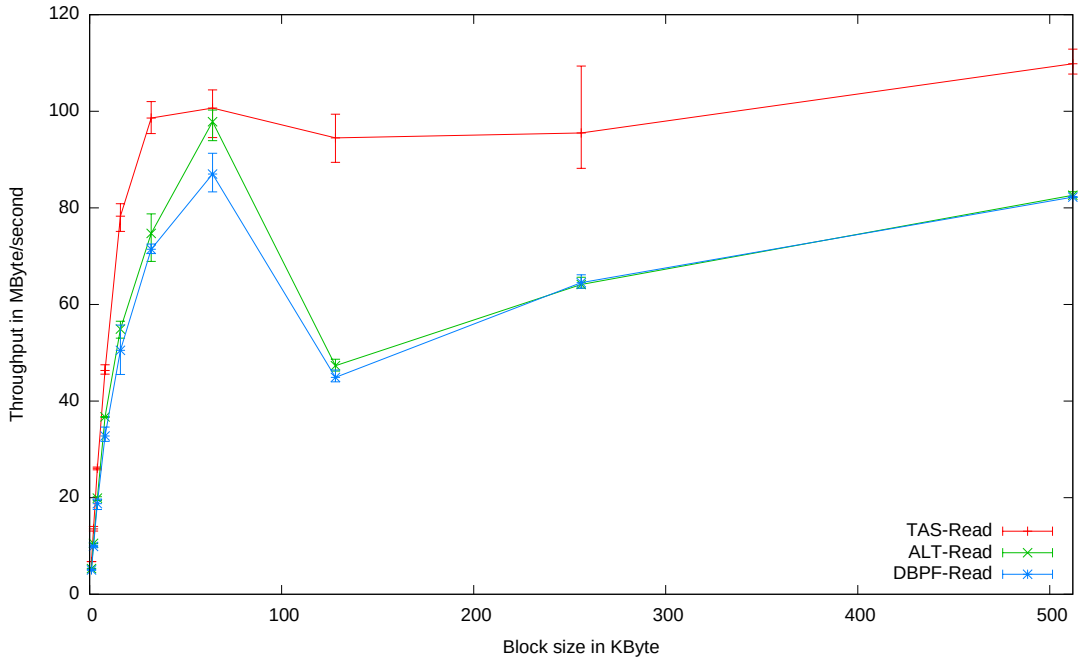

Figure 7.9: Read throughput for 5 clients using different small block sizes (data extracted from figure 7.8)

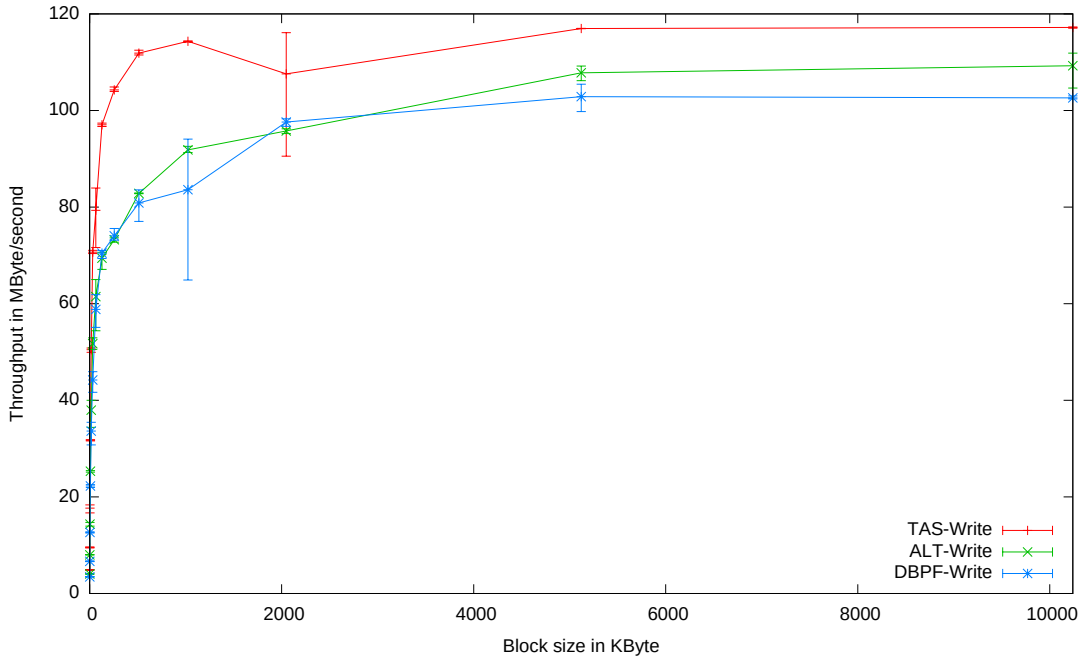

Figure 7.10: Write throughput for 5 clients using different block sizes

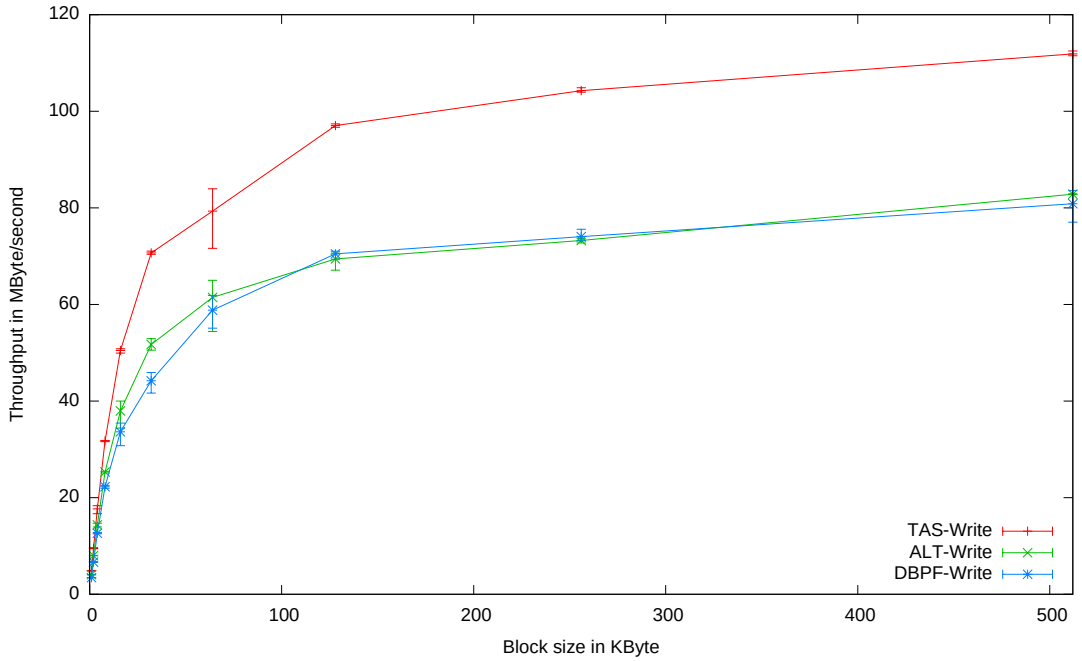

Figure 7.11: Write throughput for 5 clients using different small block sizes (data extracted from figure 7.10)

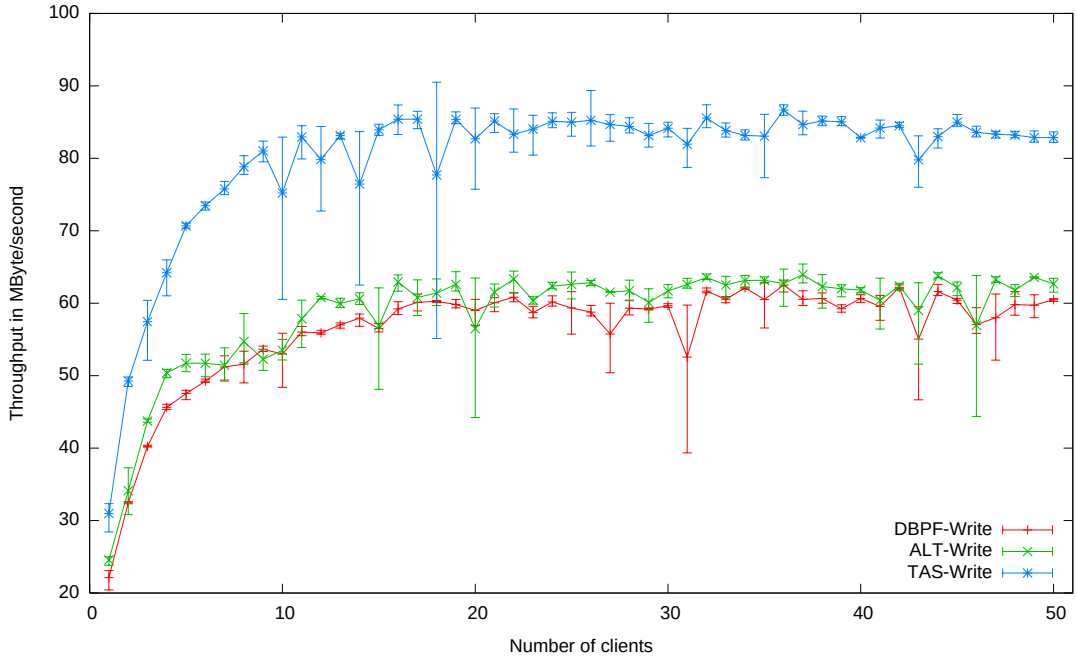

Figure 7.12: Write throughput for a variable client number and 32 KByte block size

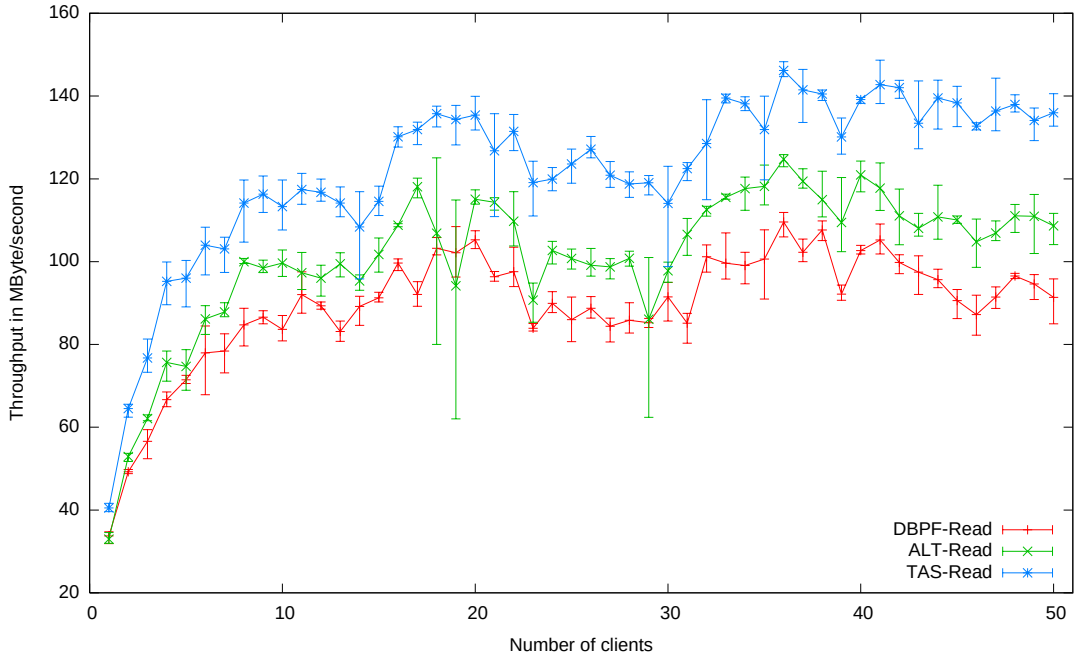

Figure 7.13: Read throughput for a variable client number and 32 KByte block size

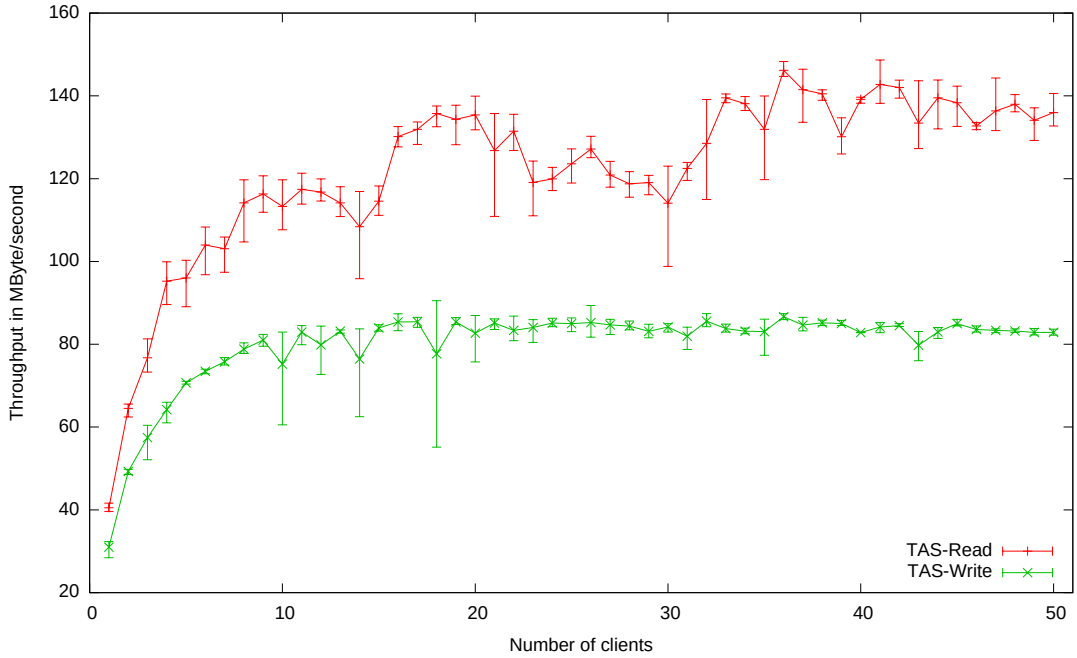

Figure 7.14: Comparison of read and write for a variable client number and 32 KByte block size (data extracted from figures 7.12 and 7.13)

7.1.2 One metadata and five data server

For this configuration only the MPI interface is analyzed. A comparison between one and five data servers is made in the next subsection.

Observations:

- The performance cut for read now shifts to 512 KBytes, which is exactly the data server count times 128 KByte (see figures 7.15, 7.16, 7.19 and 7.20). Remember, the requests to different data servers are processed in parallel.
- Throughput of ALT and DBPF is close to the bound provided by TAS in all cases (see figures 7.15 - 7.24). Thus, the aggregated I/O throughput matches the network throughput.
- For writes there are more than 130 MByte/sec measured which is higher than the client's network bandwidth (see figure 7.17). The clients benefit from the locality of a data server.
- Read performance is better than write performance for small block sizes and worse for large block sizes (see figures 7.15 - 7.22).
- The maximum aggregated network throughput of 5 network cards is not achieved for 5 clients and servers. A maximum throughput of 420 MByte/sec is measured for writes (see figures 7.19 and 7.21) .
- For a variable client number the performance is similar for read and write (see figure 7.16).
- Throughput decreases a bit for a client number which is divisible by 5 (see figures 7.23 - 7.25). It shows that the time needed to finish the read or write of a client's file area varies a lot (up to 40%) in this case, depending on the rank of the client. For the other test cases the end times

are nearby. A reason therefore might be the interaction between the clients' access pattern and the selected data distribution.

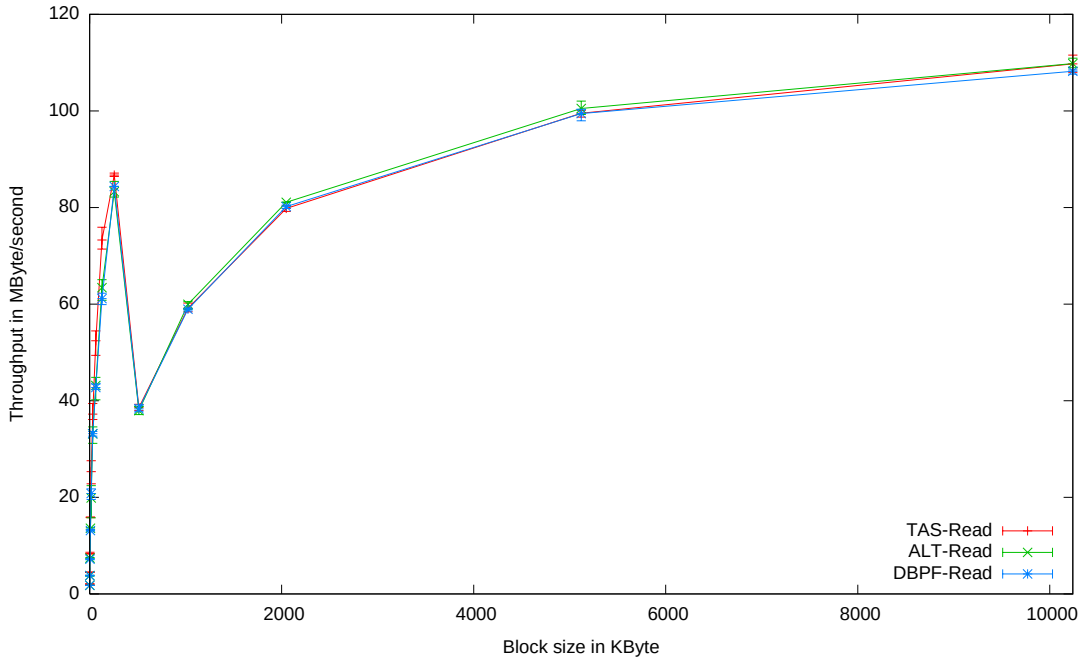

Figure 7.15: Read throughput for 1 client using different block sizes

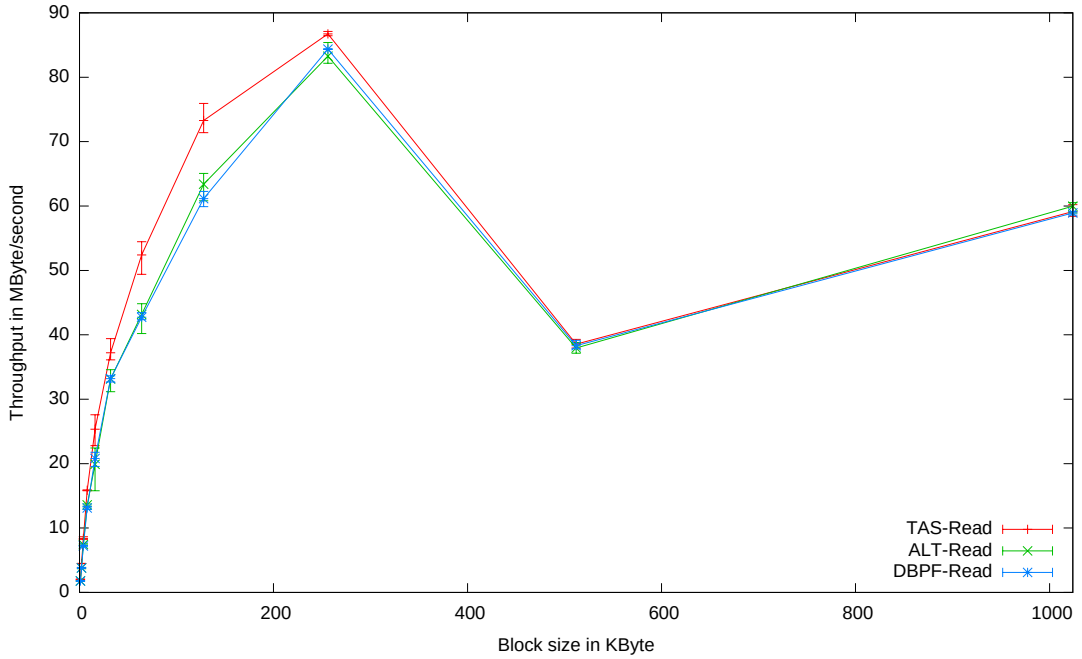

Figure 7.16: Read throughput for 1 client using different small block sizes (data extracted from figure 7.15)

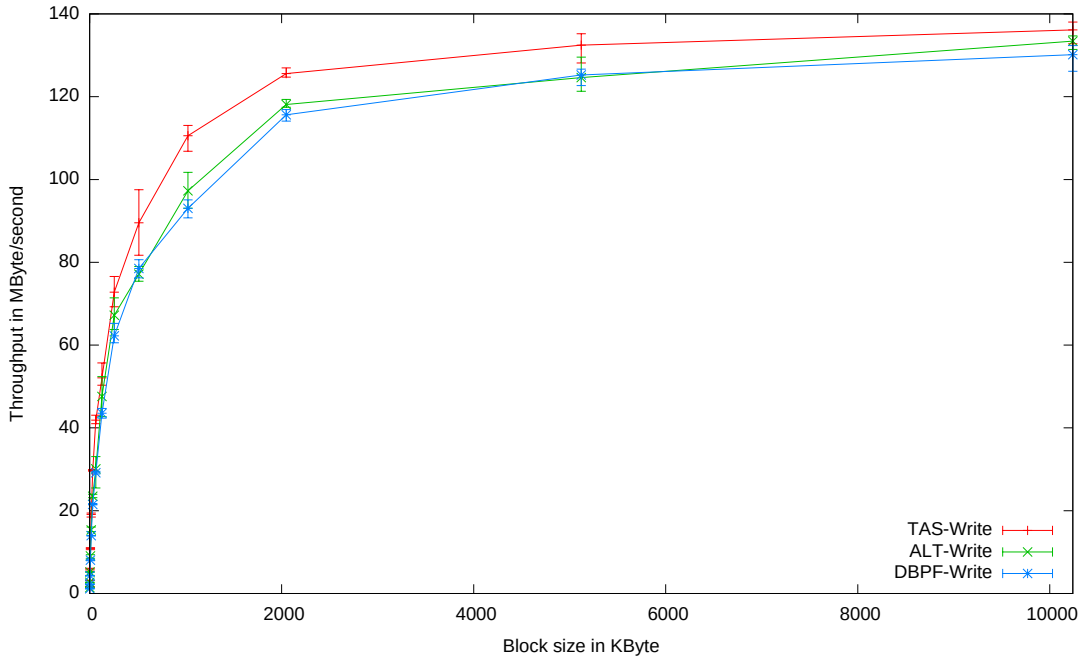

Figure 7.17: Write throughput for 1 client using different block sizes

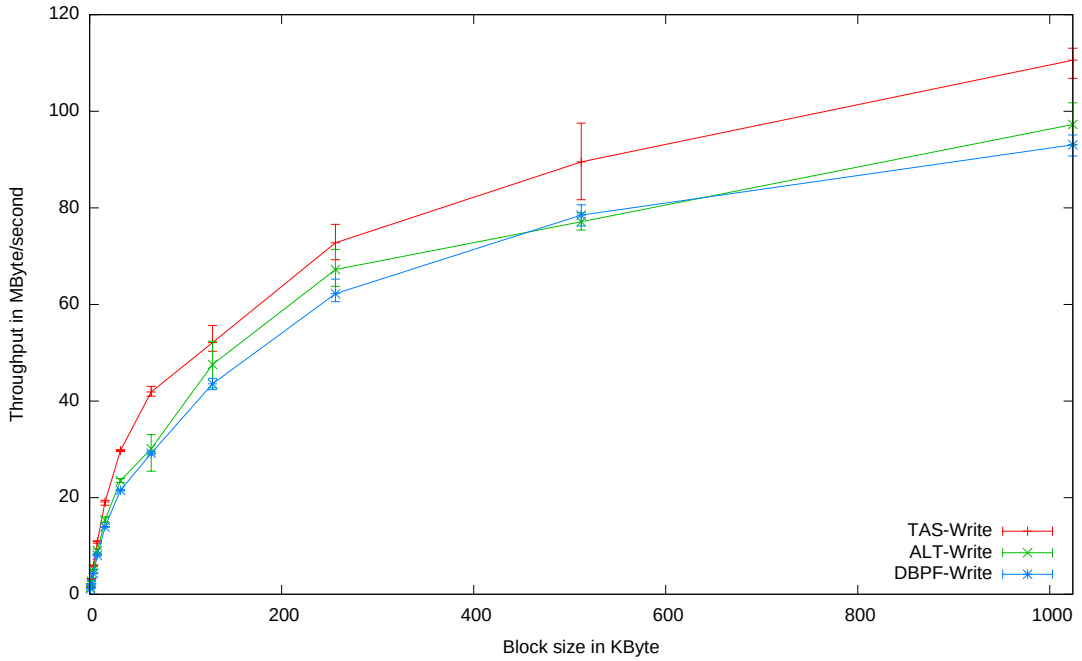

Figure 7.18: Write throughput for 1 client using different small block sizes (data extracted from figure 7.17)

