The Tower of Pizzas[*]

Michael Tan Nick Roussopoulos Steve Kelley

University of Maryland Institute for
Advanced Computer Studies (UMIACS) and
Computer Science Department
University of Maryland
College Park, MD 20742
{mdtanx, nick, skelley}@cs.umd.edu

Technical Report CS-TR-3462 or
UMIACS-TR-95-52

May 1, 1995

Abstract

CPU speeds are increasing at a much faster rate than secondary storage device speeds. Many important applications face an I/O bottleneck. We demonstrate that this bottleneck can be alleviated through 1) scalable striping of data and 2) caching/prefetching techniques. This paper describes the design and performance of the Tower of Pizzas (TOPs), a portable software system providing parallel I/O and buffering services.

[*]This research was partially sponsored by NSF under contract number ASC 9318183, by ARPA under contract number F30602-93-C-0177, by ARPA through High Performance Computing Fellowship MDA972-92-J-1019, and by NASA under contract number USRA 5555-09.

1 Introduction

Forthcoming applications in multimedia, satellite observation, electronic libraries, and scientific computing will use datasets whose sizes are an order of magnitude larger than those handled by current systems. In 1992, global change researchers wanted to manipulate databases of 100 terabytes [Sto91]). Future satellites will download terabytes of data *every day*. Because the speed of secondary storage technology (disks) is increasing at a slower rate than CPU speed, I/O is predicted to be the bottleneck in many applications. With the increasing size of datasets and the increasing speed of CPUs, the means of getting the data to the CPU must also be increasing in *throughput*. Otherwise, much of the power of these future CPUs will be wasted. Gray [Gra94] has made the observation that if we count a cycle from a CPU of the future as a minute, the idle CPU cycles spent waiting for a single request to be fetched from disk is equivalent to the time to walk from the Earth to Pluto.

At the same time, many forthcoming computer systems resemble a cluster of workstations– groups of independant nodes with large memories connected by a fast network, where each node has a large memory and disk. At a supercomputing level, such machines include the IBM SP2 and DEC Sable. Even the workstation resources on a local area network can be harnessed to form a poor man's parallel processor ([ACPtNt95, BP94, BLL92]). Today's typical workstation has a 40 MHz CPU, 32 MB of memory, and .5 GB of disk space.

1.1 The Tower of Pizzas Solution

Data striping, buffering, and prefetching are methods which can help overcome the I/O bottleneck. Data striping is a widely accepted method of boosting I/O throughput [SGM86, CLG$^+$94, NBC$^+$94]. With striping, data can be read and written from multiple disks in parallel. Another way to alleviate the high cost of disk I/O is to avoid it or to hide it. By buffering larger chunks of data, some of the disk's mechanical head movement is avoided. Similarly, prefetching data while performing other tasks hides the cost of going to disk by reading data from the disk while the CPU is busy doing useful work.

We have constructed the Tower of Pizzas (TOPs), a multi-user, striped storage system implemented in software. The main goals of TOPs are 1) to provide parallel access to data striped across nodes/workstations, 2) to exploit caching and prefetching at the client and server to diminish latency, and 3) to be efficient and portable.

To meet these goals, a TOPs process runs on a set of workstations or nodes. Data can be striped across a set of the nodes, and then retrieved in parallel by users at any of the nodes. With respect to a given file which is striped over a set of nodes, we designate the nodes which store the file as server nodes.[1] We consider all other nodes to be client nodes. Client applications running on the client nodes talk to remote servers through the local TOPs process or through a linked library. The TOPs process also provides buffering and prefetching services.

TOPs has been implemented over the past year and we have now run it on a variety of hardware clusters: our 16-node SP2 (using TCP over the high-speed switch), and SPARCs and an Alpha. In preliminary tests on the SP2, TOPs demonstrates linear scalability of global throughput as servers and clients are added.

The system is implemented in software on top of general UNIX, which allows workstations of various flavors of UNIX to work together as clients and servers. The base requirements to run TOPs are general UNIX, TCP/IP, and POSIX compliant threads (pthreads). These requirements are met by most modern UNIX workstations, which makes TOPs very portable.

[1]The local and remote TOPs process are identical, which allows TOPS to be run as a peer-to-peer collection of workstations (like a distributed memory/file system, rather than just a partitioned set of clients and servers).

1.2 Goals

We now present a more detailed list of the major goals of TOPs.

Scalable I/O The global system throughput should scale up as more clients and servers are added to the system. The limit is reached when the network is saturated. An individual client node should see throughput scale-up as servers are added, up to the point where the node is saturated (network interface or software limitations).

Performance and portability The system should deliver high performance while running on general UNIX. Given the high portability and functionality we support, efficiency is a crucial issue in the design of TOPs. In the worst case, the path between a user and the remote data contains at least eight context switches, four passes through TOPs code, and six passes through the UNIX operating system code (user → TOPs → O/S → network → O/S → TOPs → O/S → disk → O/S → TOPs → O/S → network → O/S → TOPs → user).

Configurable buffering On a per file basis, the user should be able to control factors relating to the buffer management policies. These policies have substantial influence on the buffering performance of the local and remote server processes. The page size, allocation policy, and replacement policy should be easily derived from user-supplied information.

Prefetching Where possible, the system should prefetch data from disk and from remote nodes. Here we will present a simple technique which minimizes disk head motion, greatly improving disk throughput.

Configurable striping On a per file basis, the user should have control over the striping of the file. The number of servers to use and the server pool over which to stripe the data should be options available to the user.

CPU stewardship When multiple clients are using the same server process, the CPU should do as much work as possible and avoid unnecessary idling or busy waiting. For instance, if the server has a queue of client requests and reaches a point in one of the requests which blocks for I/O, the entire server process should not block. The server should start running some of the other pending client requests while the I/O is taking place. Likewise, lower priority requests, such as utility and cleanup tasks should not run before client requests.