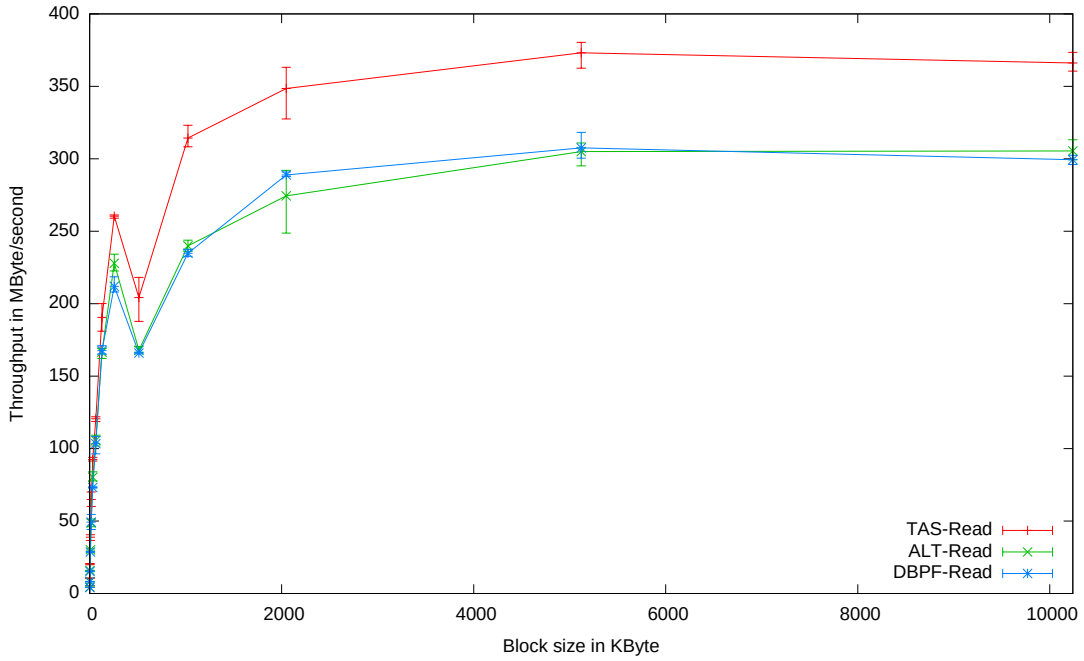

Figure 7.19: Read throughput for 5 clients using different block sizes

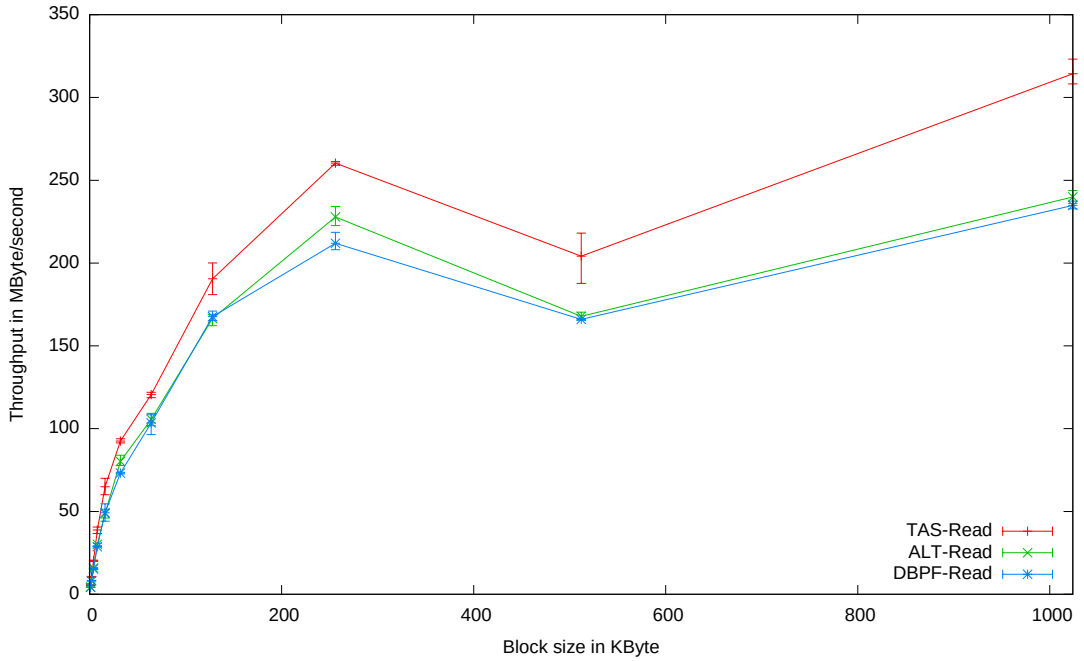

Figure 7.20: Read throughput for 5 clients using different small block sizes (data extracted from figure 7.19)

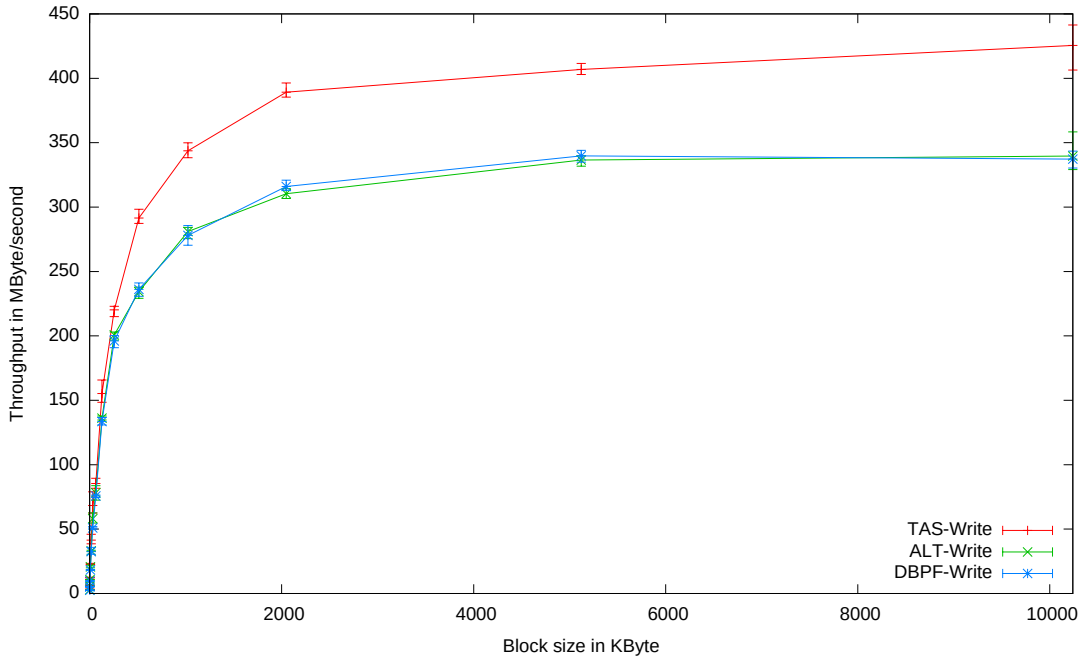

Figure 7.21: Write throughput for 5 clients using different block sizes

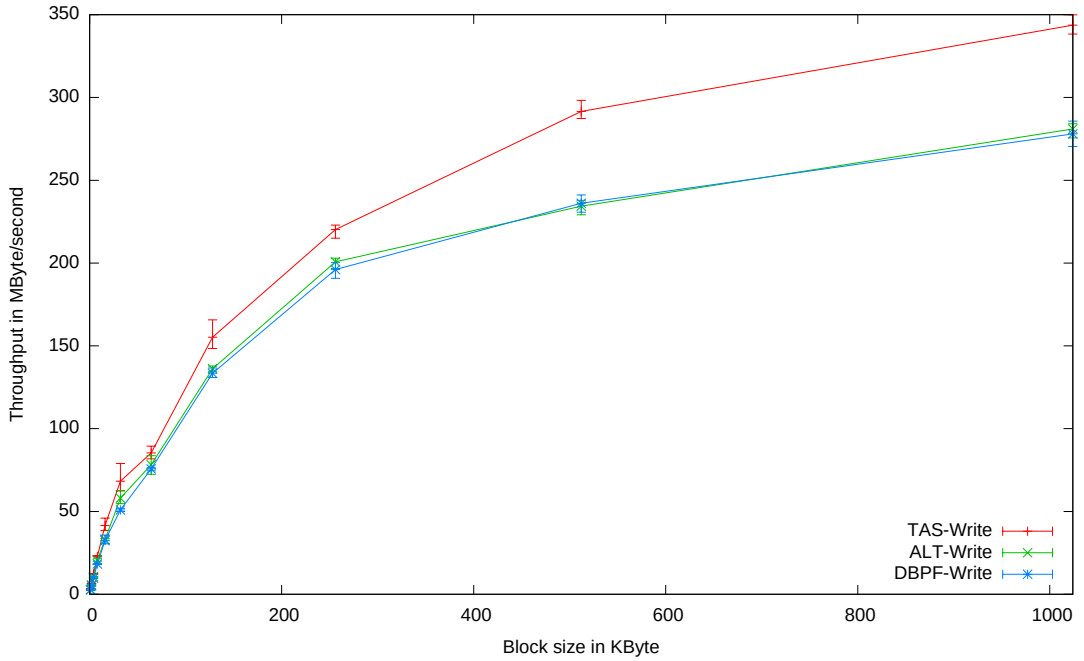

Figure 7.22: Write throughput for 5 clients using different small block sizes (data extracted from figure 7.21)

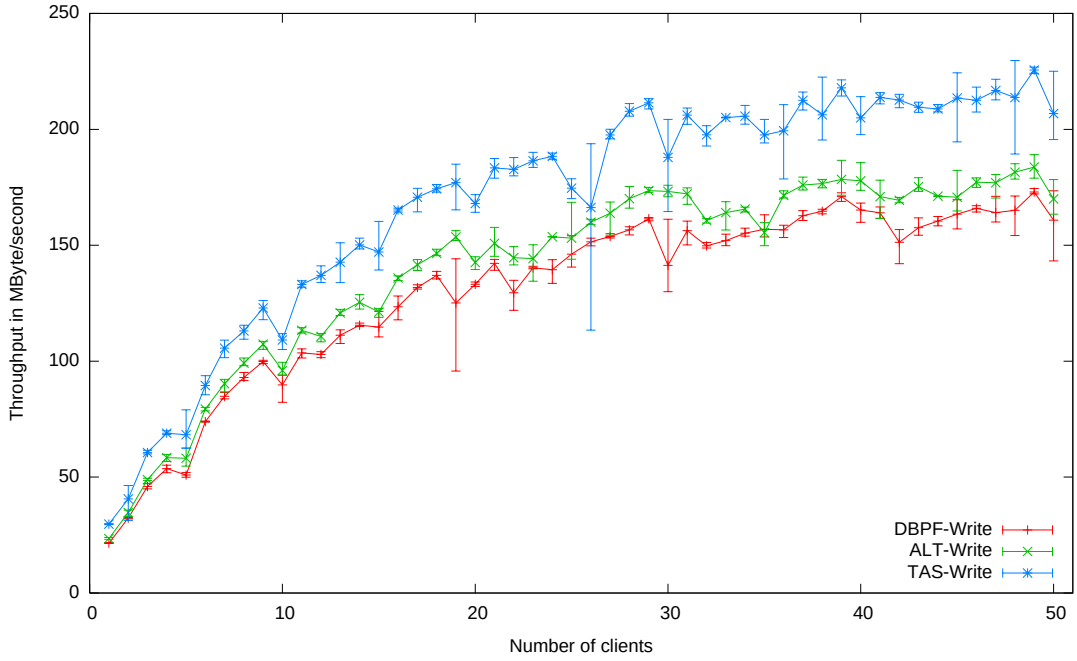

Figure 7.23: Write throughput for a variable client number and 32 KByte block size

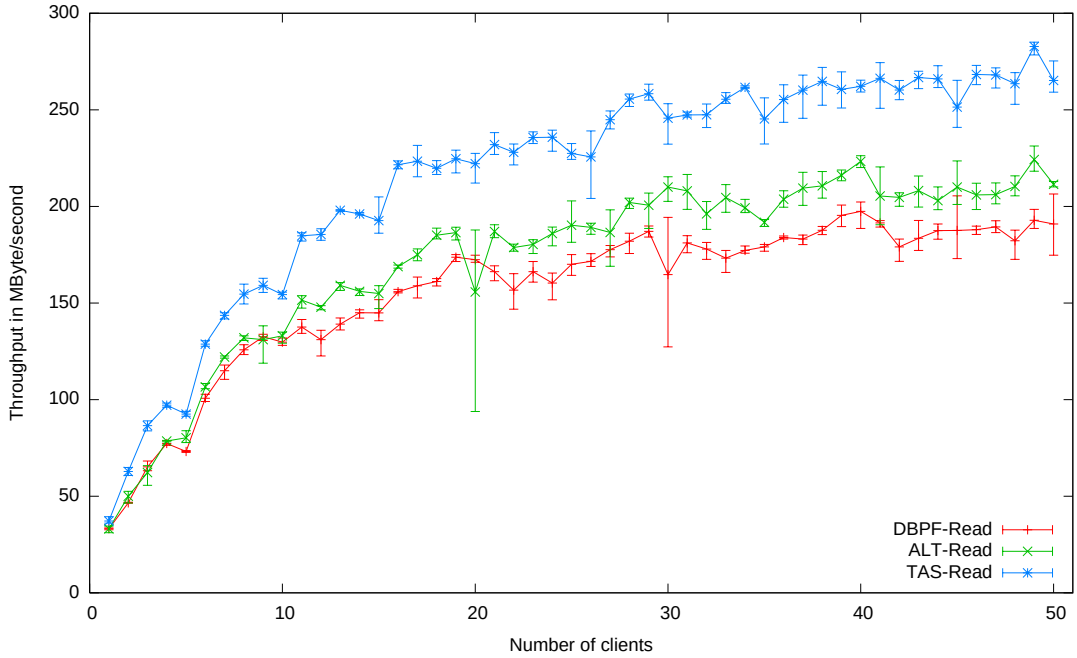

Figure 7.24: Read throughput for a variable client number and 32 KByte block size

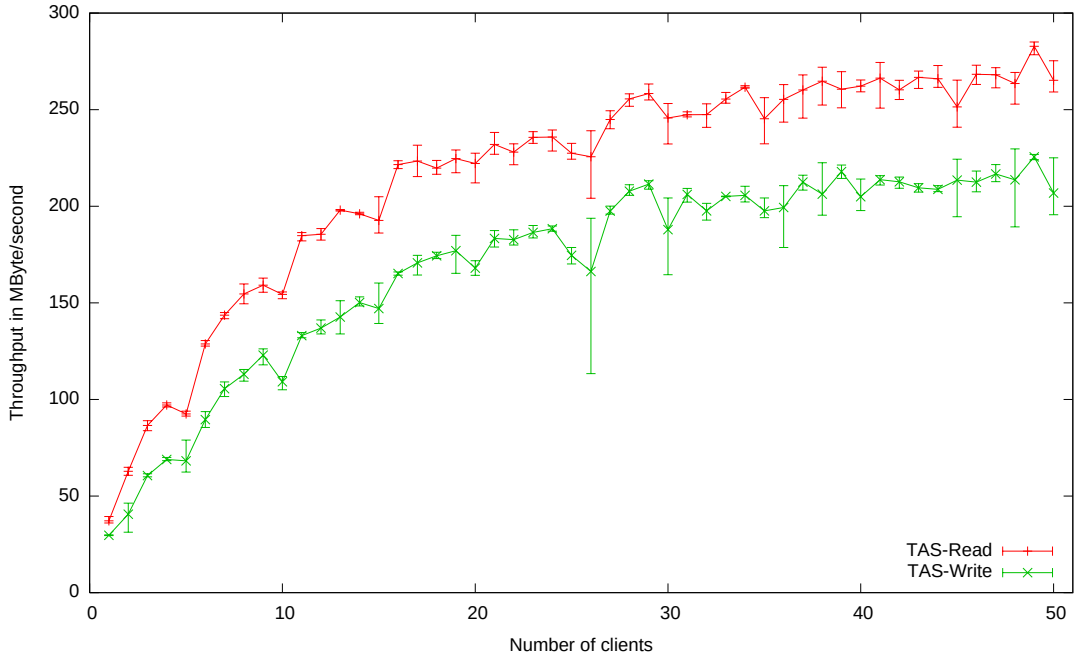

Figure 7.25: Comparison of read and write throughput for a variable client number and 32 KByte block size (data extracted from figures 7.23 and 7.24)

7.1.3 Comparison of the throughput for one and five data servers

The diagrams in the earlier section showed that ALT and DBPF perform similar using small block sizes and a 100 MByte file. In order to increase the understandability the results of ALT are omitted in the diagrams of this subsection.

Observations:

- Throughput improves by the factor 3.5 by switching to 5 data servers (see figures 26 and 29).
- The benefit from multiple servers degrades for smaller block sizes (see figures 27 and 30). For a smaller block size than 32 KByte the single server ist faster (see figures 28 and 31). This must be the consequence of the selected data distribution. Remember, the data is striped in 64 KByte blocks, thus there is only one data server hit for smaller block sizes. All the datafiles will be used for a block size larger than 256 KByte.
- In case of a variable client number and 32 KByte block size, read and write throughput fluctuates in the same way for 5 data servers while write performance stabilizes for 1 data server (see figure 32 and 33) .
- Up to 5 clients, the throughput for 1 and 5 data servers and a variable client number look similar (see figures 32 and 33). Starting with 6 clients, the aggregated throughput is higher for the configuration with 5 data servers. Note that for the small block size only one datafile should be hit per request due to the data distribution and data alignment. The throughput for 5 data servers improves for multiple clients because the requests are distributed between the data servers. Thus, the load is reduced per server. However, the throughput only improves by a factor of 3 for write operations and a factor of 2 for read operations.

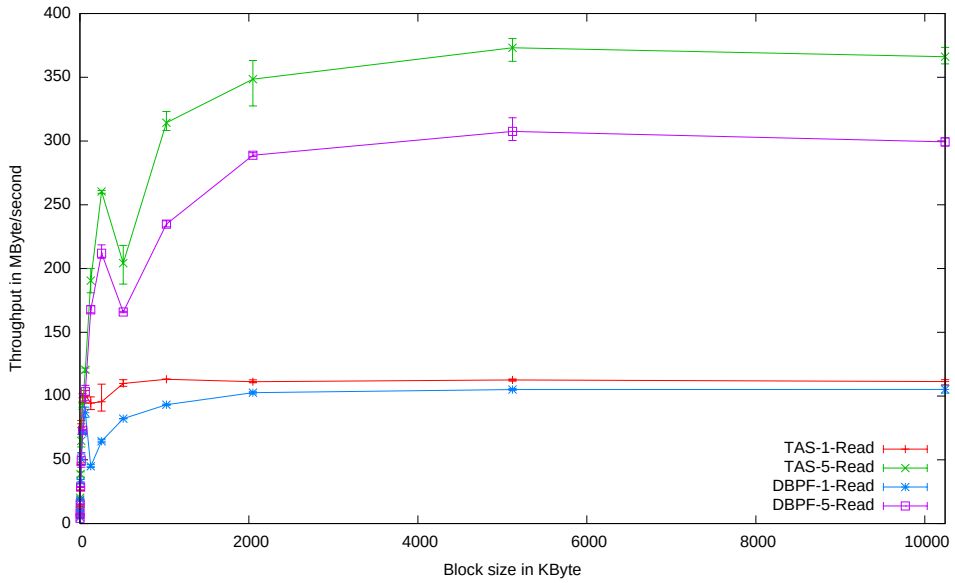

Figure 7.26: Comparison of the read throughput for 1 and 5 data servers accessed by 5 clients using different block sizes

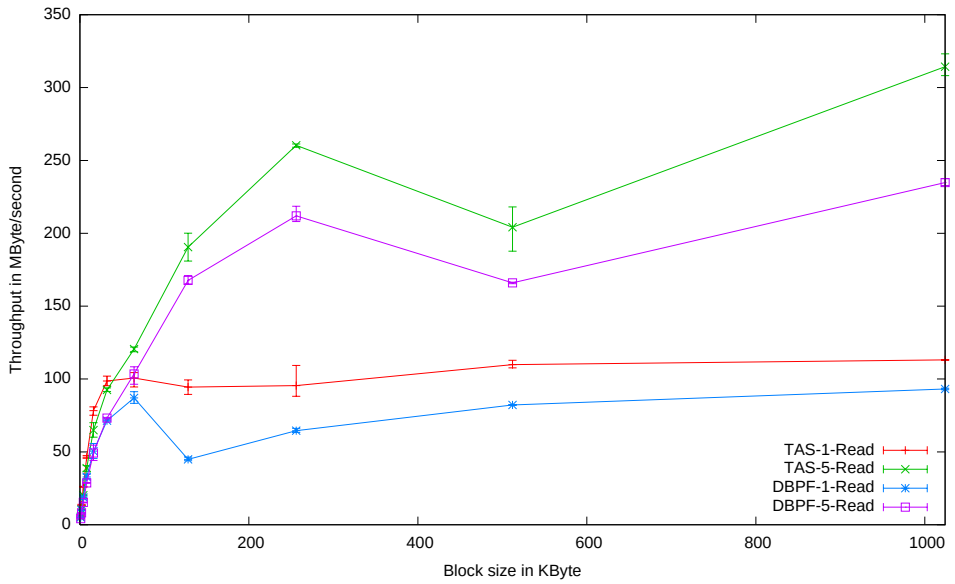

Figure 7.27: Comparison of the read throughput for 1 and 5 data servers accessed by 5 clients using different small block sizes (data extracted from figure 7.26)

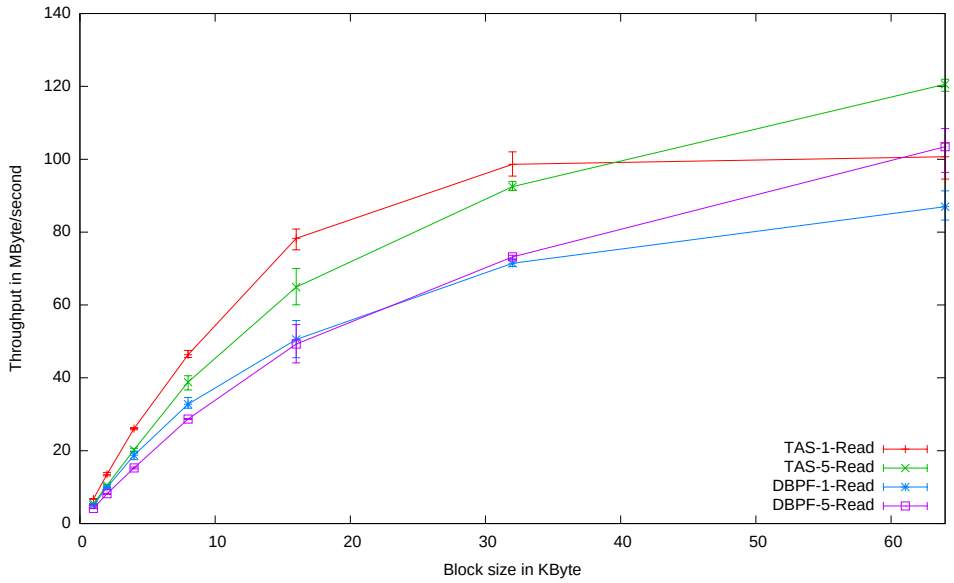

Figure 7.28: Comparison of the read throughput for 1 and 5 data servers accessed by 5 clients using different very small block sizes (data extracted from figure 7.26)

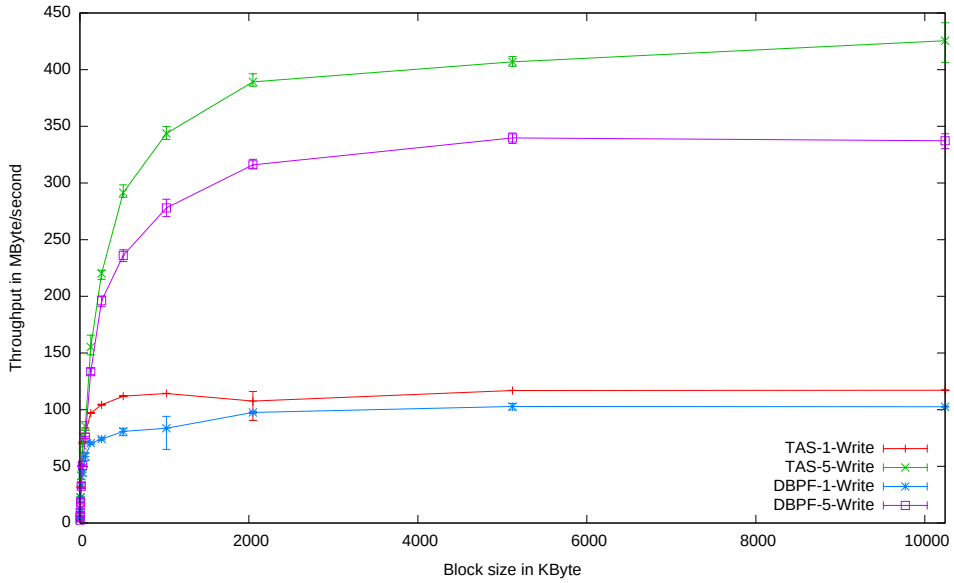

Figure 7.29: Comparison of the write throughput for 1 and 5 data servers accessed by 5 clients using different block sizes

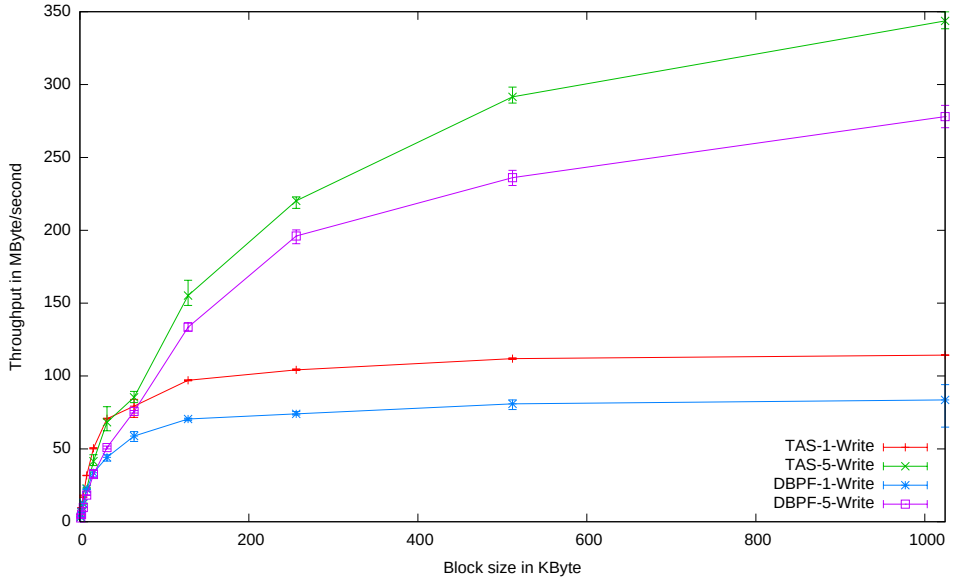

Figure 7.30: Comparison of the write throughput for 1 and 5 data servers accessed by 5 clients using different small block sizes (data extracted from figure 7.29)

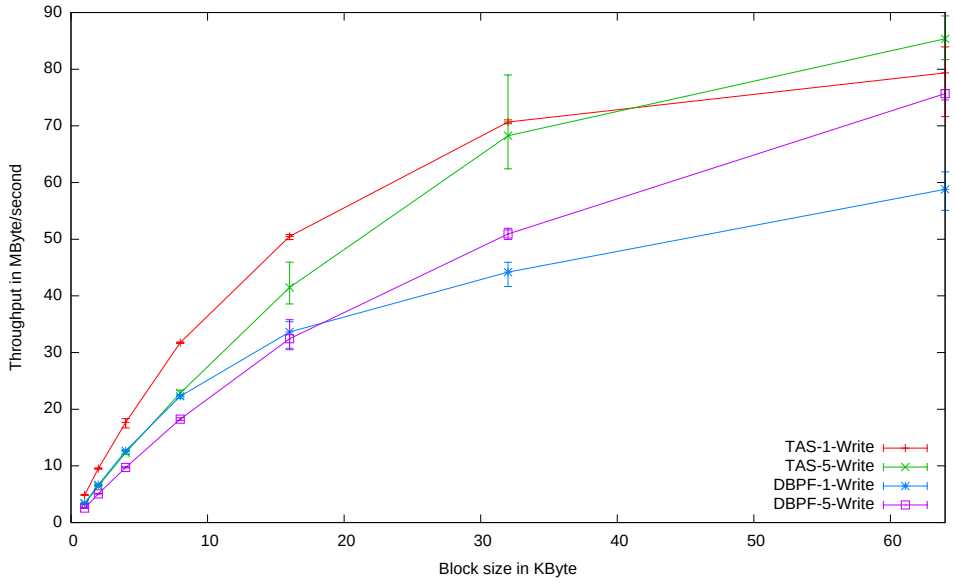

Figure 7.31: Comparison of the write throughput for 1 and 5 data servers accessed by 5 clients using different very small block sizes (data extracted from figure 7.29)

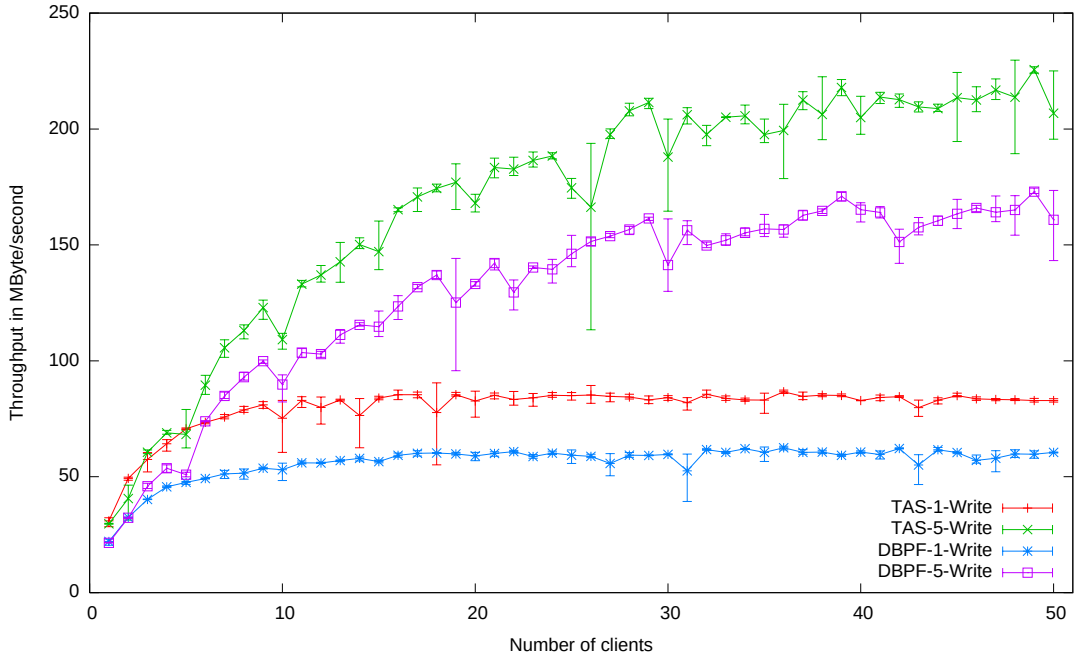

Figure 7.32: Comparison of the write throughput for 1 and 5 data servers accessed by a variable client number and 32 KByte block size

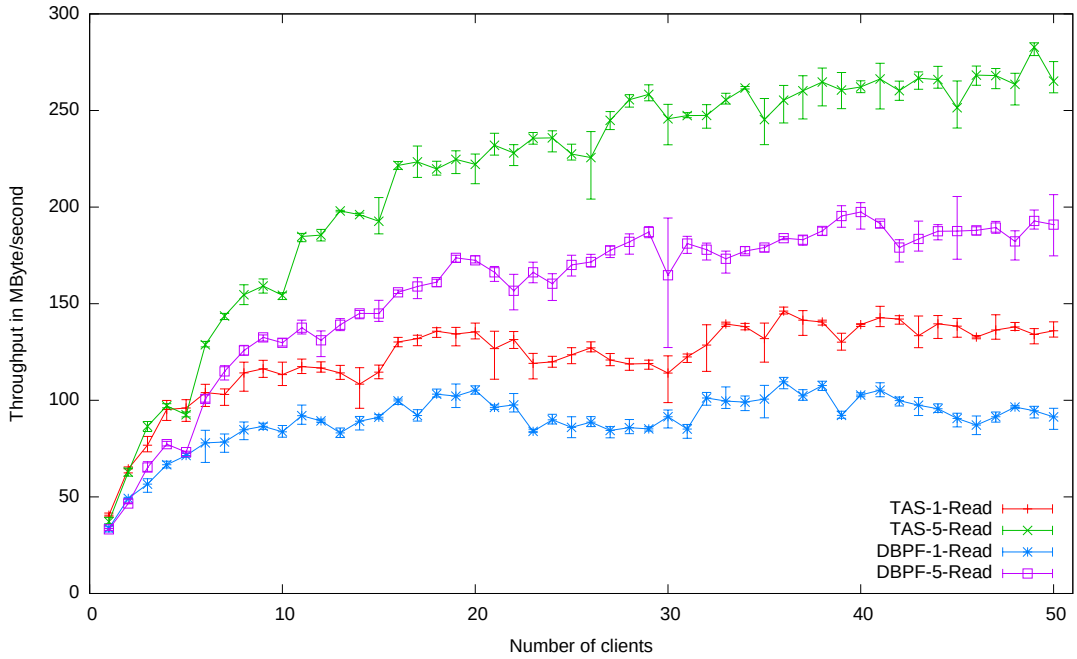

Figure 7.33: Comparison of the read throughput for 1 and 5 data servers accessed by a variable client number and 32 KByte block size

7.1.4 Access of large files with small contiguous I/O requests

The former diagrams show the throughput of only 100 MByte files. In order to reduce the impact of the kernel buffer this testset is created. A 3200 MByte file is accessed with a block size of 32 KByte. The configuration of one metadata and one dataserver is used.

Observations:

- The write throughput is close to the capabilities of the I/O subsystem (see figure 7.35). It seems that the write operations are cached efficiently. Also, multiple small blocks might be combined into one contiguous request to avoid a repositioning of the disk's heads.

- Throughput of ALT is comparable to DBPF (see figures 7.34 and 7.35). This is surprising because ALT uses a straightforward way for file access while DBPF uses a more complicated strategy. For example asynchronous I/O comes into play for DBPF and the file handles are buffered.

- Read throughput is much slower than write and degrades with additional clients (see figure 7.34) . The benefit of the prefetching strategy depends on the client number. Now, most of the accesses require the hard disk's head to be relocated. If you assume that the bytestream is written as one contiguous block on the disk, then the distance of the necessary block depends on the client number. However, it is hard to predict the behavior exactly because multiple clients might proceed with a different speed.

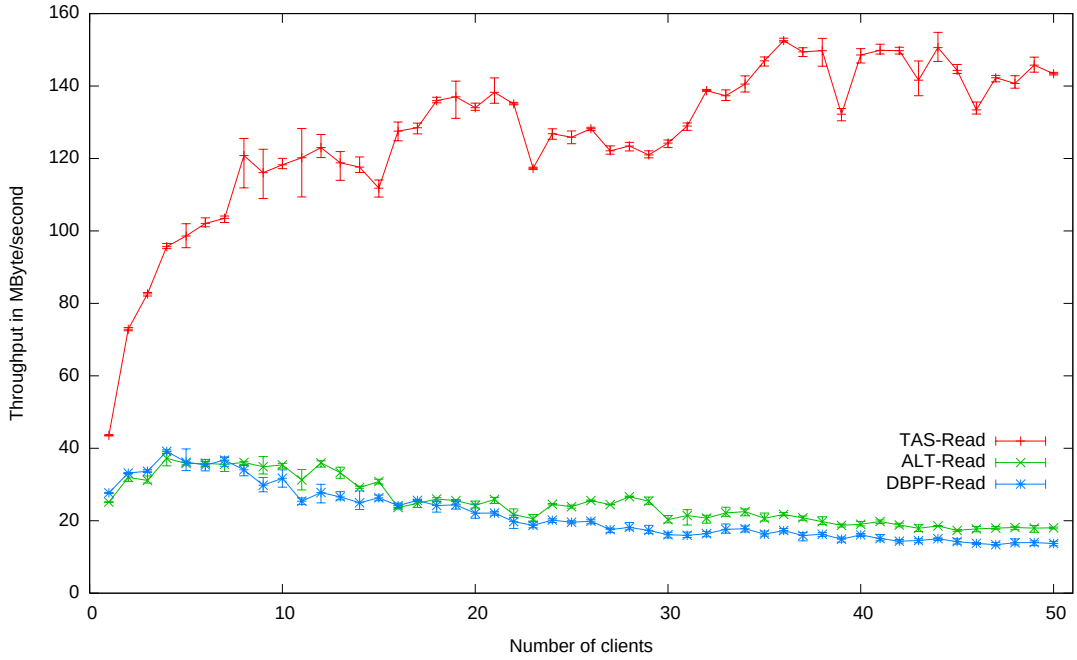

Figure 7.34: Read throughput for a variable client number and 32 KByte block size (access of a 3200 MByte file)

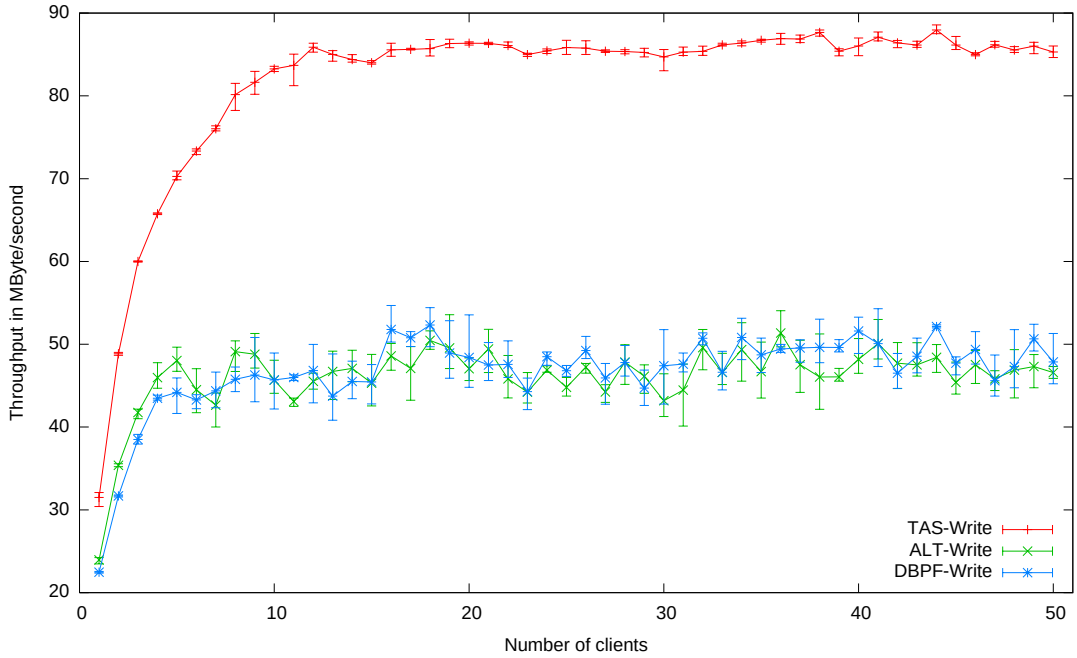

Figure 7.35: Write throughput for a variable client number and 32 KByte block size (access of a 3200 MByte file)

Summary

This paragraph summarizes the results of small contiguous I/O requests. The following results are important:

- The performance of the ALT straightforward I/O implementation is not inferior to the DBPF asynchronous I/O implementation. This is surprising but might be a consequence of the chosen test cases.

- For most cases the throughput for TAS is close to the estimated upper bound derived from netperf. There is some room for improvement for 5 data servers. However, further benchmarks and investigation is necessary to determine bottlenecks for multiple servers.

- Caching mechanisms work well for these test cases. The aggregated throughput is higher than the I/O subsystem's nominal throughput. The throughput of a large file breaks in for read requests, due to the disk's access time induced by the mechanical components of the disk. Clever strategies are necessary to improve the situation in this case.

- The data distribution function influences the performance for small block sizes. It is suggested that the influence of the data distribution is negligible for larger contiguous requests.

- Multiple servers achieve a smaller aggregated throughput for small contiguous requests than one server. It is important to select a distribution function suitable for the applications requests in order to get the maximum performance.

7.2 Large Contiguous I/O Requests

In this testset big files which have a size between 100 and 12800 MByte are accessed with a block size of 10 MByte, which means each system interface call starts a data transfer for 10 MByte of data. Remember, during an I/O request a maximum of 256 KByte is processed by the persistency layer during one call. However, the setup of an I/O operation needs only little time compared to the data transmission. The actual file size is a power of 2 times 100 MByte for a specific test. The testset shows the impact of the cache. However, files grow larger than the cache size. It is important for the file system to achieve a overall throughput equal to the disk throughput for large I/O requests because the disk limits the performance. This testset measures the throughput only for the MPI interface. Only the throughput of the network card and I/O subsystem are considered for the estimated upper bounds. Therefore, these bounds are incorporated directly into some diagrams.
Large contiguous I/O requests are analyzed for the following configurations:

- **One metadata server and one data server**
  First, throughput for one client is measured. The client creates a file, writes a specific amount of data into the file and then reads the data again.
  Then, the aggregated throughput is measured for five clients and different file sizes.
  At last, a variable number of clients is run to access a big file with MPI to determine the server's behavior and maximum throughput under a high load.

- **One metadata server and five data server**
  The tests run for this configurations are the same as for the configuration with only one data server in order to be able to compare the results.
  The results for the configurations with one data server and five data servers are not directly comparable because the clients are disjoint for one data server while client and server share the machines for five data servers.

7.2.1 One metadata and one data server

Starting with 1600 MByte the DBPF-thread hang up randomly during the write for DBPF. This happens, for example, if node06 is the server. Using the machine master2 as PVFS2-server avoids this problem. The problem remain for five data servers. However, the reason is unknown, several tries to reproduce it on different machines failed.

Observations:

- Write performance is a bit better than read (see figures 7.36, 7.37 and 7.42). Especially the achievable throughput with TAS is better.

- The kernel buffers the files up to 800 MByte well which is nearly the available memory (see figure 7.36). It looks like the whole file is in the cache because the read performance sticks at the same value up to 800 MByte. The write throughput decreases for 400 MByte (see figure 7.37), it might be the case that the kernel blocks some write operations while the cache gets filled.

- Starting with two clients the network card of the server is nearly saturated (see figure 7.42). The throughput is close to the available network bandwidth.

- ALT achieves the throughput of the I/O subsystem and performance does not degrade for an increasing number of the clients (see figures 7.38, 7.39 and 7.42).

- DBPF slows down for multiple clients. For write this starts with 10 clients (see figure 7.40). There is a dramatic cut for read and 2 clients (see figures 7.38 and 7.41). There might be a problem with the asynchronous I/O on our machines because the machines load is asymmetric for read and write. During the write phase the load is about 10 while for reads it is about 2.5.