Economony version Some applications will not need the more advanced features (shared buffer pool between clients on the client machine, asynchronous callbacks, etc.) of TOPs. These applications should be allowed to directly access the remote TOPs servers without going through an intermediate local TOPs process. We want to provide a lean client library interface for such applications. This allows applications to access the striped data without paying for the context switches in the full TOPs version.

2 TOPs Architecture

The Tower of Pizzas system architecture is a collection of nodes connected by a fast network. Each node has a CPU, large memory, and preferably one or more large, fast disks. Such a system could be a collection of workstations on a fast network, or a multiprocessor machine with a fast switch, and with private memory and disk at each node. Data can be stored on each node and retrieved from any other node. When some node n_i requests a transfer of data to or from some node n_j, we designate node n_i as the *local* node and node n_j and the *remote* node. If the client application which requested the transfer is running on node n_i, node n_i is designated to be local. Note that this peer-to-peer architecture looks similar to a distributed file system (figure 1).

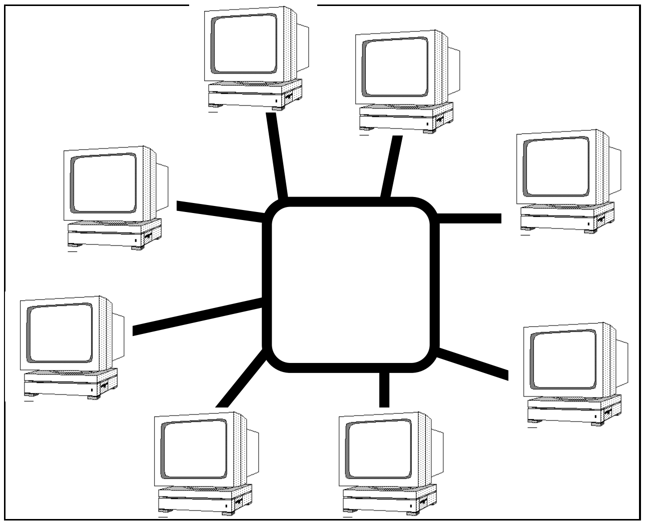

Figure 1: Tower of Pizzas system architecture. This figure depicts hardware configuration in which TOPs might be used in a peer-to-peer configuration.

A TOPs process runs on each node. This process is responsible for servicing requests, striping over the network, and managing several local resources (the buffer pool, disk I/O, and network I/O). Through knowledge of the access patterns (supplied by the user), the process exploits the local buffers to minimize disk I/O or network I/O. The TOPs process can service requests from two sources: a client application running on the same node, or a request received over the network. Client application requests may transfer data to and from the client application and the local disk, or to and from the client application and remote nodes. When a client application needs data which is stored remotely (i.e. a striped file), a request is given to the local TOPs process. The local TOPs process checks local buffers, and if the data is not buffered, it sends requests over the network to remote TOPs processes (running on remote nodes). Since these files may be striped, the local TOPs process manages the multiple remote connections and stripe mappings. When a TOPs process receives a request over the network, it looks for the data in its buffers. If the data is not buffered, the process reads it from the disk. The data is then sent back over the network to the requesting TOPs process. Note that opportunities for prefetching and caching are present in two places: in the local buffers and in the remote buffers.

If the client node is using the TOPs library interface, the flow of control is the same, except that there is no TOPs process on the local node. The functionality of the TOPs process is performed by the linked library code. Nodes running with only the library interface cannot receive requests over the network.

Because the striped files need to be visible to all of the client nodes, the meta-data for these files is managed by a file information server (FIS). The FIS mainains a catalog of the TOPs files, recording filename, page size, striping information, etc. The FIS is only accessed on major file actions (such as *open, close*, etc.), so *read* and *write* only contact the remote servers and do not cause interaction with the FIS. The FIS is simple enough to allow multiple FIS servers, should a single FIS become a bottleneck.

This system can be logically partitioned into a client-server architecture by designating some set of nodes to be servers, and the remaining nodes to be clients (figure 2). Data would be stored on the server nodes, and no client applications would run on these nodes. Conversely, client applications would run on the client nodes. The local disk of the client nodes would be used for temporary files and files used only within the client node, but the striped data files would be stored on the disks of

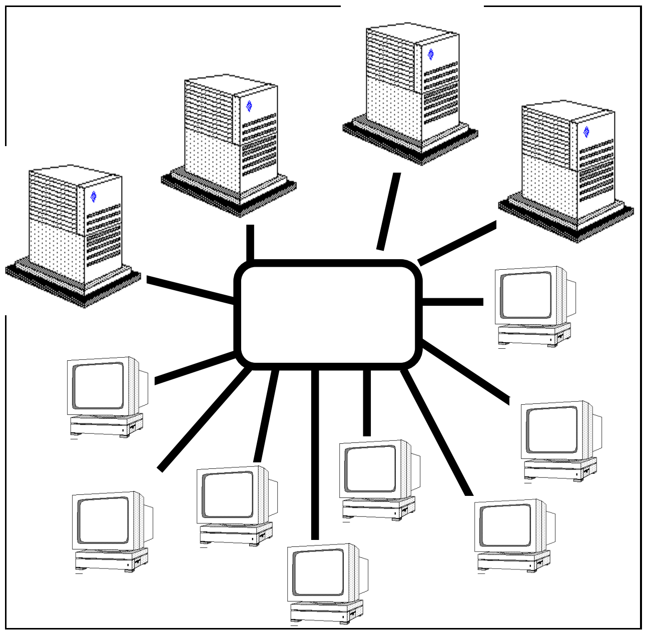

Figure 2: Tower of Pizzas system architecture. This figure depicts a hardware configuration in which TOPs might be logically divided into clients and servers.

the server nodes.