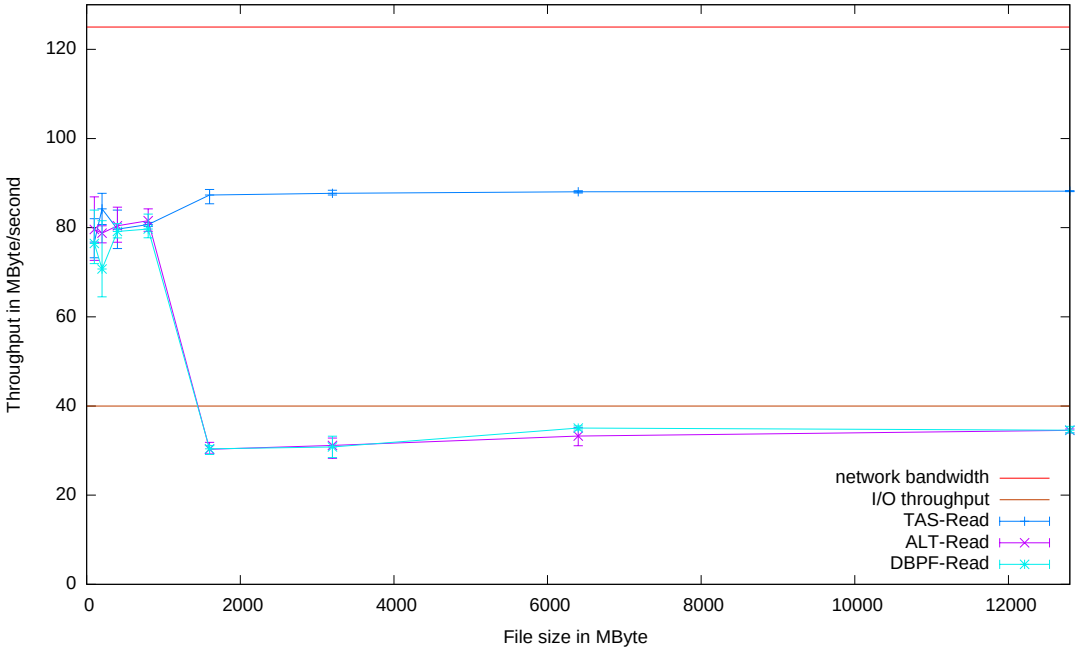

Figure 7.36: Read throughput for 1 client accessing a large file with a varying size

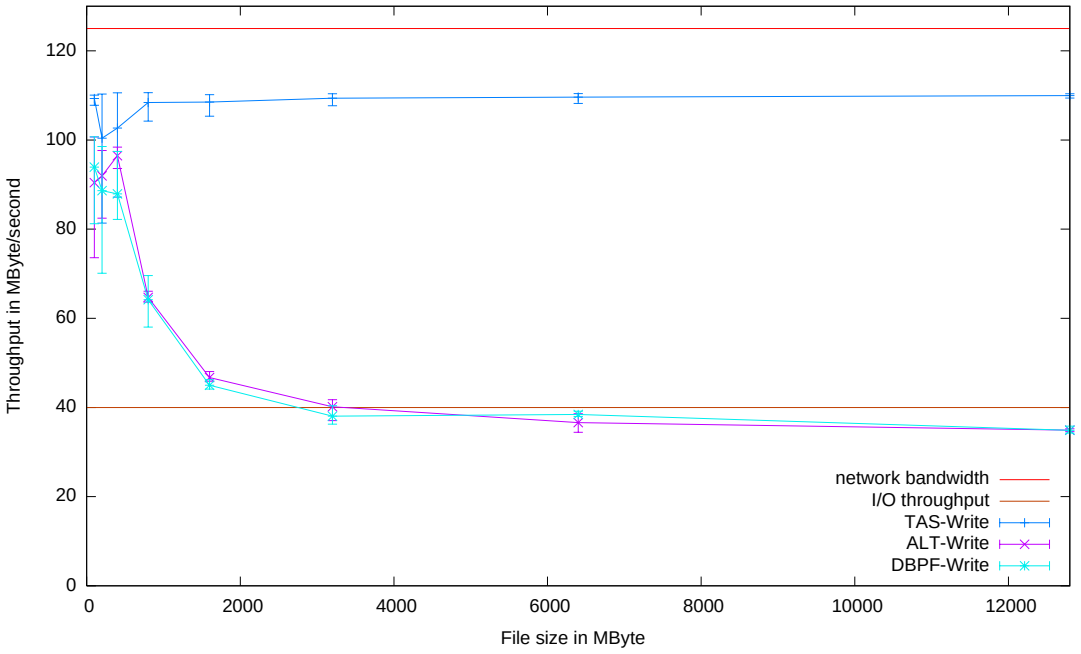

Figure 7.37: Write throughput for 1 client accessing a large file with a varying size

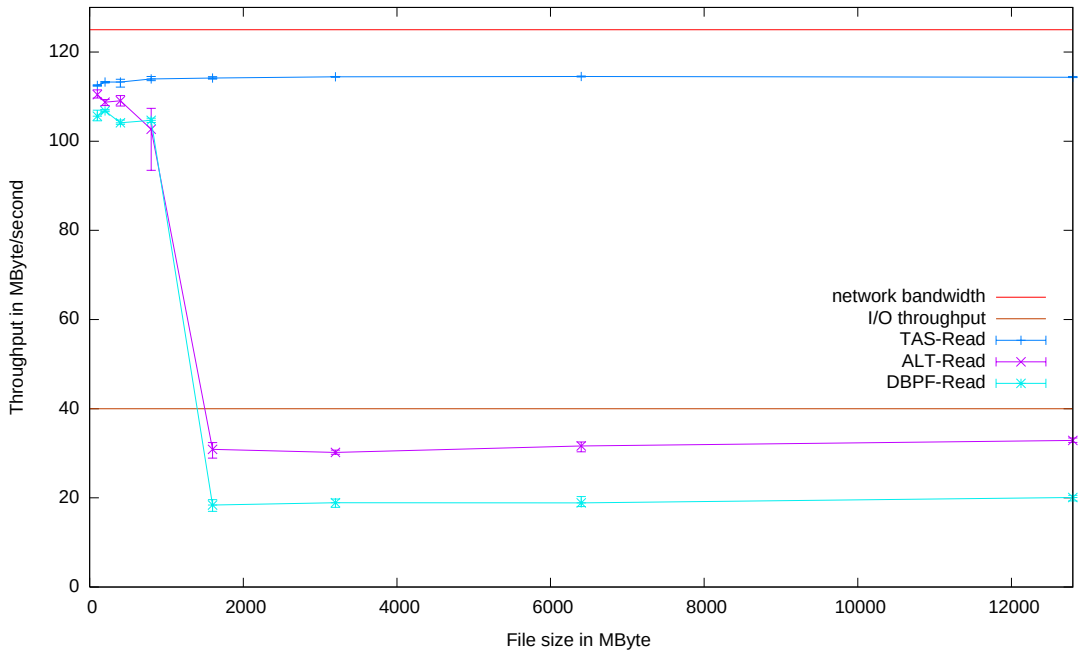

Figure 7.38: Read throughput for 5 clients accessing a large file with a varying size

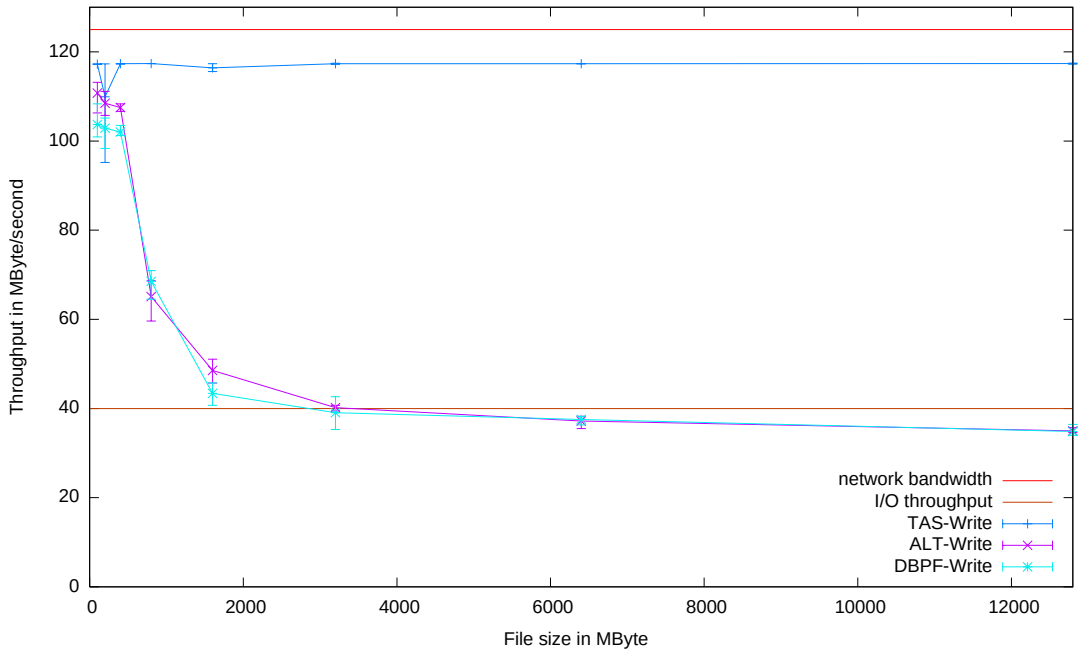

Figure 7.39: Write throughput for 5 clients accessing a large file with a varying size

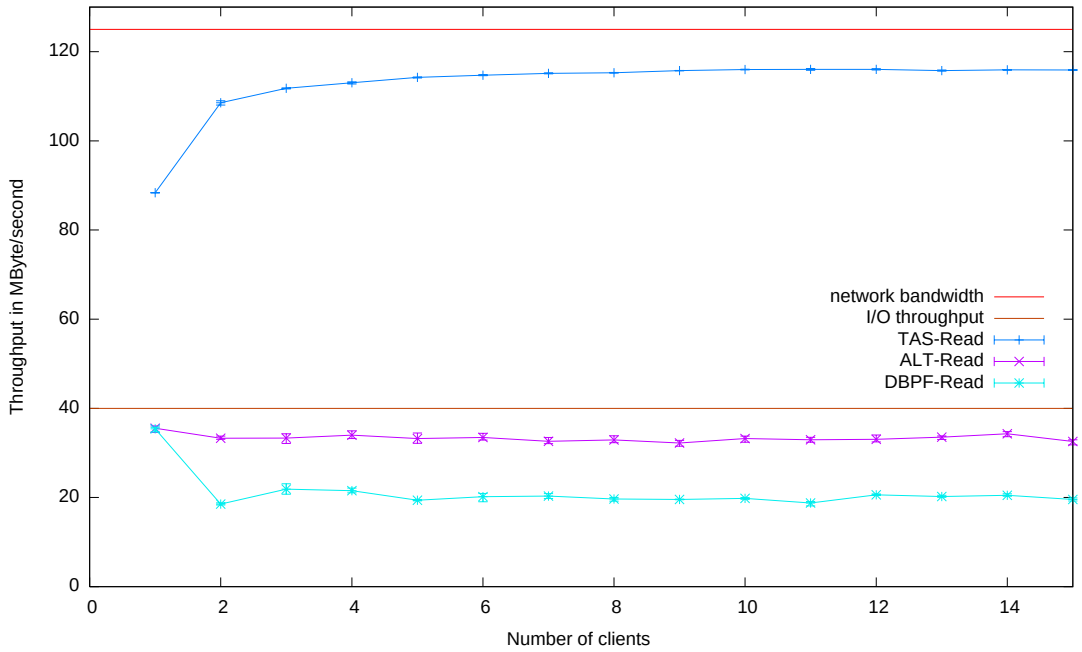

Figure 7.40: Read throughput for a variable client number accessing a large file with a total size of 12800 MByte

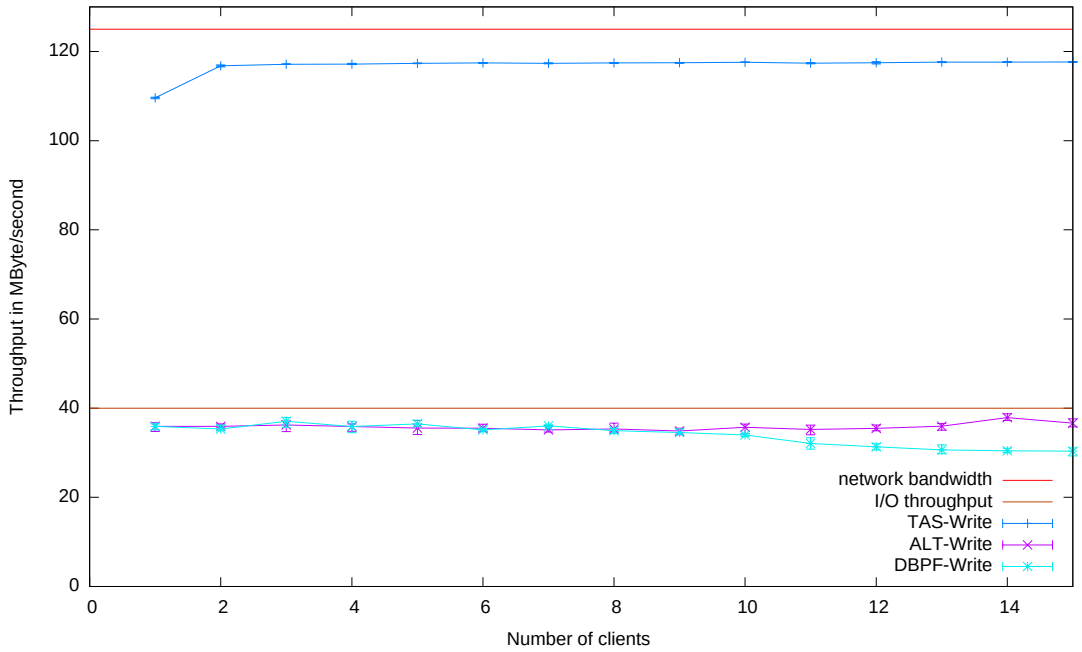

Figure 7.41: Write throughput for a variable client number accessing a large file with a total size of 12800 MByte

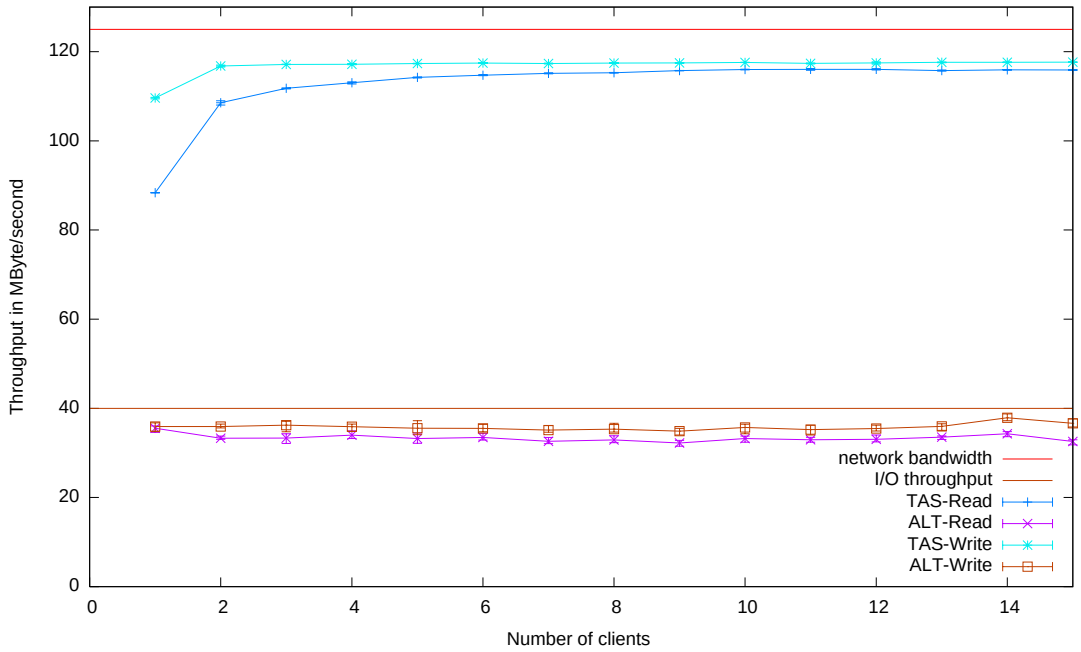

Figure 7.42: Comparison of read and write throughput for a variable client number accessing a large file with a total size of 12800 MByte

7.2.2 One metadata and five data server

The PVFS2 server's DBPF thread hung up for files with a size larger than 1600 MByte. Therefore, most DBPF results are missing for this configuration.

Observations:

- Real I/O for one client is now close to the limit provided by TAS (see figure 7.43 and 7.44). DBPF's throughput is a bit better for read of 1600 and 3200 MByte (see figure 7.43). Because reads are processed by TAS as fast as possible, I conclude DBPF must benefit from the additional thread processing the BMI callback function.

- The kernel's caching mechanisms lift the aggregate throughput for 5 clients (see figures 7.45 and 7.46) .

- Write is faster than read (see figure 7.47).

- For multiple clients a throughput of about 200 MByte/sec can be measured. This is equivalent to the estimated throughput of 5 disks (see figure 7.47).

- The estimated throughput of 5 network cards is not achieved (see figure 7.47). In the test cases 430 MByte/sec can be achieved for 5 clients. However, the expected throughput is 535 MByte/sec. A reason for this might be that the servers and clients are hosted on the same machines. Hence, client and server compete for the limited network and CPU time.

- Performance of write operations is higher for one client than the network bandwidth due to locality of one PVFS2 server (see figure 7.44).

- The sawtooth shape which can be seen in the diagram for multiple clients result from the

distribution of the clients over the nodes (see figure 7.47). A maximum for TAS is achieved for 5 clients, then each machine hosts the same client number. If one machine hosts two clients while the other host only one client, the network of the single machine is the bottleneck. Furthermore, CPU time is shared between client and server processes. Due to the sharing overall performance decreases for multiple clients hosted on one machine.

- It is interesting that ALT's performance is at the same level for 10 and 15 clients and lower for 5 clients (see figure 7.47). ALT's read performance fluctuates whereas the write performance is stable for a variable client number.

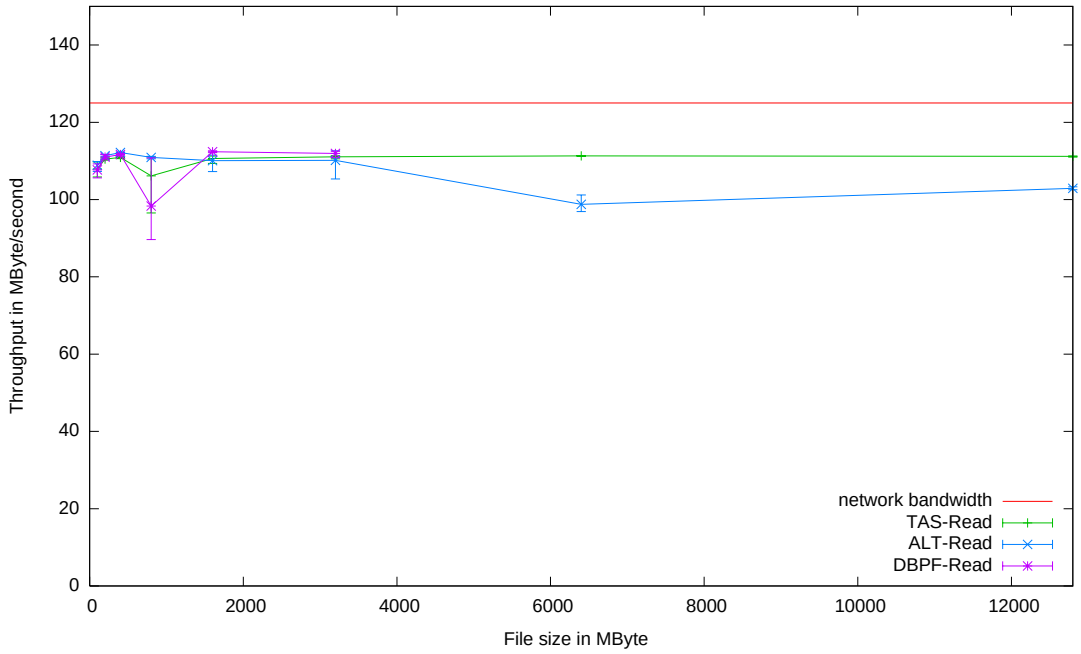

Figure 7.43: Read throughput for 1 client accessing a large file with a varying size

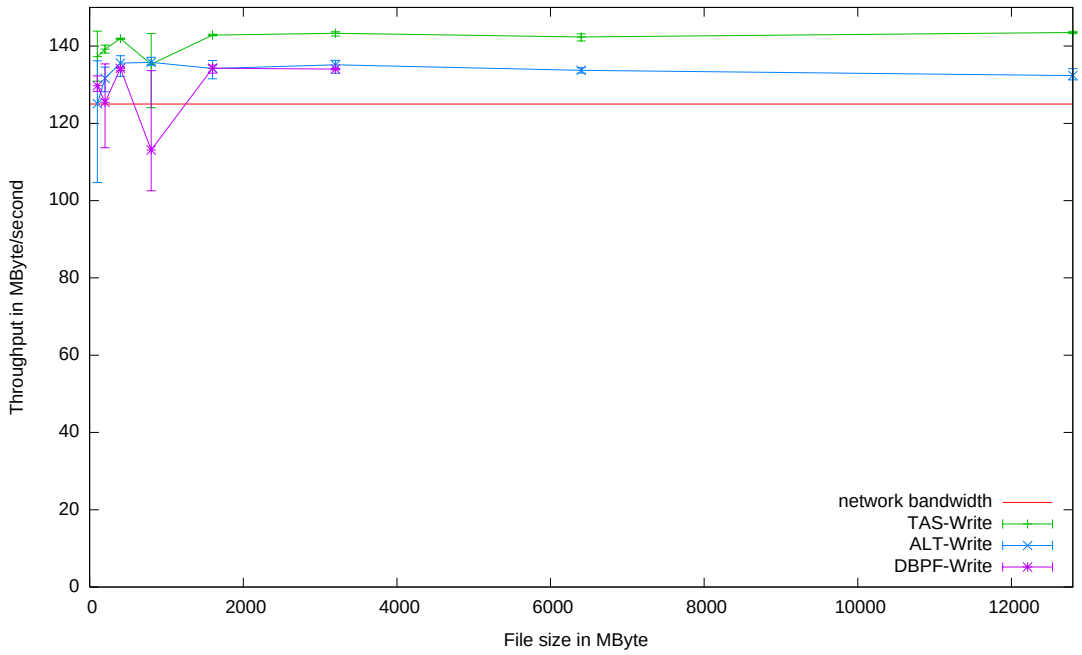

Figure 7.44: Write throughput for 1 client accessing a large file with a varying size