2.1 Software architecture of the TOPs process

The TOPs process (`tops-svr`) is responsible for most of the functionality of the Tower of Pizzas. This section describes the software architecture of `tops-svr` and its interaction with the UNIX operating system. Figure 3 illustrates the components of `tops-svr`.

2.1.1 Local application interface

The local client interface is used by client applications running on the same node as `tops-svr`. Through the interface, multiple client applications can simultaneously submit requests and receive responses. A given client application may submit one blocking request at a time (the client application blocks until the request is completed), or multiple non-blocking requests (the client application drops off several requests, and may come back later to check for completion and return values).

The client application and `tops-svr` use a combination of UNIX semaphores, FIFOs, and shared memory to manage the client's requests and `tops-svr`'s responses. Data pages and return values are passed through shared memory. Once a page has been allocated in the shared memory buffers, the client applications can directly access the page without interacting with `tops-svr` and without having to copy the page.

The local application interface includes several types of calls:

Local and remote disk I/O calls: `openfile()`, `closefile()`, `readpage()`, `writepage()`, `readpagerange()`, etc.

Buffer management calls: `getbuffer()`, `fastenpage()`, `unfastenpage()`, `flushbuffers()`, etc.

Striped-file mapping calls: `mapPageToServer()`

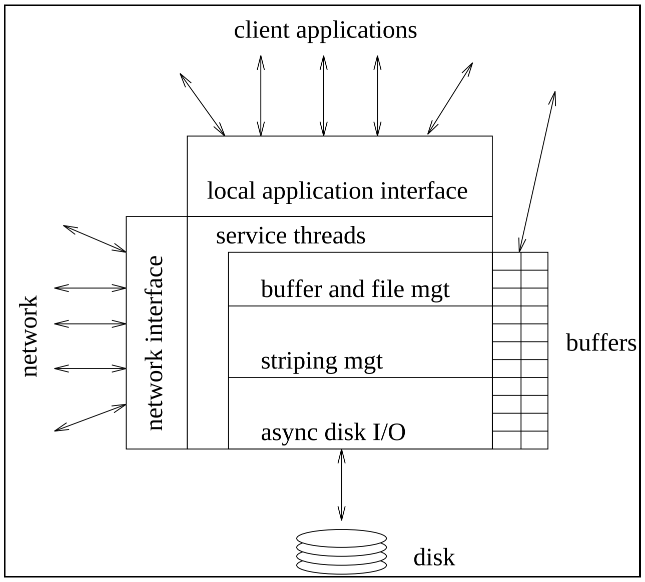

Figure 3: `tops-svr` software architecture

2.1.2 Network interface

Through the network interace, the `tops-svr`

- Sends requests to remote `tops-svr` processes
- Receives responses to its requests
- Receives requests from remote `tops-svr`s
- Sends responses to remote `tops-svr`s

`tops-svr`'s current network protocol is TCP/IP, through which it maintains a socket connection with each remote `tops-svr` with which it is interacting. Because of the need to preserve the sequentiality of remote requests, the network interface only releases one remote request per socket to the `tops-svr` service threads.[2] This means that `tops-svr` can service multiple requests simultaneously, as long as they were sent by different remote `tops-svr`s.

2.1.3 Threads and process control

An important feature of `tops-svr` is its ability to service multiple concurrent requests. When a request being serviced blocks (e.g. for disk or network I/O or a lock), other requests can continue running. An earlier version of `tops-svr` forked a process for each remote connection, but the cost of context switching was too expensive. We also experimented with using a separate process for disk I/O, but again found the cost of context-switching too expensive.

The current version of `tops-svr` is a single process which is threaded. For portability, we use POSIX-compliant threads (pthreads) which are readily available for SunOS, AIX, and OSF/1. Using pthreads calls, we implemented a set of `P()` and `V()` flavored primitives which allow `tops-svr`'s

[2] So if a server receives several `write` requests from another server, the writes are guaranteed to run in the order in which they were sent.

threads to voluntarily release control, block on conditional variables, and awaken blocked threads. This provides us with non-preemptive thread switching[3] which is easier to debug and program.

Because `tops-svr` is designed to run efficiently on the same CPU with other processes (such as client applications), there is no polling or busy waiting in `tops-svr`. When there is no work to do, `tops-svr` goes to sleep on a `select()` call. When a local request arrives through the FIFO or when a remote request arrives through a network socket, the `select()` call returns, waking up `tops-svr`. Likewise, a client application which has issued a blocking request to the local `tops-svr` does not poll or busy wait, but sleeps on a UNIX semaphore. When `tops-svr` has finished the request, it awakens the client application through the semaphore.

Currently, two types of threads run in `tops-svr`, although more (such as prefetching threads) are expected to be added. A single scheduler thread runs, and is responsible for reading requests from the client application interace and network interface, and dispatching the requests to client threads. The scheduler thread also checks various system queues for network I/O jobs or disk I/O jobs. Disk I/O jobs are picked up from the operating system when `tops-svr` is otherwise idle, or when the disk I/O queue is full.

The other thread type is the client thread. Client threads service requests received through the local client interface and through the network interface. These client threads may block while waiting for network I/O, disk I/O, and locked pages.

2.1.4 Buffer and file management

The buffer pool of `tops-svr` is primarily designed for buffering file pages. It supports multiple page sizes (where each file might have a different page size), and unique buffer allocation and replacement policies for each open file. In the current version, when a user opens a file, the following information is optionally submitted: access type (sequential scan, looping, random, etc.), maximum and minimum pages to buffer, and content type (image, text, etc.). This information can often be used to determine the optimal number of pages to allocate and the optimal replacement policy.[4] Hooks in the buffer manager code also allow a programmer to add custom replacement and allocation policies.

The freespace list is currently maintained by a doubly linked list. Contiguous regions of arbitrary sizes can be allocated from the buffer pool (in multiples of 1KB). Pages from the buffer pool can be pinned in memory (needed for multi-threading and for asynchronous release-and-flush operations). The buffer pool is kept in shared memory through UNIX `shm` or `mmap` calls (depending on the operating system) so that no page copying between a client application and `tops-svr` need occur.

When a file is opened, `tops-svr` maintains a file descriptor block for the file. This block stores buffering information, MRU and LRU page lists, an indexed pagelist, striping information (if appropriate), and access statistics. The striping information associated with a striped file includes the list of servers, information about the network connections, and the page mapping strategy (round-robin or custom).

Users are given a *handle* to a file, so that if more than one user opens the same file, only one instance of the file is open. This allows multiple users to share the same buffers for the file, which may improve performance and simplify consistency maintenance.

2.1.5 Asynchronous disk I/O

`tops-svr` supports asynchronous read and write calls, through threads and native (operating system specific) asynchronous disk I/O calls. When a thread requests a disk read or write, the thread blocks itself and control is passed to other threads. These other threads can run while the I/O takes place. For example, if a `readpage` request is running and blocks on disk I/O, another `readpage` request resulting in a buffer hit might run while the disk is accessed.

[3] Before another thread of the process can run, the running threads must *voluntarily* release control.

[4] TOPs automatically calculates the replacement policy and buffer allocation for the standard databse access patterns (sequential scan, looping-sequential, looping randone, etc.).

2.1.6 Network I/O

`tops-svr` uses TCP/IP to transfer data over the network. For each remote connection, a read queue and write queue are maintained. Between two interacting `tops-svrs`, a pair of sockets is maintained– one socket for control messages and the other for miscellaneous messages.). We expect a second socket to be needed as more advanced features are added to TOPs. The semantics of the socket queue are complex (and still evolving) because of several constraints: sequentiality of requests must be maintained, double buffering must be avoided, and the FIFO nature of reading data from a socket (vs. message passing) requires an entire message to be read from the socket before the next one can be read.

2.2 Prefetching for large sequential reads

TOPs has prefetching support for global sequential reads. Consider the case where multiple clients are going to read through a single region of a file and disjointly partition the data among themselves (e.g. Client A will request pages 1, 4, 7, 10, etc., and client B will request pages 2, 5, 8, 11, etc., and client C will request pages 3, 6, 9, 12, etc.). Because we want to handle asynchronous arrival of these requests, the server has a problem. The page requests may arrive in an order such as 2, 3, 1, 6, 5, 9, 4, 8, 7,... Globally, the requests are arriving in an order which approximates a single sequential pass through the file.

Our tests show that when these requests are read from disk the exact order in which they arrive (a roughly sequential access), the filesystem and disk throughput on the SP2 is about 1 MB/s. However, our measurements also show that an SP2 filesystem and disk are capable of delivering about 3.5 MB/s when the accesses are *exactly* sequential (e.g. 1, 2, 3, 4, 5, 6, ...).

As a solution to this, we devised a simple and effective prefetching technique. When the file is opened, the clients declare that they intend to access the file in a global sequential read pattern. The server keeps track of the last page of the file which was fetched from disk (page p_i). When the next request for a page arrives (page p_j), the server takes one of two actions. If $i < j$, then the server reads pages p_{i+1} through p_{j+k} (k is some arbitrary constant) into the buffers. Page p_j is then sent to the client. If $i > j$, then page p_j should already be buffered (unless so many pages past p_j have been read that p_j was already flushed from the buffers).

Conceptually, we are buffering a sliding window of the file. The buffered window moves sequentially through the file as pages are added and swapped out. As long as this window is large enough, client requests can asynchronously arrive and either fall into this buffer window or incrementally extend the window forward, avoiding any non-sequential disk access.

There are two main advantages with this method:

1. Disk I/O is purely sequential (and no unnecessary disk I/O is done if there are enough buffers).
2. Asynchronous access is supported. We do not require clients to synchronize their accesses with each other in order to present a purely sequential request stream to the filesystem and disk.

Very good performance is obtained by this method. Results are reported in a later section.

2.3 Platform-specific notes

There are a few minor variations between TOPs versions for different operating systems. On some versions of AIX and OSF/1, there is a bug in the `select()` and FIFO interaction which requires some maneuvering to circumvent. Because pointers are stored and accessed in the shared memory, it is desirable for all of the shared memory to be allocated in a single `mmap` or `shm` segment. The default maximum `shm` size in SunOS and OSF/1 was too small for our tests, but the `mmap` implementation in some versions of AIX does not allow asynchronous disk I/O involving `mmap`ped memory. Thus, for some platforms `shm` is used, and for some platforms `mmap` is used. Also, differences in the conceptual model and calling semantics for the various platforms' asynchronous disk I/O calls caused variation.

3 Performance

This section presents results from preliminary tests conducted on a 16-node IBM SP2.[5] The tests are organized into four parts. In the first part, we test the SP2 performance (network throughput, and filesystem and disk throughput). These tests provide us with a baseline with which to estimate the performance of TOPs. The second part is comprised of tests which measure TOPs overhead. These tests show the maximum throughput that TOPs can deliver if all data is in the TOPs buffers. In the third part, the tests use large files (150MB per server) and show performance of TOPs under random access workloads. The final set of tests show the performance of TOPs under a global sequential read workload.