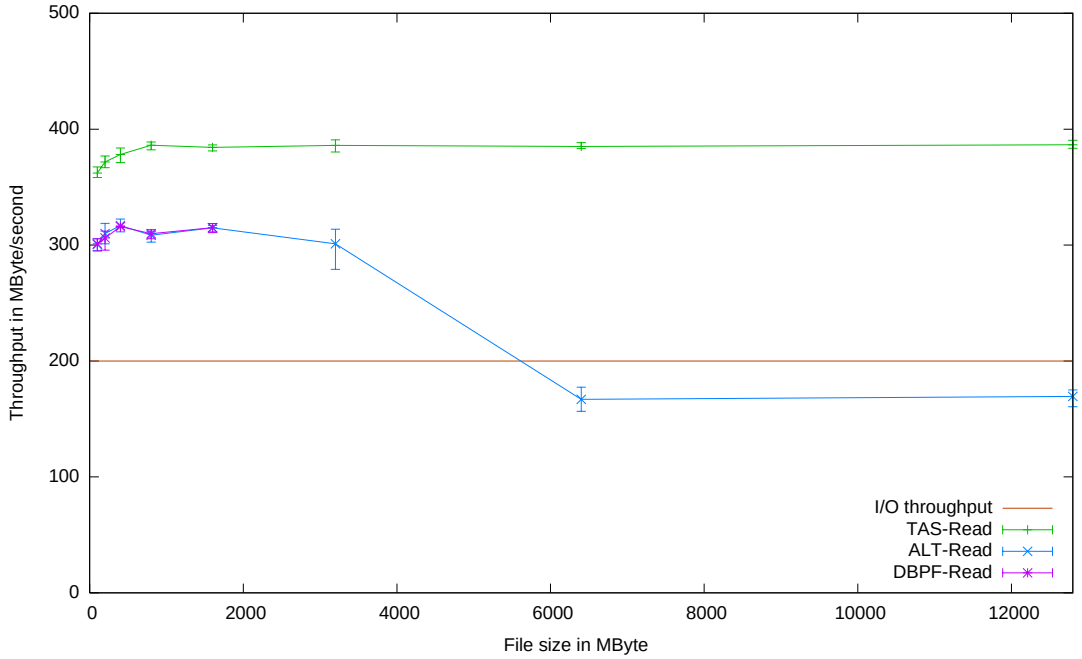

Figure 7.45: Read throughput for 5 clients accessing a large file with a varying size

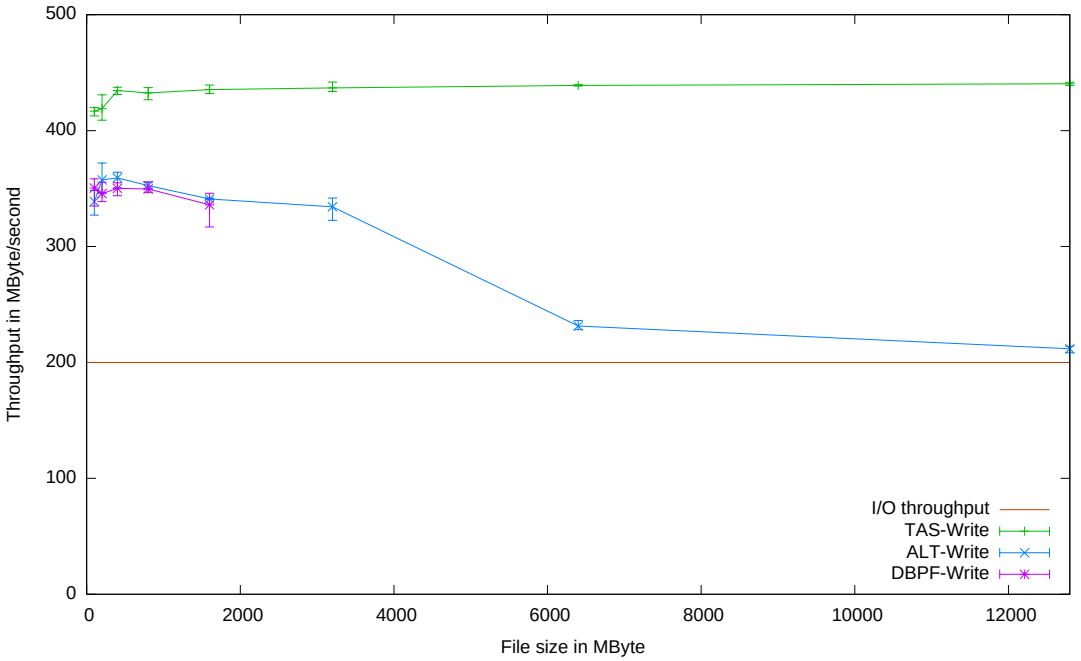

Figure 7.46: Write throughput for 5 clients accessing a large file with a varying size

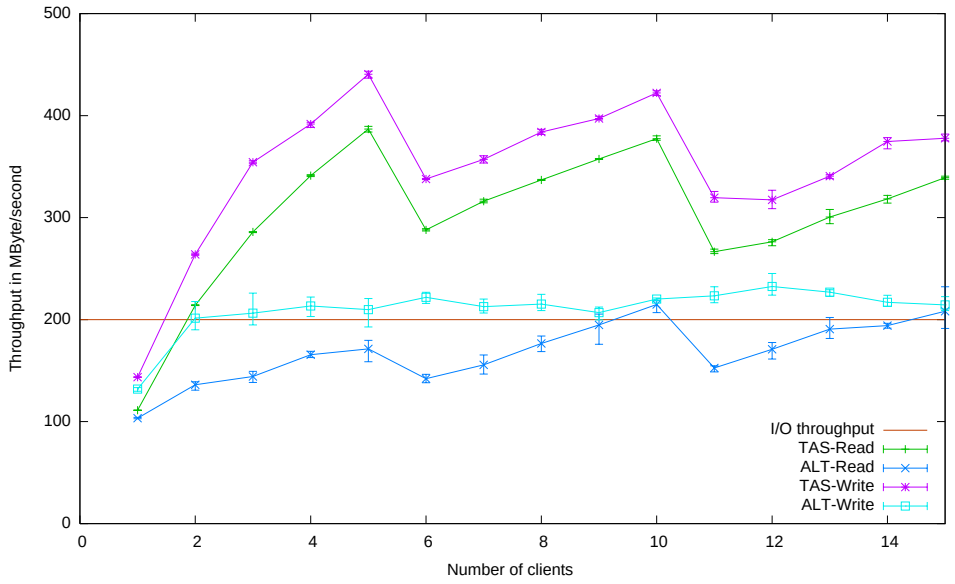

Figure 7.47: Comparison of read and write throughput for a variable client number accessing a large file with a total size of 12800 MByte

Summary

The following results of this section are important:

- The available aggregated throughput for the I/O subsystem is fully exploited by PVFS2.
- Caching mechanisms improve overall throughput. However, the performance of the I/O subsystem limits the throughput for files larger than the memory.
- Read performance is below write performance due to the additional access time. This result is the same as for small contiguous I/O requests.
- In some cases the simple I/O mechanisms of ALT outperforms DBPF.
- If only one client is used, this client benefits from locality of the PVFS2 server.

7.3 Metadata Operations

The utilities `mpi-io-md-more` and `pvfs2-bench` are used to measure the performance for create operations as representative for meta operations. For the system interface the performance for delete operations is measured as well. In a single run a variable number of operations between 200 and 51200 is almost equally distributed between the clients. Each diagram shows the aggregated number of operations which can be done per second.

To give the reader a impression of the size of the persistent representation the size of the ALT databases is determined for 51200 files. In this case the size of the handle database is about 15.5 MByte and the size of the keyval database about 17 MByte. Thus, in average a file system object including the required internal data structures for Berkeley DB needs about 665 Bytes.
The testset for metadata requests is structured as following:

- **Results for one metadata server and one data server**
  The system interface creation and deletion rate is measured for one client with `pvfs2-bench`. In the next step the performance of the MPI interface is compared with the performance of the system interface.
  Then, these results are compared with the estimated upper bounds from chapter 4.

- **Comparison of the performance for one and five data server**
  This section evaluates the results of the configuration with one and five data servers. It turned out that the diagrams look similar for these configurations. Therefore, the diagrams for the configuration with one metadata server and five data servers are omitted in this paper. However, some results are presented later, when the results of one and five metadata servers are analyzed.

- **Results for five metadata servers and five data server**
  For this configuration the same tests are run as for the configuration with one metadata server and one data server.

- **Comparison of the performance for one and five metadata servers**
  In this section the results for the two configurations with one metadata and five metadata servers are compared.

- **Large scale metadata requests**
  The Chiba cluster is used to verify the results of the other configurations for a variable number of clients and servers between 1 and 35.

7.3.1 One metadata and one data server

At first, the results of the system interface are presented and then results achieved with MPI. The throughput of DBPF is measured for syncing and non-syncing mode, and on the underlying filesystems ext3 and tmpfs. The results for DBPF using the in-memory file system tmpfs are expected to be an upper bound for DBPF's results.

System interface

With the help of `pvfs2-bench` the throughput for file creation and deletion is measured. Note that deletion needs exactly four network messages, which is similar to the file creation. Thus, the estimated

upper bound for the deletion rate, which is based on the network latency, has the same value of 480 operations/sec.

Observations:

- TAS forms a upper bound for the other implementations as expected (see figures 7.48 and 7.49).
- With an increase of the subsequent creations ALT and TAS stick to the same throughput whereas DBPF's throughput decreases (see figures 7.48 and 7.49).
- ALT is faster than the non-syncing DBPF on tmpfs (see figures 7.48 and 7.49).
- DBPF on tmpfs is much slower than TAS (see figures 7.48 and 7.49).
- The non-syncing operation mode lifts the performance only a bit (see figures 51-54).
- In creation mode, the non-syncing option improves the throughput slightly but is much worse than DBPFs on tmpfs (see figure 7.48). This is an indicator that I/O operations are necessary for the creation. However, the non-syncing option lifts the file deletion to the level of DBPF-tmpfs, which means in this case there are no hidden I/O operations (see figure 7.49).

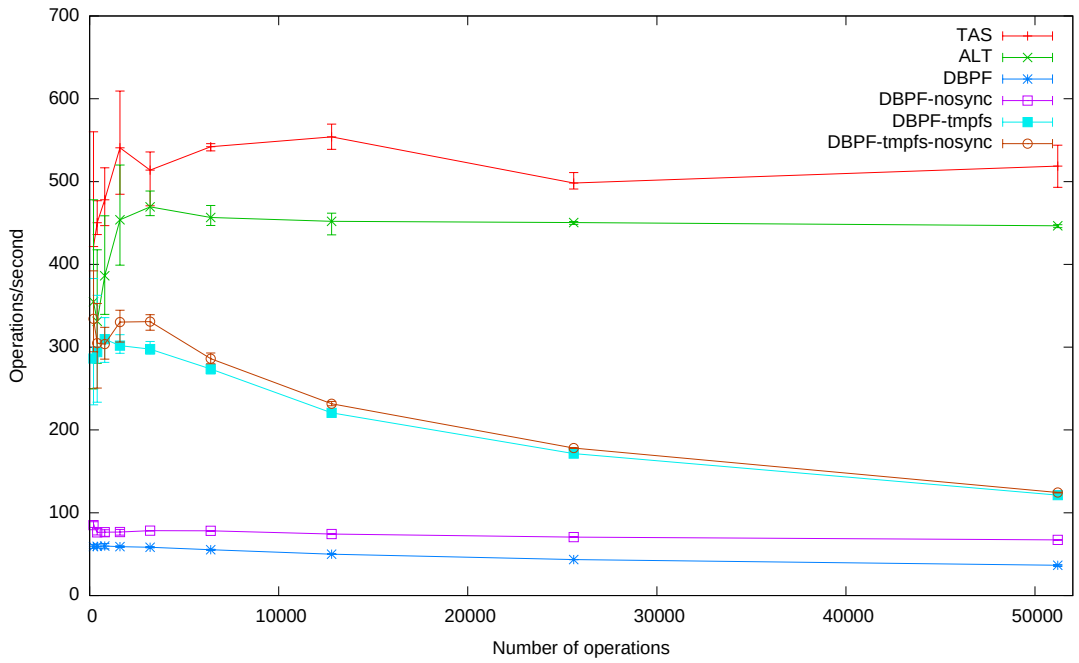

Figure 7.48: System interface creation rate for 1 client creating a different total number of files

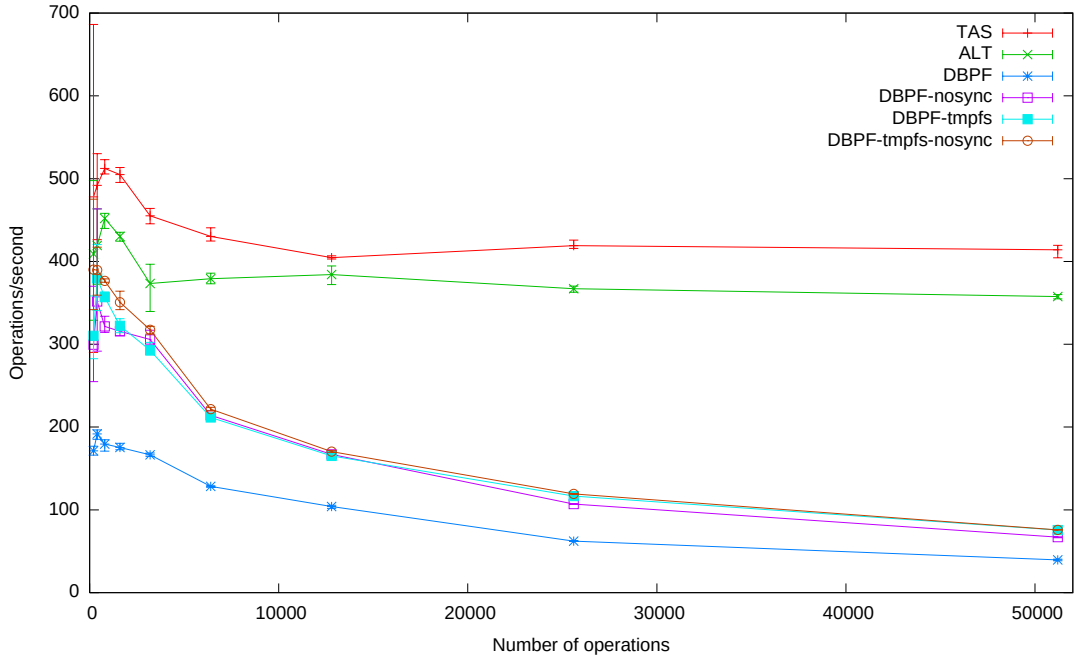

Figure 7.49: System interface deletion rate for 1 client creating a different total number of files

MPI-I/O interface

For the MPI-I/O interface the performance of 1 and 5 clients, using a different total number of create operations, is measured. In addition, the creation rate for a variable client number between 1 and 25 is determined for a total of 6400 and 51200 files. Note that for this configuration clients are distributed over 7 machines. No client is placed on the machine hosting the PVFS2 server. The results of the MPI interface and the system interface are compared for 1 client to point out the overhead of the additional MPI layer. Therefore, data is extracted from the other figures.

Observations:

- Results for 1 client are similar to the system interface results (see figure 52).However, it is noticeable for TAS and DBPF on tmpfs that performance is lost due to request processing by the client's additional layer. DBPF's throughput using the system interface is indistinguishable from the results of the MPI interface.

- TAS overall performance increases proportionally up to 10 clients (see figures 53 and 54). Then the server's CPU is busy processing the requests. ALT aggregated creation rate increases linearly up to 5 clients. However, the throughput never decreases for additional clients.

- DBPF-tmpfs throughput increases with additional clients non-linearly and is outperformed by ALT (see figures 53 and 54).

- Once a resource limit is hit, the overall performance sticks on that level and does not decrease (see figures 53 and 54).

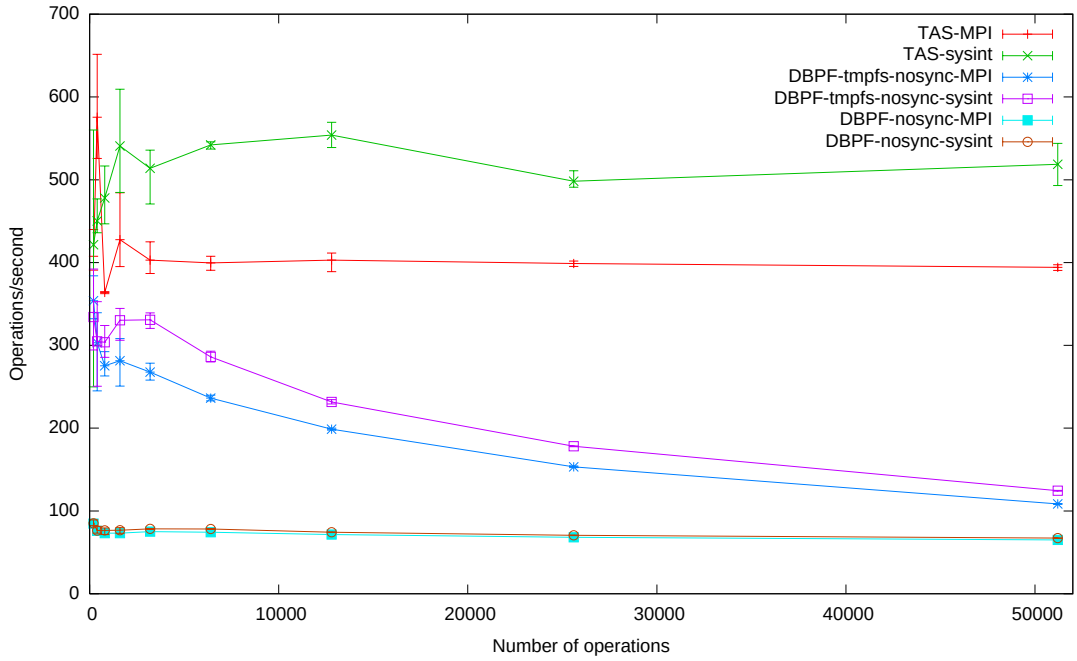

Figure 7.50: Comparison of the MPI and system interface creation throughput for 1 client creating a different total number of files

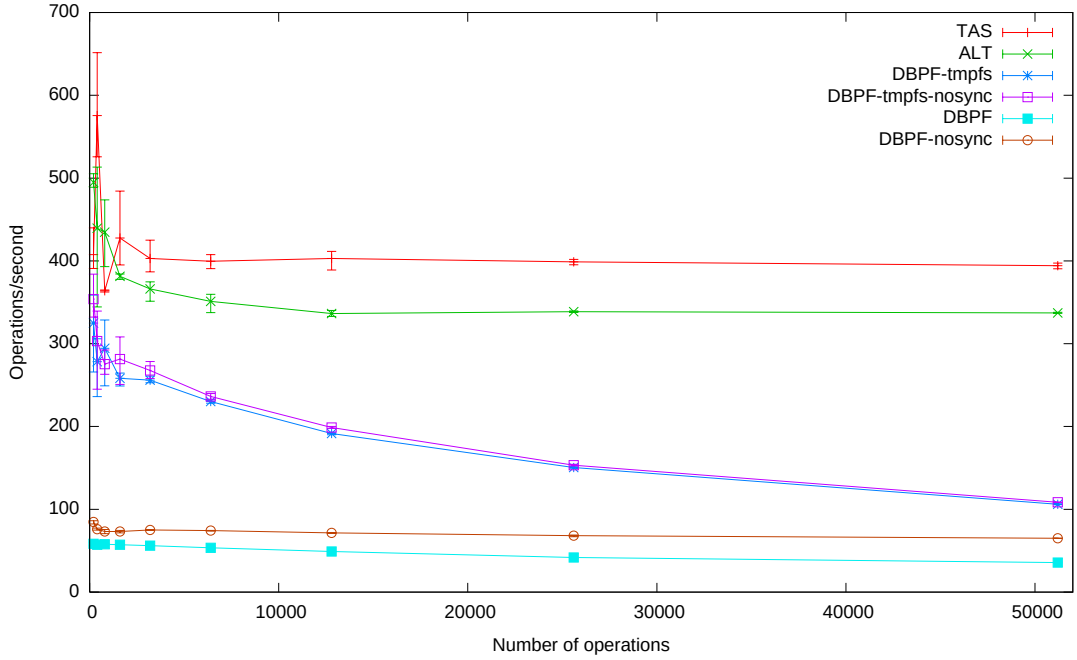

Figure 7.51: Creation rate for 1 client creating a different total number of files

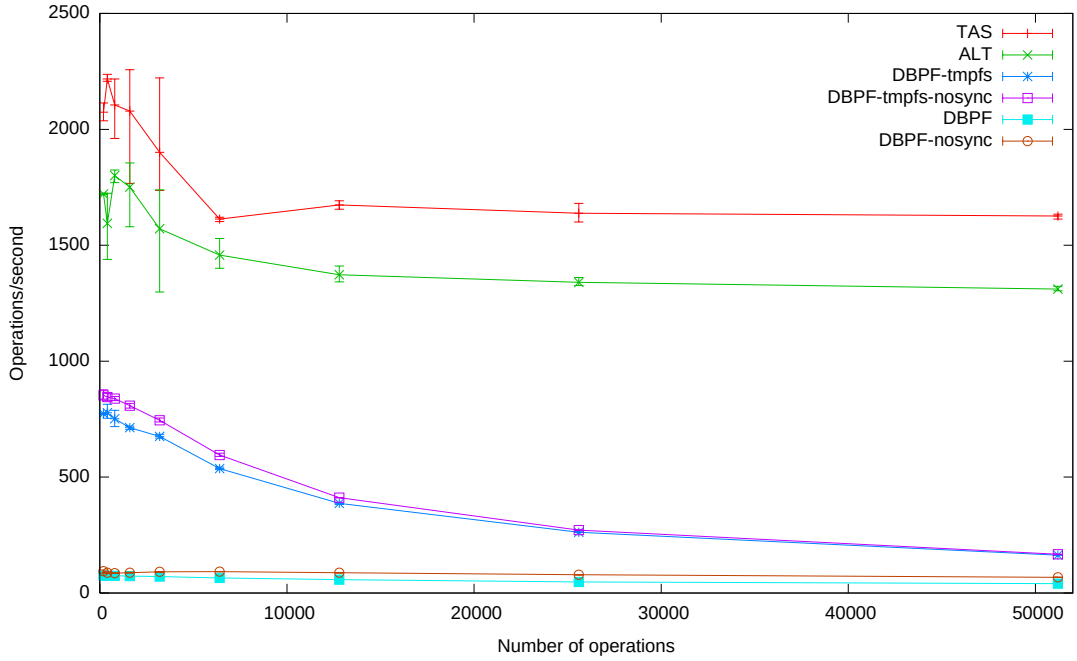

Figure 7.52: Creation rate for 5 clients creating a different total number of files