The tests in the last three parts test actual TOPs throughput. In these tests, the nodes are partitioned into clients and servers. Data files are broken into 8 KB pages and are striped across the servers' disks using round-robin striping (e.g. With two servers, the first 8 KB gets sent to server 1, the second 8 KB gets sent to server 2, the third 8 KB gets sent to server 1, etc.). The tests consist of read-only workloads or write-only workloads.

In these TOPs tests, the client reads/writes in sets of 16 pages (128 KB total), regardless of the number of servers. Thus, for a 1 server configuration, each client reads/writes a 16 page set to the server, and for a 8 server configuration, each client reads/writes a 16 page set to the servers (2 pages per server). As more servers are added, we expect system throughput to increase, but it should be noted that in these tests, as the number of servers increase, the amount of pages requested from a single server by a singe client decreases, since the clients request a fixed amount of pages. We also note that even if clients are requesting a set of sequential pages from a server, as more clients are added to a system, the disk access pattern will become more and more random since the server may be servicing more than one client request at a time. In such a case, the pages requests from the multiple clients are more likely to be interleaved as more clients are added to the system.

A client does no processing on a page except to check a few header bytes when a page is read from a server, or to write a header and some dummy data into a page when a page is to be written to a server. When a client reads a set of pages from the server, the client blocks until all pages arrive. A context switch to the local server is incurred here. When a client writes a set of pages to the server, the client may or may not wait for an ack message from the server (depending on the test). Additionally, in these tests, a server is given a buffer pool of 1000 pages (about 8 MB) and a client is given a buffer pool of 16 pages.

In these tests, clients are started simultaneously, At the end of each test, the sum of the throughput observed by each client is computed, giving the total system throughput.

3.1 SP2 Performance

Two potential bottlenecks for TOPs performance are the network throughput and the filesystem/disk throughput. In this section we present the performance of these parts of the SP2.

3.1.1 SP2 TCP Performance

The SP2 nodes are connected by a high-performance omega network switch, providing point-to-point throughput of 45 MB/s (using user space PVM). We wrote a set of C programs which used TCP to send streams of 8KB messages from the memory of one node to the memory of another node. By timing the execution of this program, we measured the point-to-point throughput for TCP over the switch to be about 12 MB/s. This throughput is probably lower that that of user-space PVM because of the multiple copies made by TCP. Running up to eight of these point-to-point TCP tests simultaneously, we observed a not-quite-linear scaleup. Figure 4 shows global switch throughput reaching 68 MB/s for eight simultaneous point-to-point transfers.

[5]Each node on this SP2 consisted of an RS/6000 processor, 64 MB of memory, and a local disk.

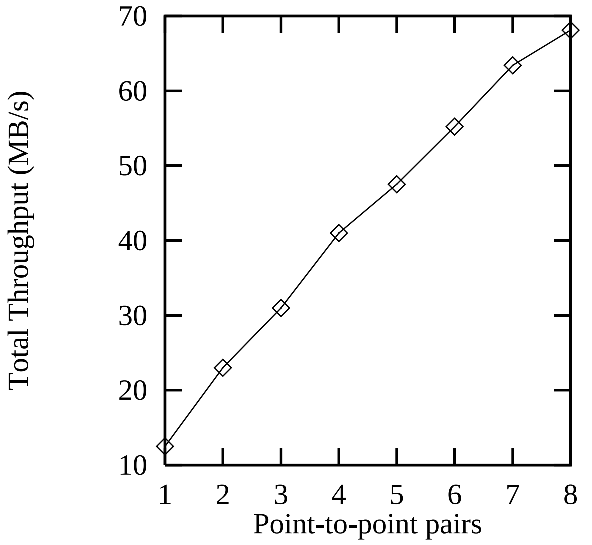

Figure 4: SP2 switch TCP test

3.1.2 SP2 Filesystem/Disk Performance

Access Pattern	*Throughput (MB/s)*	
	Read	Write
sequential	3.5	3.7
looping sequential	3.5	3.7
range read (16 pages)	3.4	2.6
range read (2 pages)	1.1	1.1
random	0.8	1.0

Table 1: SP2 filesystem and disk throughput under various access patterns

In order to have a baseline of filesystem and disk performance, we ran a series of tests on a single SP2 node's filesystem and disk. We wrote a C program which accessed a large file using UNIX `read()/write()` and a variety of file access patterns (table 1). In all of these tests the unit of transfer that we read/wrote was 8KB and the filesize was about 160 MB. The sequential test consisted of a single sequential pass through the file, in which each page was read/written one at a time. The looping sequential test sequentially read/wrote through the entire file several times. This test was done to reveal any benefits of file caching that might have been reflected in the one-time pass of the sequential test. The range read test looped many times and in each iteration randomly picked some spot in the file at which it read/wrote a set of (16 or 2) sequential pages. The random test simply read/wrote a large number of random pages. The tests clearly show that greater sequentiality in the access pattern improves disk throughput. For purely sequential access the read/write throughput through the filesystem is about 3.5 MB/s. For completely random access to the file, the filesystem and disk could read (write) about .7 (.9) MB/s.

3.2 TOPs Overhead

In this set of tests, we measure the amount of overhead in the TOPs software. The files are small enough (8 MB/server) that they are completely cached in the TOPs buffers. Thus, all client accesses result in buffer hits, and no disk I/O occurs (except to initially read the file into the buffers). In these tests clients read/write 1000 sets of 16 pages (thus, each client transfers a total of (8 KB/page * 16 pages * 1000 iterations = 128 MB). Based on the TCP maximum throughput measurements from the previous section, we expect that the upper bounds on throughput will be 12 MB/s per server.