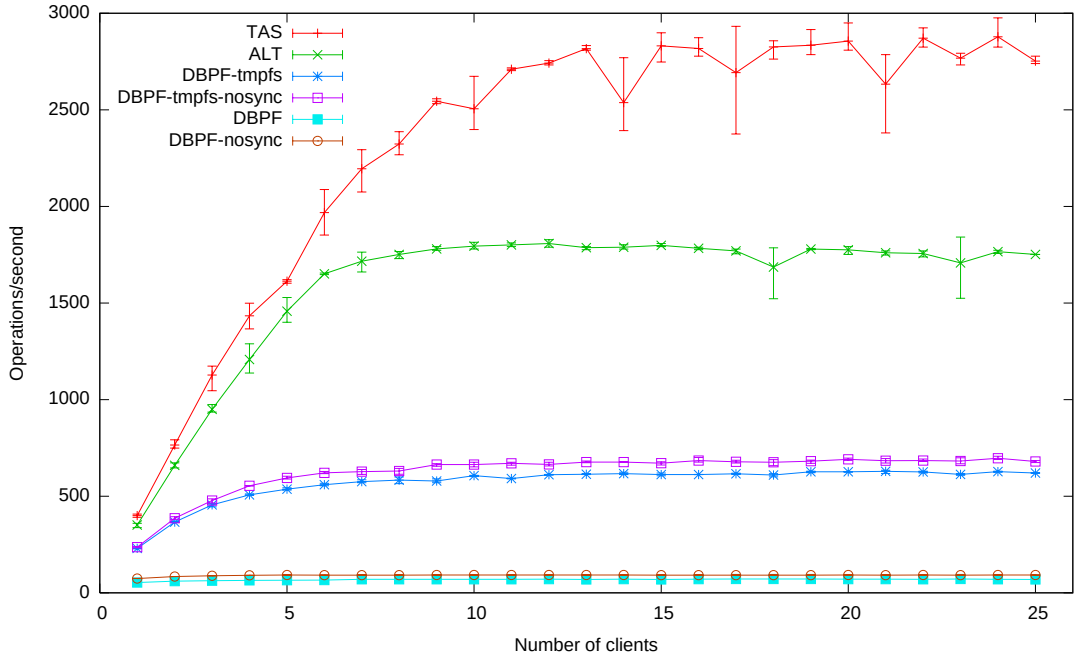

Figure 7.53: Creation rate for a variable client number creating a total number of 6400 files

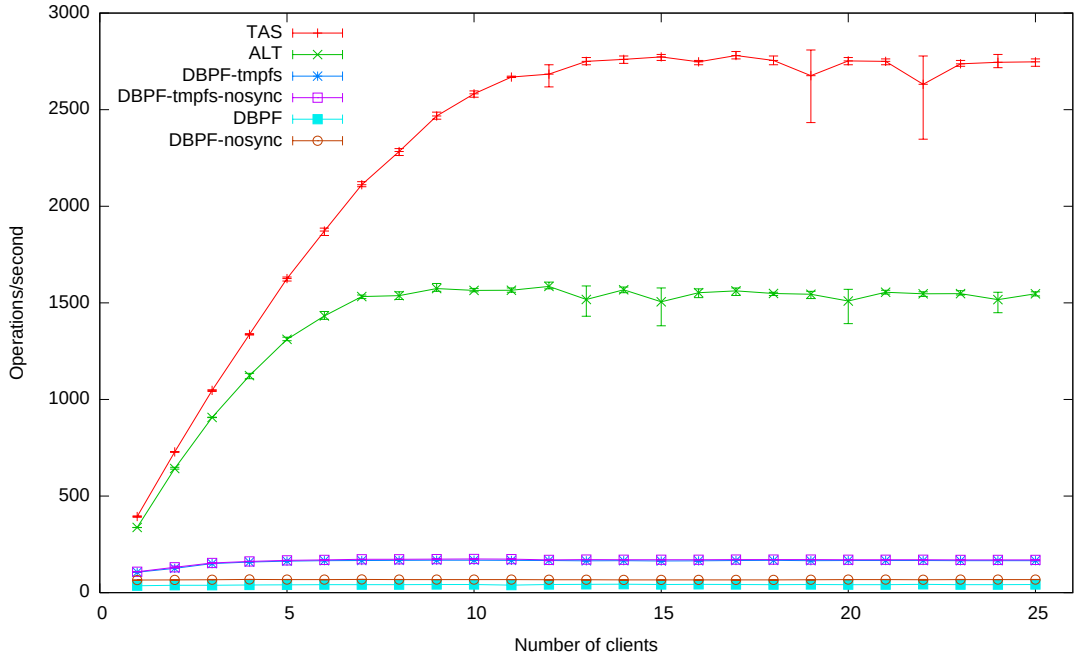

Figure 7.54: Creation rate for a variable client number creating a total number of 51200 files

Comparison of the measured performance and the estimated upper bounds

The data for the diagrams is extracted from earlier diagrams and is not measured again. Figure 7.55 shows that the network latency bound derived from netperf's request-response test is exceeded by TAS. This shows that netperf's results are too high. However, the diagrams 7.55 and 7.56 show that ALT and TAS performance are very close to each other. They must be close to the estimated bound provided by netperf. Thus, there is only little overhead of the intermediate layers.
The throughput of DBPF is close to the optimistic I/O bound, under the assumption that each create operation initiates 10 I/O operations on the server (see figures 7.55 and 7.56).

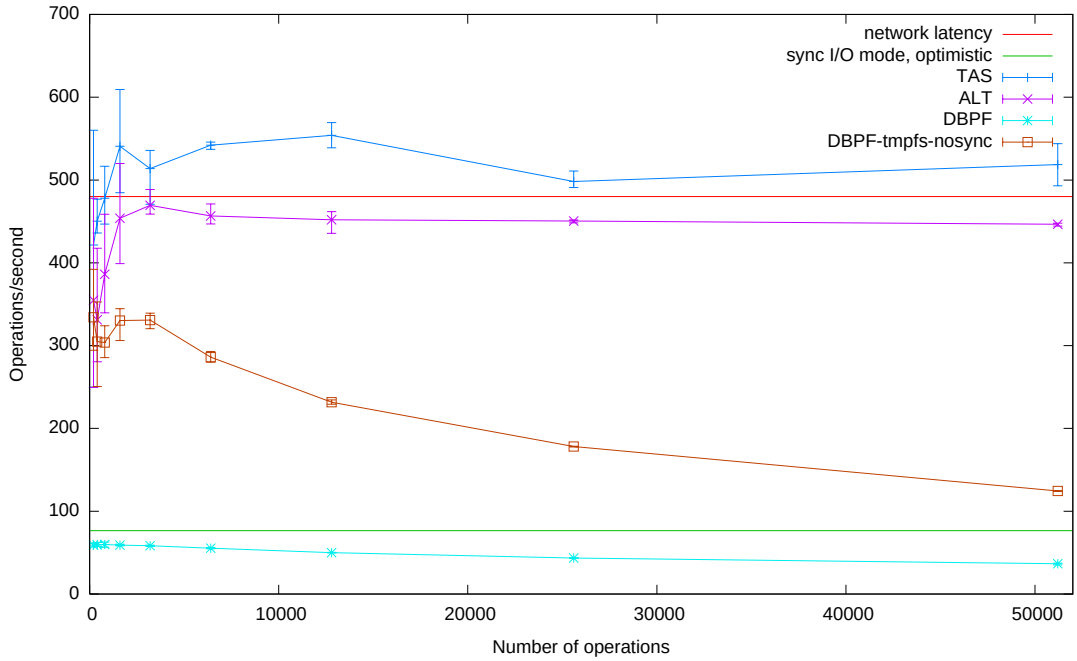

Figure 7.55: Comparison of the estimated upper bounds and the system interface creation rate for 1 client creating a different total number of files

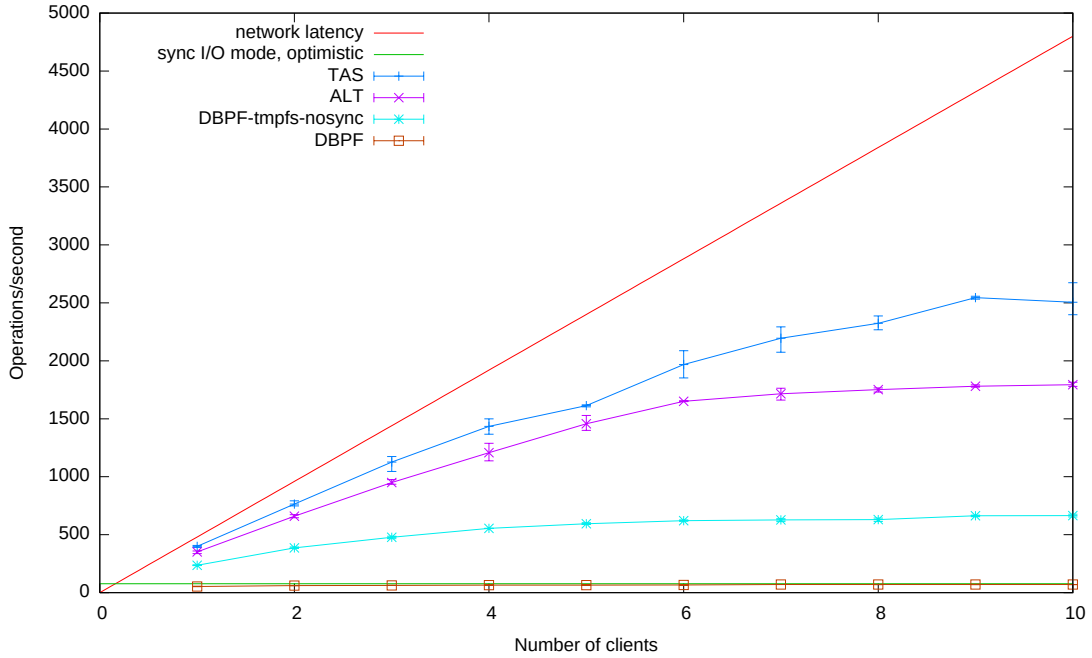

Figure 7.56: Comparison of the estimated upper bounds and the MPI interface creation rate for a variable client number creating a total number of 6400 files

7.3.2 Comparison of the performance for one and five data server

Remember, the clients and servers are now hosted on the same machines. The throughput is below but the diagrams look similar to them for one data server. As a consequence, only the results of one and five data servers are compared in this section. In the next section additional comparisons are made with five metadata servers.

The first client is started on the same machine as the metadata server. However, the locality of the metadata server improves the throughput only a bit (see figure 57). The datafiles are created on every server, thus there is still network communication necessary.
TAS and ALT reveal that the overall performance drops, due to the additional network communication. This cannot be measured for DBPF on tmpfs for the creation of 51200 files.

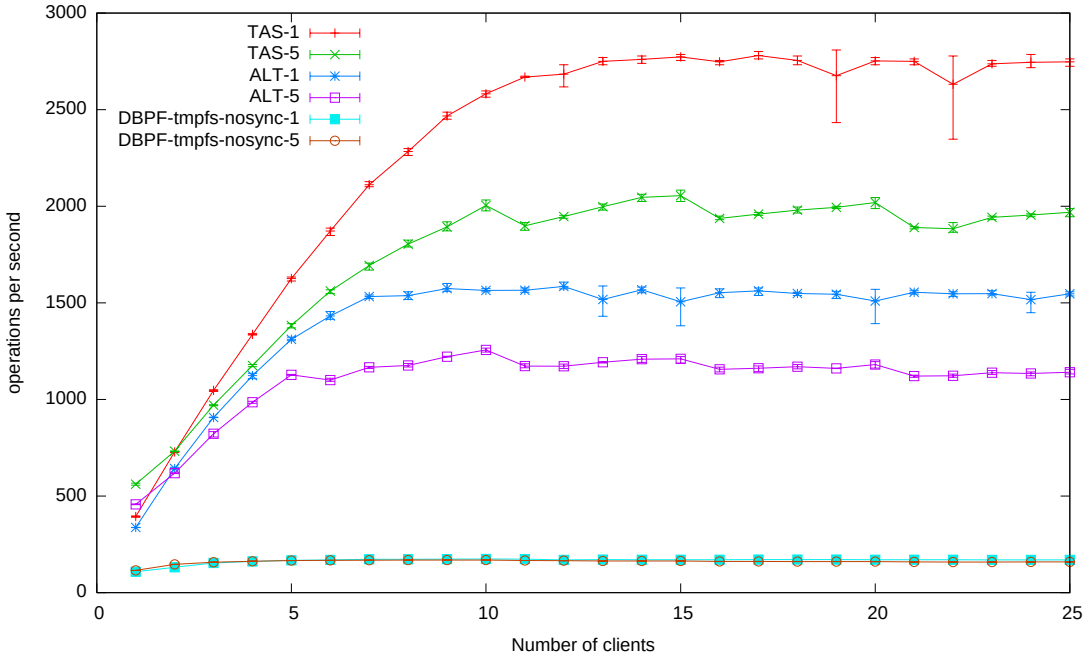

Figure 7.57: Comparison of the creation rate for the configuration with 1 and 5 data servers using a variable client number creating a total number of 51200 files

7.3.3 Five metadata servers and five data servers

Most observions are equal to them for one metadata server and one data server with on exception, the rate of DBPF-nosync is now twice the rate of DBPF. A comparison between the two configurations using 5 data servers is made in the next section. Therefore, the measured data of this configuration is used.

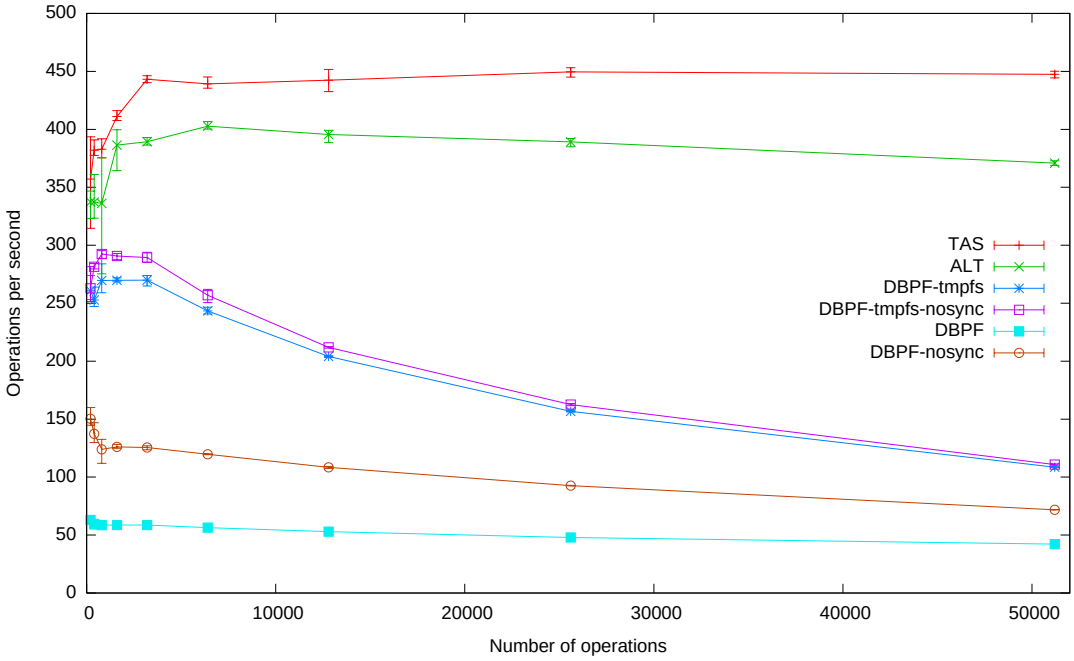

Figure 7.58: Creation rate for 1 client creating a different total number of files

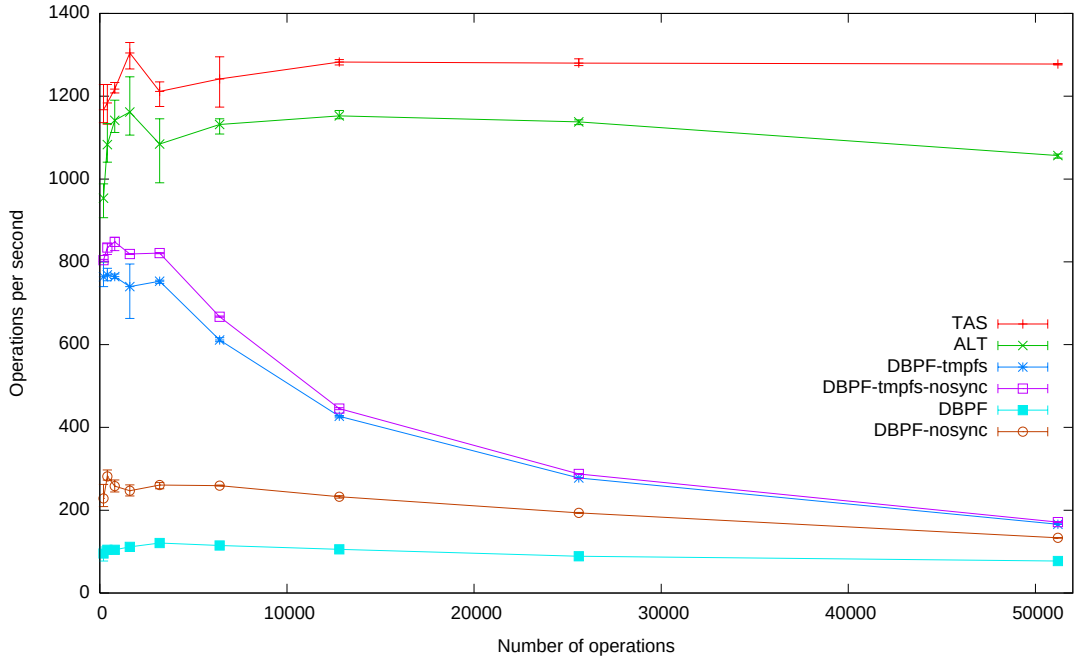

Figure 7.59: Creation rate for 5 clients creating a different total number of files

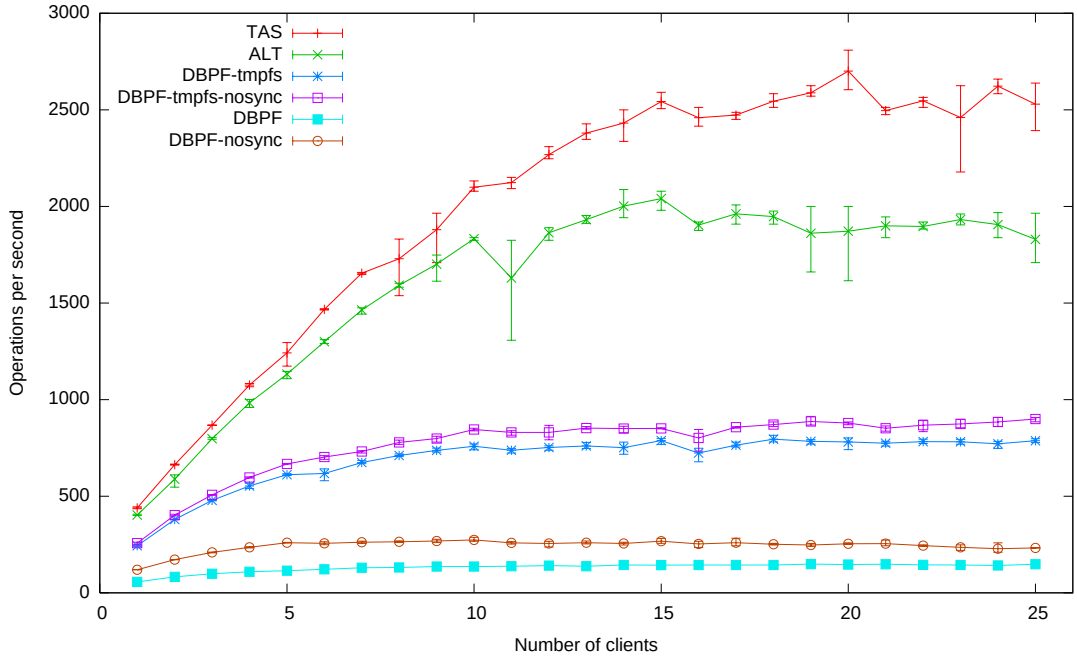

Figure 7.60: Creation rate for a variable client number creating a total number of 6400 files

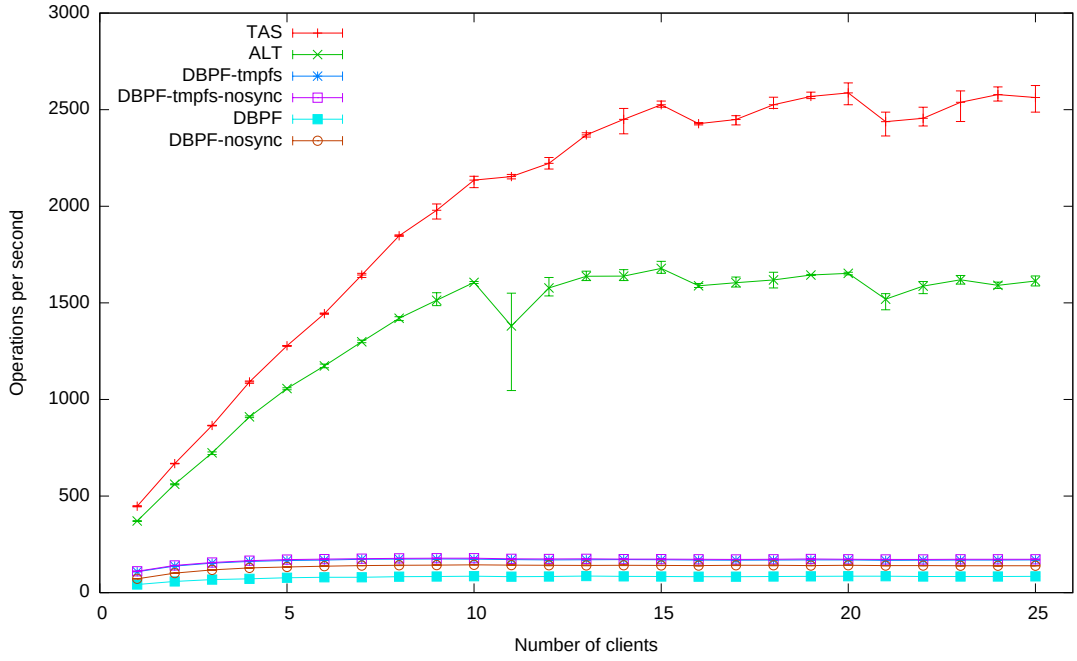

Figure 7.61: Creation rate for a variable client number creating a total number of 51200 files

7.3.4 Comparison of the performance for one and five metadata servers

The following MPI-I/O tests of the two configurations with one and five metadata servers are evaluated in this subsection. The creation rate is measured for 1 and 5 clients using a different total number of create operations and for a total of 51200 files using a variable client number between 1 and 25.
The data for the five metadata servers is extracted from the diagrams of the previous subsection where these were already shown.