These tests represent the maximum throughput delivered by TOPs when all the data to be accessed by clients is already in the server buffers. As such, the throughput represents an absolute upper bound on TOPs server performance. With 8 servers, we show that TOPs delivers about 40 MB/s in this situation.

3.2.1 TOPs buffered read test

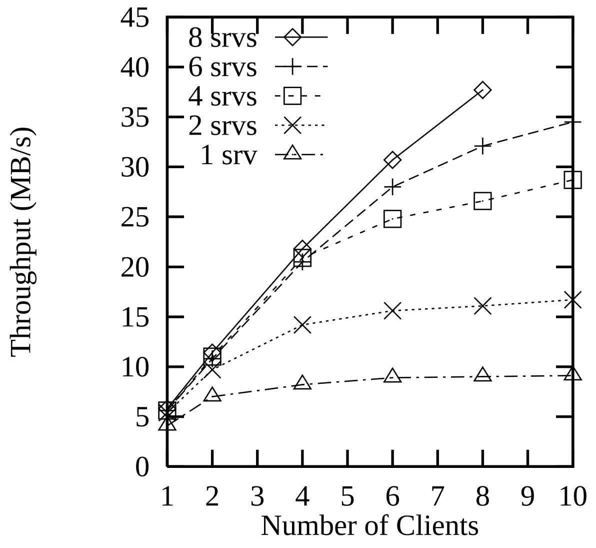

Figure 5: TOPs read test on completely buffered file.

In this test, clients read data from the servers. The results (figure 5) show that a single server can deliver up to 7.6MB/s and that a single client (with a local TOPs process) can receive 5.5MB/s. Remember that in these tests, context switches between the client application and local TOPs processes occurred every 128KB. In addition, these results are specific for the given TCP message size (we used 8KB pages). For each server configuration, the total system throughput scaled up as more clients were added, up to the point where the servers were saturated with requests.

For a fixed number of clients (8), we note that the throughput scaleup is not perfectly linear as more servers are added. However, because the request size *per server* decreases as more servers are added, and because clients wait for all pages to be delivered before making the next request, we expect that this would diminish the throughput per server as more servers are added.

3.2.2 Library interface TOPs read test on completely buffered file.

This set of tests (figure 6) was performed under the same conditions as the previous test, but used a linked library version of the TOPs process on the client machines. The tests demonstrate that the linked library version allows the system to reach its maximum throughput more quickly (that is, with fewer clients), which was expected since the linked library interface enables the client application to make requests directly to the remote servers (without interaction (and context-switches) with a local `tops-svr`).

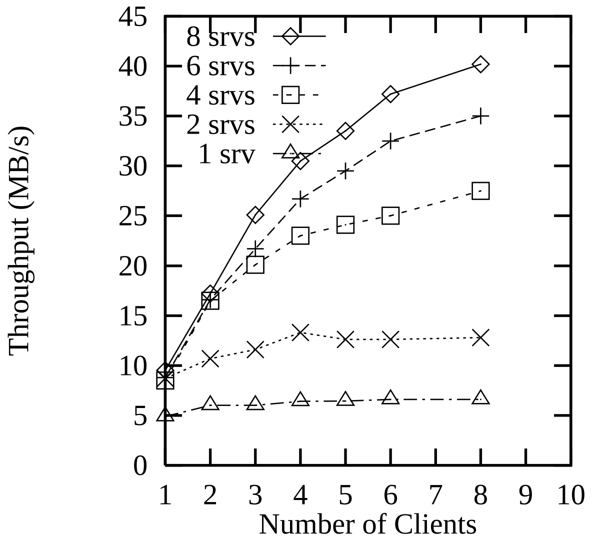

Figure 6: Library interface TOPs read test, file is completely buffered.

3.2.3 TOPs buffered write test

This test (figure 7) is the same as the small file read test described above, except that the clients are now writing, not reading, 16 pages at a time. All pages were written into the buffers and no pages were forced to disk. Because the writes here were asynchronous, the clients did not need to wait for an acknowledgement before sending the next set of pages. As expected, this allowed the system to reach its maximum with fewer clients.

3.3 TOPs Performance with Random Access Workloads

These tests represent the worst case performance of TOPS. They demonstrate that under random access to uncached data, disk performance severely limits system performance. In these experiments, each client application generated 1000 requests. A request consisted of 16 consecutive pages (128KB), starting at some random point in the file. The data files were sized so that each server stored a 150MB portion of the file (e.g. for one server the total data file was 150MB, for two servers the file size was 300MB, for eight servers the file size was 1200MB, etc.). Thus, between two requests, disk head motion was expected. Each client was started with a different seed for its random number generator.

There were two independant factors in these experiments, each increasing the degree of non-sequential disk access. These factors must be considered because depending on the amount of sequentiality in a disk reference string, disk throughput can range from .7 MB/s to 3.5 MB/s, as measured in table 1. The first factor affecting sequentiality was the number of servers. Because each client requested a fixed amount of pages (16), the amount of sequential pages a client requested per server decreased as the number of servers increased (up to 16 servers). The second factor affecting the amount of sequentiality was the number of clients. As more and more clients were added to a fixed size server configuration, the sequentiality in a given client's request was lost because TOPs serviced multiple client requests by interleaving the page requests.

Thus, for a fixed number of clients, we expect the throughput per server to decrease (down to the random access performance of the disk) as the number of servers is increased. Likewise, for a fixed number of clients, we expect the throughput per server to decrease (down to the random access performance of the disk) as the amount of clients increases.

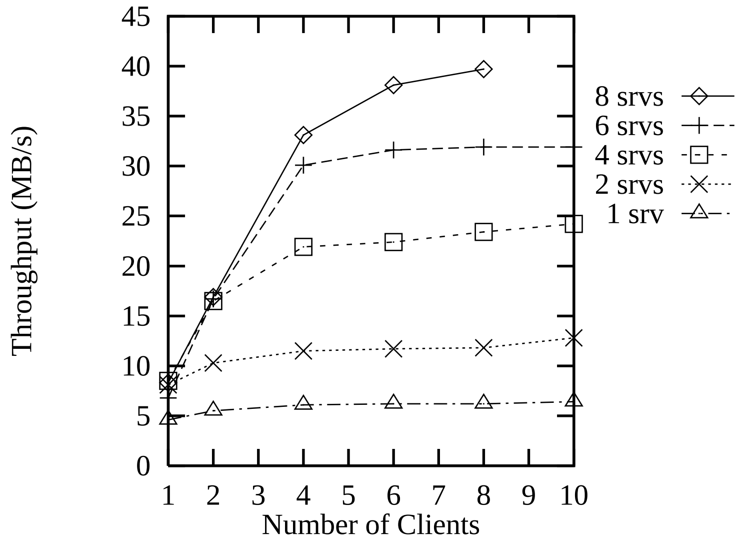

Figure 7: TOPs write test on completely buffered file.

3.3.1 TOPs read test, random ranges

In this test (figure 8) clients issued only read requests. Based on the performance of the SP2 filesystem and disk under random access patterns (see table 1), we would expect to see no more than 0.8 MB/s delivered per server (in 8 client tests). The set of throughput measurements for the 8 client tests reflect this.[6] As predicted in the previous section, we see that for 8 clients, the throughput per server for 8 servers (0.7 MB/s) is lower than the throughput per server for 1 server (1.0 MB/s). However, the 1 server throughput would probably drop to meet the 8 server throughput as more and more clients were added. Likewise, for the 1 server tests, we see that server throughput drops as more clients are added and sequentiality decreases. For the 8 server tests, the request stream per server is almost completely random with ony a few clients, so we do not expect to see it drop as client are added. However, we do see it rise (up to a point) as clients are added, since with a few clients, the 8 servers are not fully utilized (unlike the 1 server case where the server is fully utilized with only one client).

Although the throughputs in this test are much lower than the previous memory-only tests, two points should be noted:

1. The throughput in this test is close to the maximumpossible throughput for the given workload (with random read, an SP2 disk can deliver a maximum of 0.8 MB/s on a 160MB file).

2. Throughput scales up as servers are added.

3.3.2 TOPs random range write test, large file

In these tests (figure 9), the page reference stream was similar to the previous test except that clients only wrote pages. However, there were significant differences in this test. The file was logically partitioned between the clients so that clients wrote to disjoint locations of the file. This partioning was done in such a way that a client still accessed all of the servers.

[6]With a buffer pool of 1000 pages at each server, we measured the amount of TOPs buffer hits in these tests to be about 5%. However, performance was not significantly changed when the tests were re-run with a smaller buffer pool and a hit ratio of about 0.1%.

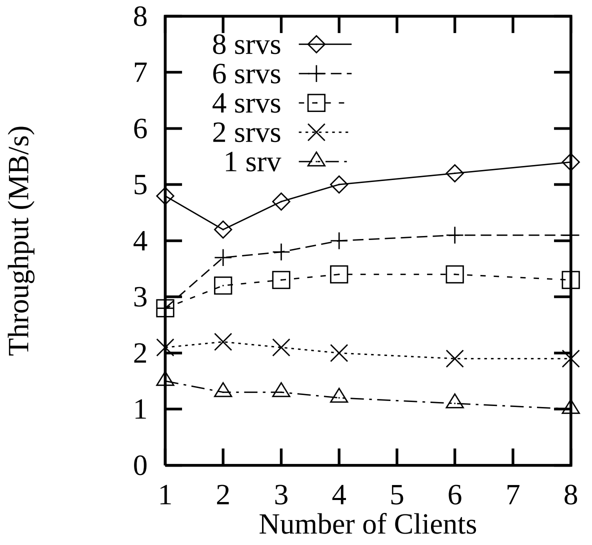

Figure 8: TOPs read test on large file using random ranges.

Another difference in this test was that the clients were sending (vs. receiving) data to the servers faster than the data could be written to the server's disk. From figure 7 we see that with 8 servers and 8 clients, the clients could send 40 MB/s to the servers to be written. But from table 1, we know that that a server's disk could only write a maximum of 1-3 MB/s (depending on the amount of sequentiality). With 8 servers' disks running in parallel, only 8-24 MB/s could be written to disk. Thus, if clients were unrestrained, they sent data to servers more quickly than the servers could handle. In the read tests of the previous sections, the clients were not flooded with data from the servers because clients explicitly requested any data that was to be sent to them. In the read case, clients always allocated enough buffer space before requesting and receiving data.

In addition, there was not enough physical memory on the servers to buffer the data received from the clients (each client was writing out a total of 1000 sets * 16 pages/set * 8 KB per page = 128 MB). Because of these two factors, the operating system would thrash when this test was run. To remedy this, it was necessary to add a flow control for network writes. For this test, we made a client wait for a single ack, after it had sent out the 16 pages to be written. On the server side, an `fsync(fd)` was issued after every 16 writes to the file took place. The `fync` was used to avoid taking up too much memory with file system buffering. The client ack ensured that the servers did not becaome glutted with client data. The results presented here (figure 9) reflected this ack-`fsync`ed write test.

Again, we see in these tests that throughput scales up as servers are added, and that the servers deliver close to the maximum possible (based on the SP2 disk performance for random access writes).