The performance of the DBPF variants is much lower than the performance of the new modules. Therefore, the benchmark results of the DBPF variants are displayed in dedicated diagrams.

Observations:

- Less throughput for in-memory methods TAS and DBPF-tmpfs for one client and five metadata servers (see figures 7.62 and 7.63). The same performance reduction is observed for ALT.
- For one client DBPF benefits only a bit from the additional hardisks (see figure 7.63). This is due to the required synchronizing with the disk, which is done on all machines in the same way. Remember, the requests are processed serially by a client.
- The creation rate improves for multiple metadata servers using the configuration with a deactivated syncing mode (DBPF-nosync) (see figure 7.63). However, the performance of the two different configurations converge for a higher number of create operation.
- TAS with one metadata server is faster until the CPU is busy processing the requests (see figure 7.66). Multiple servers then benefit from the sharing of the requests.
- For one metadata server the performance of ALT increases linear up to 5 clients, whereas it increases linear for five metadata servers up to 10 clients (see figure 7.66).

- Performance of DBPF on tmpfs does not improve for 5 metadata servers (see figure 7.67).
- Throughput for DBPF doubles for five metadata servers, when accessed by multiple clients (see figure 7.67).

For multiple metadata servers performance is less than for a single data server because in average only every 5th request is processed locally for five data servers. Whereas for one data server and one client all requests omit network communication, except the creation of the datafiles which require communication.

Servers and clients share the same machines for this configuration. Therefore, the two CPUs are shared between multiple processes, i.e. for 5 clients each server hosts an additional client, for 10 clients two clients and so on. The clients and servers compete for the CPU, thus the overall performance is reduced.

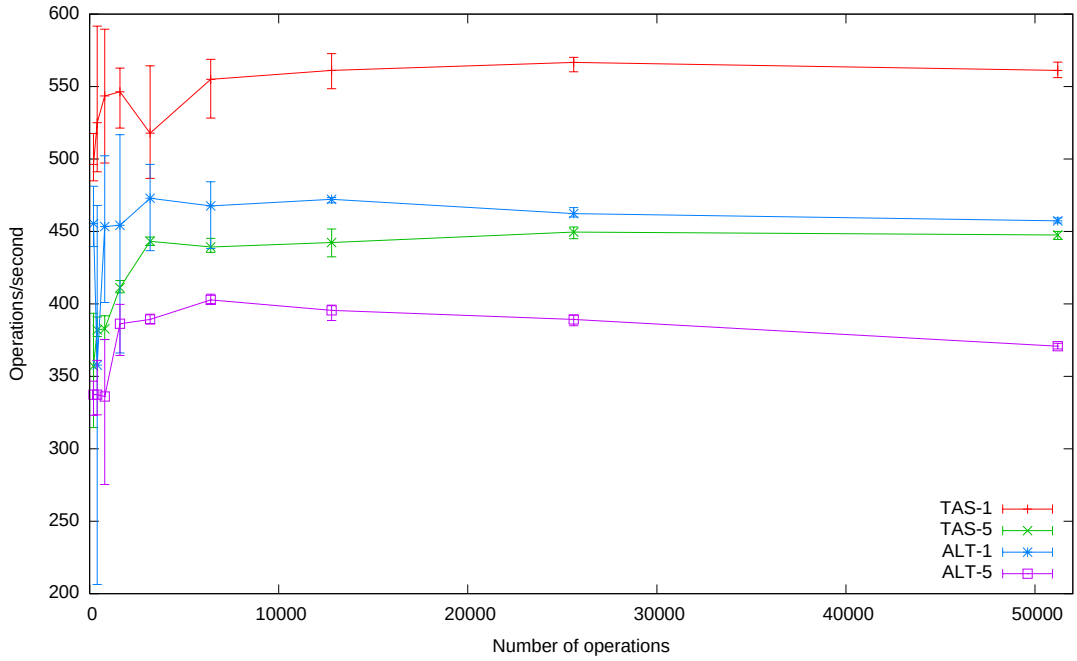

Figure 7.62: Comparison of the creation rate for the new modules using the configuration with 1 and 5 metadata servers accessed by 1 client creating a different total number of files

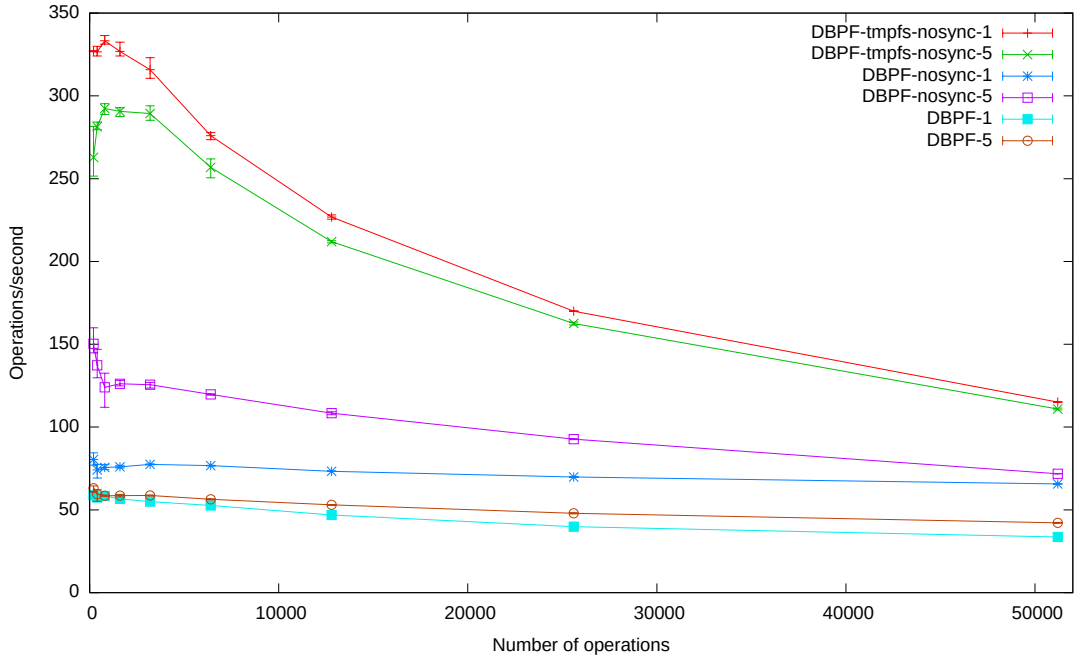

Figure 7.63: Comparison of the creation for the DBPF module using the configuration with 1 and 5 metadata servers accessed by 1 client creating a different total number of files

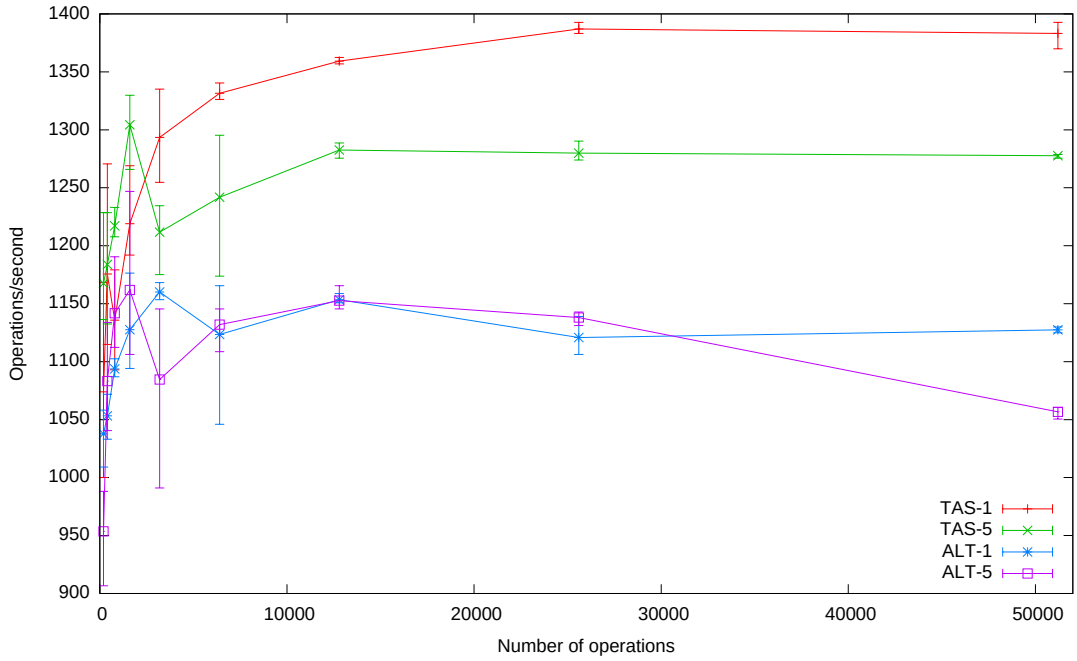

Figure 7.64: Comparison of the creation rate for the new modules using the configuration with 1 and 5 metadata servers accessed by 5 clients creating a different total number of files

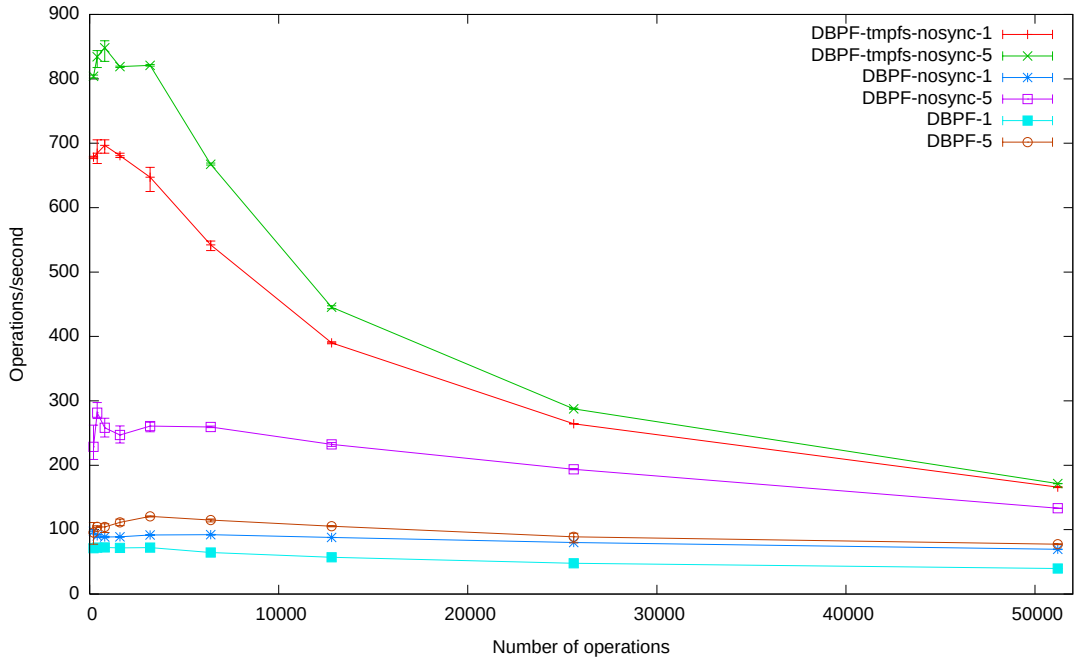

Figure 7.65: Comparison of the creation rate for the DBPF module using the configuration with 1 and 5 metadata servers accessed by 5 clients creating a different total number of files

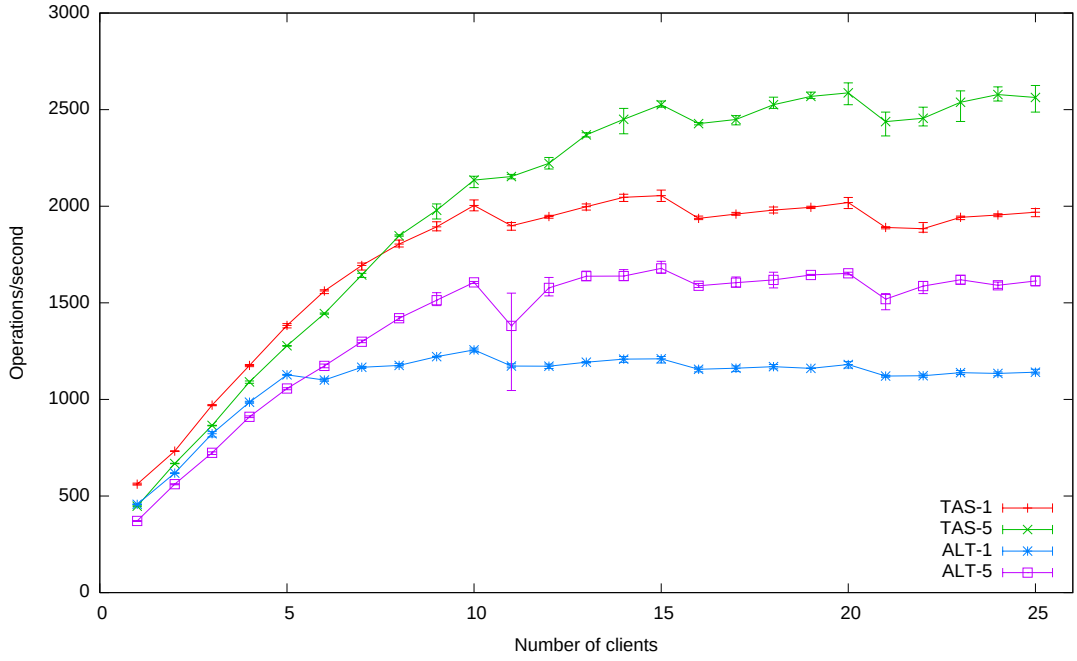

Figure 7.66: Comparison of the creation rate for the new modules using the configuration with 1 and 5 metadata servers accessed by a variable client number creating a total number of 51200 files

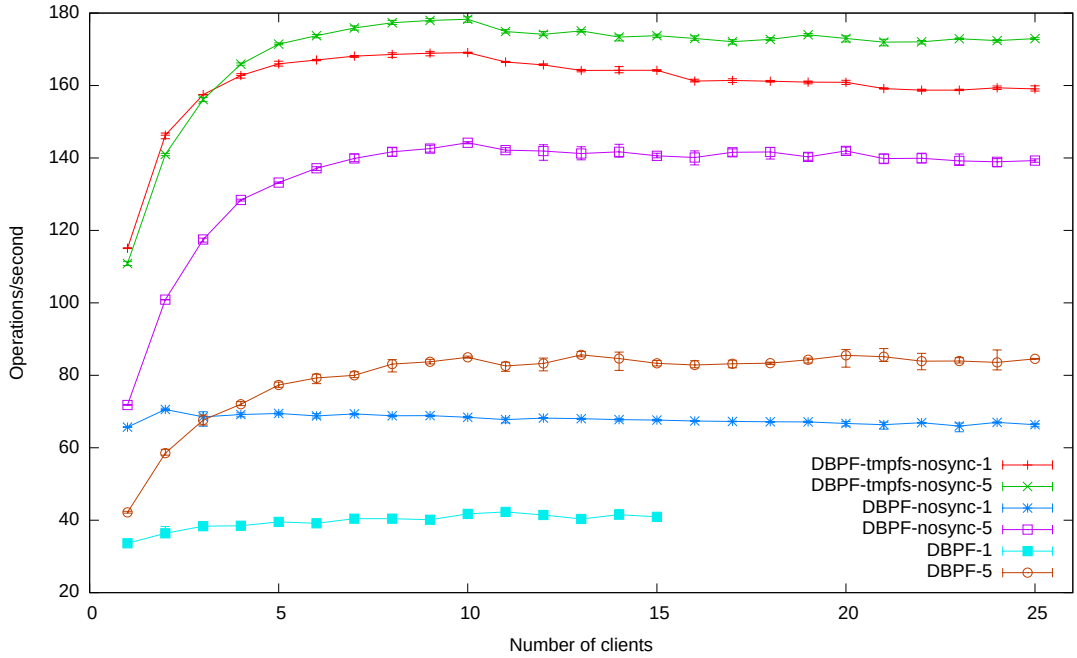

Figure 7.67: Comparison of the creation rate for the DBPF module using the configuration with 1 and 5 metadata servers accessed by a variable client number creating a total number of 51200 files

Summary

The following results of the metadata analysis are important:

- There is much room for improvement for DBPF. The creation rate of DBPF is only one tenth of ALT's creation rate.
- With the help of the upper bound provided by TAS we get a impression about the capabilities of the architecture and the performance which the persistency layer could exploit.
- It is vital to cache metadata operations in order to achieve a good performance.
- With an increasing number of operations, the creation rate decreases for the DBPF variants. This observation is made especially for DBPF on tmpfs.
- The non-syncing variant improves the performance only slightly.
- Under most circumstances the aggregated performance of an environment providing multiple metadata servers is lower than for a single metadata server. A small speedup for multiple servers can be observed for a high number of clients. The maximum speedup is 2 which is measured for 5 metadata servers. This is beyond the expected speedup of 5.

7.4 Large Scale Metadata Requests

The Chiba City cluster that we use to verify the large scale metadata throughput is build with older hardware: dual-cpu Pentium III 500 MHz systems with 512 MB of memory. The nodes are interconnected with Myrinet and a 100 MBit Ethernet, which is used as network for the tests.

MPI file creation throughput is measured for 1, 2 and 5 processes running on each client node. However, the results of a different number of processes per node are similar. Therefore, only the diagram for 1 client is presented and a diagram showing the results for 1 and 5 processes per node. The client and server nodes are disjunct for the tests. Client and servers get the same node count. Each client node creates exactly 1000 files which are shared between the processes per node.
ALT is not at the newest version and uses the default Berkeley DB cache size of 256 KByte for the keyval and the dataspace database. The new version uses a database environment for both databases with 1 MByte cache. This boosts overall performance.

The diagrams 7.68 and 7.69 point out that starting with 5 clients the creation rate is nearly constant. This means PVFS2's architecture scales well up to 35 servers. I got some single data points for 50 servers as well, which stick to the same throughput. However, it is not enough data to present the results in a nice diagram.

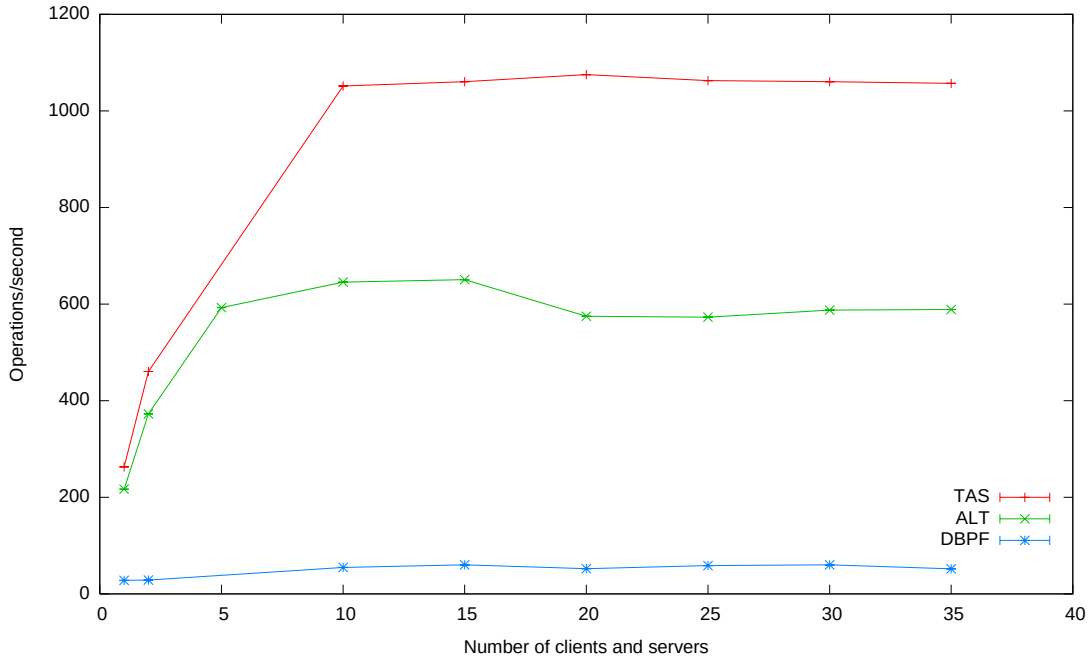

Figure 7.68: Creation rate for a variable number of clients and servers. Each client-node starts 1 process creating 1000 files

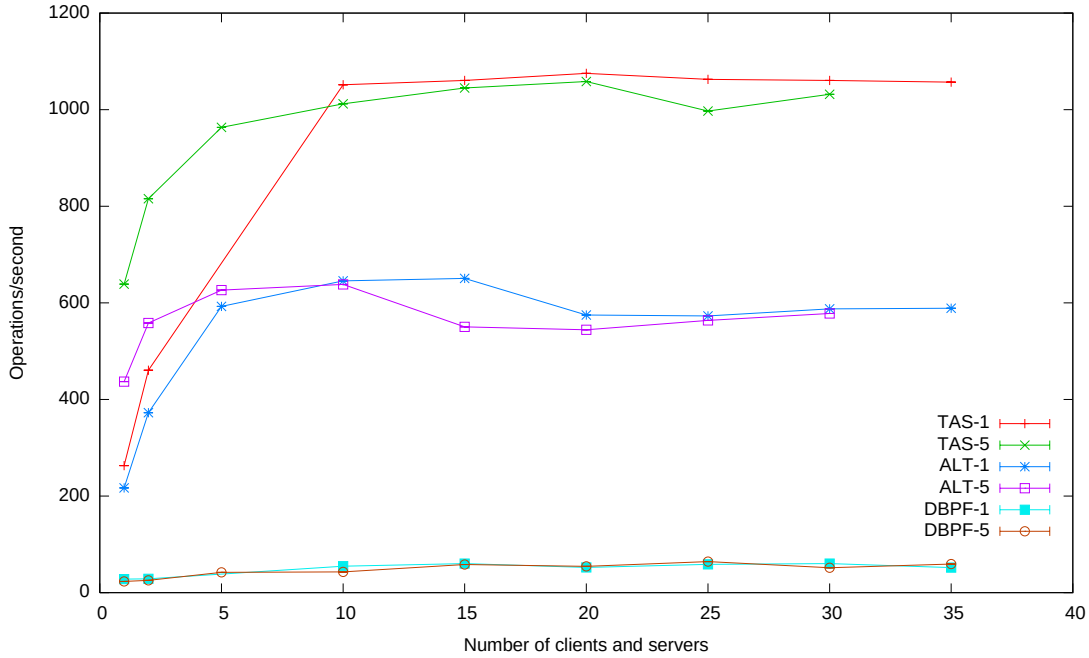

Figure 7.69: Comparison of the creation rate for 1 and 5 processes per client-node. A variable number of clients and servers gets started. Each client-node creates a total of 1000 files

Summary

The following observations are made for a larger number of clients:

- The aggregated creation rate is nearly constant for a different number of clients and servers. This means the architecture of PVFS2 scales well for metadata operations up to 50 clients.
- DBPF is outperformed by ALT and TAS. The results are comparable to the results of newer hardware.

7.5 Suggestions for a New Trove Implementation

In my opinion the following strategies look promising for a Trove module. Most strategies result directly from the observations of the benchmarks.

Caching Caching is vital to lower the impact of the slow persistent storage. Consequently an efficient caching is necessary.

- Berkeley DB caching mechanisms can be used to keep the code clear.
- Introduction of a file system configuration option for the cache size that users can set the DB's cache size to an appropriate value. It is advised to use as much cache as possible.
- Reduce the memory usage of the objects:
  Store only necessary attributes. Not all objects need the whole range of attributes, e.g. the datafile needs no timestamps.
  Maybe it would be worthwhile to have a dataspace and keyval database per collection. Then,

the collection id has not to be part of the object's persistent representation. However, Berkeley DB does not store the whole key for each entity, thus the modifications might be useless.

- Store a datafile's bytestream size in the datafile's attributes to avoid the lookup with `fstat`. It is expected that the file's attributes are in the cache during the I/O because the server fetches them for the request, thus no I/O operation would be necessary to lookup the size.

- Berkeley DB's non-syncing writes should be used. A scratch file system is useful for some users. In this case the metadata throughput should not suffer. To guarantee the databases consistency, the user may select a transaction mode for single operations. The file system check utility `pvfs2-fsck` fixes inaccessible objects and cleans the file system.

Threads All blocking I/O operations should be enqueued and handled by different threads. There should be threads responsible for:

- Metadata operations

- I/O operations

- BMI callback function

DBPF already uses different threads. A separate handling of metadata processing and BMI callback might improve the overall throughput. It is important to keep all limited resources busy to achieve the best performance. A parallel processing of I/O and network access keeps the I/O subsystem and the network busy. The benefit of such a strategy can be seen for large I/O and multiple servers in the evaluation. I have seen this for small accesses and a large amount of clients, as well. However, this results were not published in this paper. In order to achieve the best overall performance, it might be necessary to choose the strategy depending on the server's load.

Handle management Never free handles for a reuse explicitly. Instead all servers memorize only one free handle range for each collection, which gets initialized with the servers responsible handle range. When a server's available handles are nearly exhausted, a thread can be started, which tries to reclaim a set of contiguous unused handle numbers. If the server detects that the present range is empty, it switches the present range with the threads' determined range. So the handle reuse time is controlled by the time difference between the thread determining a range for reuse and the time the handles are actually exhausted, thus is implicit.

The calculation of the time difference depends on the server's capabilities and the timeout. Assume the following facts: The server can create file system objects at a rate of 2000 objects per second. It takes 5 minutes to iterate over the present objects. Handle recycle timeout should be set to 5 minutes.
Then it takes 600 seconds to find the largest handle range available. During this time about one million of files can be created. A handle number is a 64 Bit unsigned integer, thus it is seldom that the range gets exhausted. Also, it is possible to start the reclaim thread early to guarantee enough available handles for creation.
If every second handle is free, this realization is problematic because all ranges have the size of only one handle. To prevent this uncommon situation an additional memory region can hold the required amount of free handles. However, this is really unrealistic because there is not enough storage space to hold all the objects. It is also unrealistic that the maximum possible rate is used for file creation. However, this shows that such an approach is applicable. Another simple strategy is to reclaim less handles and to delay the creation of new objects if there are no handles left.

8 Summary

This section summarizes the project and its results.

We have the idea to create a upper bound for the performance of the different operations. Therefore, the new module TAS is created for the persistency layer, which is a low-level layer of PVFS2. This module has a well known complexity for the performance relevant operations and is independent of the I/O subsystem. Some simple considerations allow us to make a close estimate of the expected performance of the operations.

Three test cases are evaluated: contiguous I/O requests using small block sizes per request, large contiguous I/O requests and file creation as representative for metadata operations. There is no common benchmark which is suitable for such an analysis of the parallel file system. It is necessary to have reference results in order to estimate the use of an optimization. For this project some scripts were developed which started the benchmark programs with different parameters in batch mode.

A comparison of the estimated upper bounds and the results provided by TAS shows bottlenecks of the architecture i.e. the cut for a block size of 128 KByte and the acache. The author claims that the flow-protocol is the reason for this. However, the achieved performance is very close to the upper bounds for most test cases. Thus, the PVFS2 architecture exploits the available performance.

The TAS module acts as a reference for throughput achievable from the other Trove modules. With the help of such an upper bound the real costs of a module's I/O strategy are pointed out. As an example this shows that the metadata is handled suboptimally by DBPF. There is much room to improve the metadata handling. It can be seen that I/O operations reduce the metadata processing significantly. The Trove module processing of the metadata should be optimized to avoid as many I/O operations as possible.

An alternative module for Trove is designed for evaluating different strategies. Interestingly, results show that a straightforward I/O handling of data is not worse than a more complex I/O handling. In several aspects a detailed analysis is necessary to optimize the performance. It is important to do performance analysis. Otherwise, complex mechanisms are developed that are not faster than the simple strategies. However, such mechanisms complicate the source code, increase the effort for maintenance and make it hard to introduce new algorithms.

9 Future Works

The performance analysis of the Trove modules showed that different mechanisms may lead to a higher overall throughput and revealed some bottlenecks. However, the work is not done. There are still some unresolved issues with the modifications made. Also, much further evaluation is necessary, in order to eventually come to an optimal implementation.

The following list provides ideas on major steps that could be made:

- Analysis of other critical operations like lookup of a file system object.
- Experimenting with different Trove strategies: e.g. figure out the impact of multiple threads processing the callback function and a different handle management.
- Benchmarking of more systems and configurations. It is necessary to measure performance of multiple servers which are disjunct to the clients.
- The performance of ALT should be measured for a very high number of files. To figure out the impact of the cache size, it is necessary to access the metadata later. Therefore, a benchmark is required. For further analysis a benchmark looking up existing and non-existing file system objects, randomly or subsequent, might be useful.
- Evaluate a change to the interface for keyval iteration which is suited for ALT.
- Setup of a test environment to compare performance of different code revisions in order to reveal unfavorable modifications.
- Adaption of the paper's insights for a new module or adaption of DBPF. This should be done when it is figured out which strategies optimize the performance.

The following are minor activities:

- Complete to divide Trove into storage method and collection method. Create a better mechanism for the lookup of the responsible collection method.
- TAS and ALT can be modified to support all Trove functions i.e. extended attributes.
- Investigate the client's I/O performance cut for read operation with 128 KByte. The author claims that the flow protocol is involved in this issue. This is not so urgent because a server is accessed by multiple clients normally.
- Improvement of the client side caches: the ncache does not store path efficiently and compares the objects in a slow linear fashion.

10 Appendix

10.1 Comparisons of ALT

Multiple databases for key/value pairs In the creation benchmark the performance of multiple databases for the key/value pairs achieve worse results than the single database. The two common keys which specify the distribution and datafile handles for a PVFS2 file are placed in separate databases.

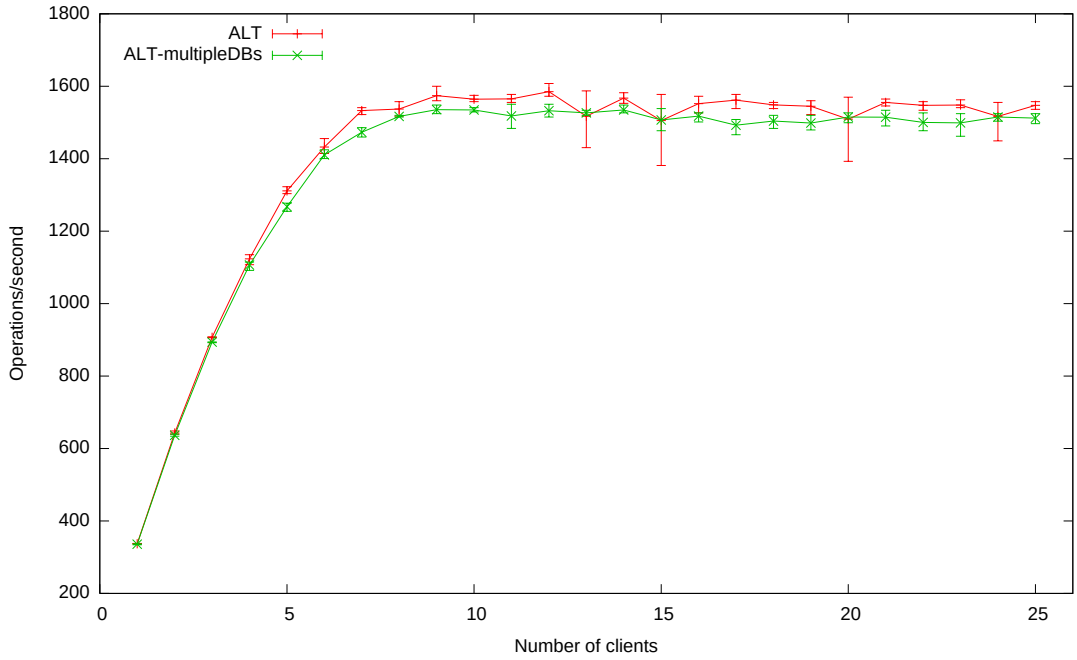

Figure 10.1: Comparison of ALT with one and three keyval databases for the configuration with 1 metadata and 1 data server. Creation rate for a variable client number creating a total number of 51200 files

Different cache sizes of the databases The performance is comparable for file creation. In this case metadata of an object is only needed during the creation. Thus, there is little impact due to different cache sizes. The cache size is important if the benchmark operates on older files later. In addition, it is likely that the kernel buffers the whole databases in memory because the databases have a total size of about 30 MByte.

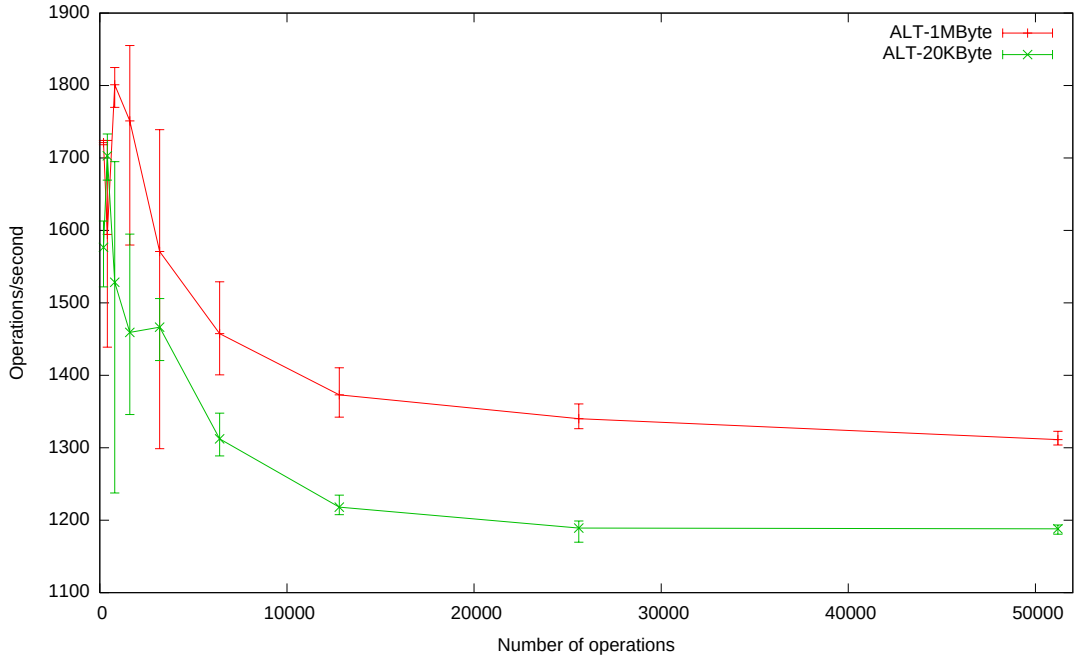

Figure 10.2: Comparison of ALT with different cache sizes for the configuration with 1 metadata and 1 data server. Creation rate for 5 clients creating a different total number of files

10.2 Difficulties

During the work, the following time consuming problems arose:
Some bugs in PVFS2 were revealed while some problems still remain. Especially, the acache expensive cleanup of outdated entries needed a lot of time for investigation.
For large contiguous I/O operations the DBPF implementation hung-up on the testbed. This happened randomly and looked like a timing problem. Several tries to reproduce the problem on Chiba and my local system were in vain.

The infrastructure of the clusters wasted a lot of time: First an initial setup with PVFS2 and MPI has to be compiled.
Benchmarks have to fit to the clusters job scheduling mechanisms. In order to get the tests to run on a cluster, it was necessary to understand the underlying infrastructure and to create an appropriate set of scripts capable to run the tests in batch mode.

On the working groups cluster, sometimes the hard disk's DMA mode was switched off. The reason for this behavior is the kernel's IDE driver. This issue is resolved on newer kernels. However, a kernel update is not so simple on the cluster. A deactivated DMA mode on an arbitrary node has more or less impact on the benchmarks' results, thus is not easy to detect. When I realized that there is a DMA problem, I started to check the DMA modes on every node during the benchmarking and restarted the tests whenever the problem was detected and resolved. To switch the DMA mode back on, it was necessary to reboot the affected machine.
Also, for the disjunct configuration, `rsh` did not work properly sometimes. It happens that a host could not be found. Then, no connection was possible for some time.

10.3 PVFS2 file system configuration files

The configuration files are for the DBPF non-syncing case. For metadata synchronizing the storage hint `TroveSyncMeta` is set to `yes`. The `HandleRecycleTimeoutSecs` for MPI-I/O operations is set to 5000 to avoid additional lookup of the file attributes and name. However, this has little influence on the throughput.

10.3.1 One metadata and one data server

```
<Defaults>
        UnexpectedRequests 50
        LogFile /tmp/pvfs2-kunkel.log
        EventLogging none
        LogStamp usec
        BMIModules bmi_tcp
        FlowModules flowproto_multiqueue
        PerfUpdateInterval 1000
        ServerJobBMITimeoutSecs 30
        ServerJobFlowTimeoutSecs 30
        ClientJobBMITimeoutSecs 300
        ClientJobFlowTimeoutSecs 300
        ClientRetryLimit 5
        ClientRetryDelayMilliSecs 2000
</Defaults>

<Aliases>
        Alias master2 tcp://master2:6555
</Aliases>

<Filesystem>
        Name pvfs2-fs
        ID 48650554
        TroveModule dbpf
        RootHandle 1048576
        <MetaHandleRanges>
                Range master2 4-2147483650
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 2147483651-4294967297
        </DataHandleRanges>
        <StorageHints>
                TroveSyncMeta yes
                TroveSyncData no
                AttrCacheKeywords datafile_handles,metafile_dist
                AttrCacheKeywords dir_ent, symlink_target
                AttrCacheSize 4093
                AttrCacheMaxNumElems 32768
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>

<Filesystem>
        Name pvfs2-tas
        ID 6187900
        TroveModule tas
        RootHandle 10
```

```
        <MetaHandleRanges>
                Range master2 4-2147483650
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 2147483651-4294967297
        </DataHandleRanges>
        <StorageHints>
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>

<Filesystem>
        Name pvfs2-alt
        ID 6000000
        TroveModule alt
        RootHandle 10
        <MetaHandleRanges>
                Range master2 4-2147483650
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 2147483651-4294967297
        </DataHandleRanges>
        <StorageHints>
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>
```

10.3.2 One metadata and five data servers

```
<Defaults>
        UnexpectedRequests 50
        LogFile /tmp/pvfs2-kunkel.log
        EventLogging none
        LogStamp usec
        BMIModules bmi_tcp
        FlowModules flowproto_multiqueue
        PerfUpdateInterval 1000
        ServerJobBMITimeoutSecs 30
        ServerJobFlowTimeoutSecs 30
        ClientJobBMITimeoutSecs 300
        ClientJobFlowTimeoutSecs 300
        ClientRetryLimit 5
        ClientRetryDelayMilliSecs 2000
</Defaults>

<Aliases>
        Alias master2 tcp://master2:6555
        Alias node05 tcp://node05:6555
        Alias node06 tcp://node06:6555
        Alias node07 tcp://node07:6555
        Alias node08 tcp://node08:6555
</Aliases>

<Filesystem>
        Name pvfs2-fs
        ID 61878687
        TroveModule dbpf
        RootHandle 1048576
```

```
        <MetaHandleRanges>
                Range master2 4-715827885
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 715827886-1431655767
                Range node05 1431655768-2147483649
                Range node06 2147483650-2863311531
                Range node07 2863311532-3579139413
                Range node08 3579139414-4294967295
        </DataHandleRanges>
        <StorageHints>
                TroveSyncMeta yes
                TroveSyncData no
                AttrCacheKeywords datafile_handles,metafile_dist
                AttrCacheKeywords dir_ent, symlink_target
                AttrCacheSize 4093
                AttrCacheMaxNumElems 32768
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>

<Filesystem>
        Name pvfs2-tas
        ID 6187900
        TroveModule tas
        RootHandle 10
        <MetaHandleRanges>
                Range master2 4-715827885
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 715827886-1431655767
                Range node05 1431655768-2147483649
                Range node06 2147483650-2863311531
                Range node07 2863311532-3579139413
                Range node08 3579139414-4294967295
        </DataHandleRanges>
        <StorageHints>
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>

<Filesystem>
        Name pvfs2-alt
        ID 6000000
        TroveModule alt
        RootHandle 10
        <MetaHandleRanges>
                Range master2 4-715827885
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 715827886-1431655767
                Range node05 1431655768-2147483649
                Range node06 2147483650-2863311531
                Range node07 2863311532-3579139413
                Range node08 3579139414-4294967295
        </DataHandleRanges>
        <StorageHints>
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>
```

10.3.3 Five metadata and five data servers

```
<Defaults>
        UnexpectedRequests 50
        LogFile /tmp/pvfs2-kunkel.log
        EventLogging none
        LogStamp usec
        BMIModules bmi_tcp
        FlowModules flowproto_multiqueue
        PerfUpdateInterval 1000
        ServerJobBMITimeoutSecs 30
        ServerJobFlowTimeoutSecs 30
        ClientJobBMITimeoutSecs 300
        ClientJobFlowTimeoutSecs 300
        ClientRetryLimit 5
        ClientRetryDelayMilliSecs 2000
</Defaults>

<Aliases>
        Alias master2 tcp://master2:6555
        Alias node05 tcp://node05:6555
        Alias node06 tcp://node06:6555
        Alias node07 tcp://node07:6555
        Alias node08 tcp://node08:6555
</Aliases>

<Filesystem>
        Name pvfs2-fs
        ID 61878687
        TroveModule dbpf
        RootHandle 1048576
        <MetaHandleRanges>
                Range master2 4-143165581
                Range node05 143165582-286331158
                Range node06 286331159-429496735
                Range node07 429496736-572662312
                Range node08 572662313-715827885
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 715827886-1431655767
                Range node05 1431655768-2147483649
                Range node06 2147483650-2863311531
                Range node07 2863311532-3579139413
                Range node08 3579139414-4294967295
        </DataHandleRanges>
        <StorageHints>
                TroveSyncMeta yes
                TroveSyncData no
                AttrCacheKeywords datafile_handles,metafile_dist
                AttrCacheKeywords dir_ent, symlink_target
                AttrCacheSize 4093
                AttrCacheMaxNumElems 32768
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>
```

```
<Filesystem>
        Name pvfs2-tas
        ID 6187900
        TroveModule tas
        RootHandle 10
        <MetaHandleRanges>
                Range master2 4-143165581
                Range node05 143165582-286331158
                Range node06 286331159-429496735
                Range node07 429496736-572662312
                Range node08 572662313-715827885
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 715827886-1431655767
                Range node05 1431655768-2147483649
                Range node06 2147483650-2863311531
                Range node07 2863311532-3579139413
                Range node08 3579139414-4294967295
        </DataHandleRanges>
        <StorageHints>
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>


<Filesystem>
        Name pvfs2-alt
        ID 6000000
        TroveModule alt
        RootHandle 10
        <MetaHandleRanges>
                Range master2 4-143165581
                Range node05 143165582-286331158
                Range node06 286331159-429496735
                Range node07 429496736-572662312
                Range node08 572662313-715827885
        </MetaHandleRanges>
        <DataHandleRanges>
                Range master2 715827886-1431655767
                Range node05 1431655768-2147483649
                Range node06 2147483650-2863311531
                Range node07 2863311532-3579139413
                Range node08 3579139414-4294967295
        </DataHandleRanges>
        <StorageHints>
                HandleRecycleTimeoutSecs 5
        </StorageHints>
</Filesystem>
```

10.3.4 Server configuration

The configuration for each server looks similar - only the hostname is changed e.g. node05 instead of master2.

```
StorageSpace /tmp/pvfs2-kunkel
HostID "tcp://master2:6555"
```

List of Figures

Bibliography

[1] Withanage Don Samantha Dulip. Performance Visualization for the PVFS2 Environment. Bachelor's thesis, Ruprecht-Karls-Universität Heidelberg, Institute of Computer Science, Research Group Parallel and Distributed Systems, November 2005.

[2] Trinity Logic Inc. IC35L090AVV207-0. Online-document `http://www.tl-c.ru/pub/ccatalog?l=0&v=34&typ=3240`, 2003.

[3] Julian Martin Kunkel. Das parallele Dateisystem PVFS2. PDF-document `http://pvs.informatik.uni-heidelberg.de/Teaching/BACC-0304/`, January 2004. Slides of a talk at the Ruprecht-Karls-Universität Heidelberg.

[4] Julian Martin Kunkel, Thomas Ludwig, and Hipolito Vasquez. Weit verteilt - Dateisystem für parallele Systeme. *iX - Magazin für professionelle Informationstechnik*, (6):110–113, 2004.

[5] Jeff Layton. Cluster Monkey - Benchmarking Parallel File Systems. Online-document `http://www.clustermonkey.net//content/view/62/32/`, 2003.

[6] Sleepycat Sofware: Makers of Berkeley DB. Performance Metrics & Benchmarks: Berkeley DB. PDF-document `http://www.sleepycat.com/pdfs/wp_perf_0705c1.pdf`.

[7] Sleepycat Sofware: Makers of Berkeley DB. Getting Started with Berkeley DB for C. PDF-document `http://www.sleepycat.com/docs/gsg/C/BerkeleyDB-Core-C-GSG.pdf`, september 2004.

[8] PVFS Development Team. Parallel Virtual File System Version 2. PVFS2 Internal Documentation included in the source code package, 2003.

[9] PVFS Development Team. Trove Database + Files (DBPF) Implementation. PVFS2 Internal Documentation included in the source code package, 2005.

[10] PVFS Development Team. Trove: The PVFS2 Storage Interface. PVFS2 Internal Documentation included in the source code package, 2005.