3.4 TOPs sequential global read test

In this test (figure 10), each client made a sequential pass through the file, but each client requested unique pages (e.g. with 3 clients (A, B, and C), A requested pages 1-16, B requested pages 17-32, C requested pages 33-48, A requested pages 49-64, etc.). As before, each client blocked until the pages from its last request were all delivered, then checked the pages, and then made the next request. However, there was no synchronization between the clients, so there was no guarantee of the order in which the page requests arrived at the servers.

When this test was run without the group-sequential-read prefetching mechanism, performance looked similar to that of figure 8. But with the prefetching mechanism, sequential access to disk is

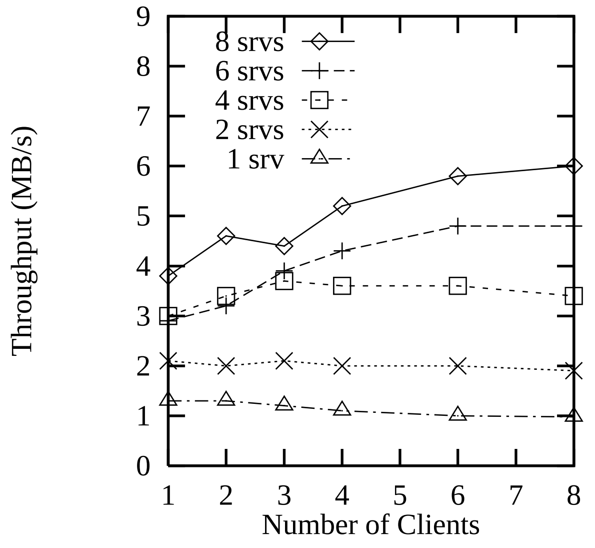

Figure 9: TOPs write test on large file using random ranges.

guaranteed, if the buffer window is large enough. From the figure 10, we see that each disk delivered about 3MB/s, which is consistent with the sequential read tests on the SP2 disk (table 1).

4 Related Work

A number of projects have contributed to the research in scalable I/O. Parallel file systems such as [HO93, Mon93, MS94] can run on workstation clusters. Zebra [HO93] has a client-server architecture and is implemented on the Sprite operating system, using a log structured file system to store files on the servers. Zebra also implements parity-codes in its striping (RAID level 4), but does not address buffering or prefetching issues. The PIOUS [MS94] system uses PVM to communicate over the network, targeting workstation clusters or parallel clusters. PIOUS stripes files and supports a form of transactions (which guarantee file system integrity in case of a crash) and global shared files pointers. [MS94] reports results for ethernet tests. PIOUS also does not focus on buffering and prefetching issues.

In the realm of parallel processing, several parallel file systems with interesting ideas have emerged [Kot94, DSE88, DS89, FBD94, CBF93, TBC+94]. These systems are often geared toward matrix operations, and a number of descriptors to encapsulate typical access patterns are described [Kot94, CBF93]. SPIFFI [FBD94] implements global shared file pointers of varying flavors (multiple clients access the same file pointer). SPIFFI also implements a form of prefetching to handle the out of order requests generated when multiple clients sequentially access the same file. Their approach seems to use larger disk blocks and some buffering to handle out of order requests. However, it is unclear whether SPIFFI completely preloads all the pages in the "gap" between two non-consecutive page requests. In [Kot94], I/O requests are all passed to the I/O nodes, which sort and perform the requests in some order favorable for disk head motion (disk directed I/O). This work focuses on predicting the next page reference, whereas TOPs deals with a known access pattern. JOVIAN [BBS+94] provides ideas for dealing with many small references by conglomerating the references into disk block requests, but doesn't deal with buffering/prefetching. ADOPT [SC94] is a parallel I/O system which provides for user level and compiler level generation of prefetch hints, including conditional I/O requests. The prefetching is based on explicit requests for pages and does not deal with a group sequential read prefetch scheme such as the one outlined in this paper.

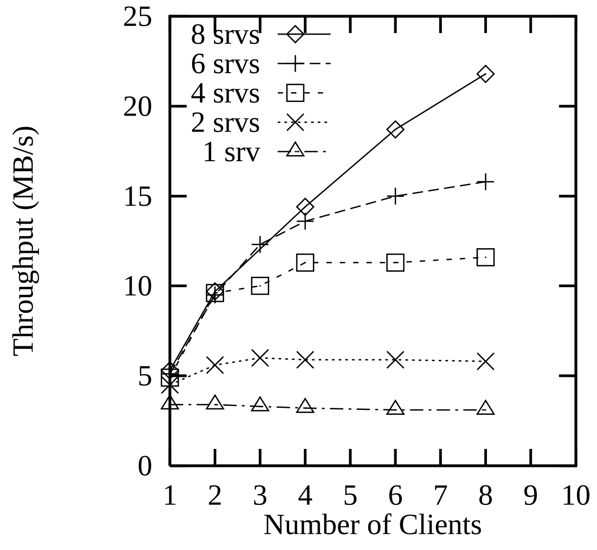

Figure 10: TOPs global sequential read test.

Another approach to scalable I/O is to argue that scalable I/O must also be addressed at a lower level. RAID-II [CLD+93] views the network as a backplane, with several RAID-5 devices directly connected to a HIPPI network and ethernet. RAID-II seeks to deliver high bandwidth for large requests and low latency for small requests. Other projects stripe across networks and tape drives [Sto91, DK93]. However, with such low-level approaches, there is less user control over buffering and prefetching policies. In addition, it can be argued that software striping can always be used to stripe over a collection of such devices.

5 Future Work and Conclusions

We have implemented the Tower of Pizzas, a scalable I/O system in software, exploiting parallel I/O and buffering. The system is designed for collections of nodes connected by a fast network. Data is striped across multiple servers to increase read and write throughput. We also describe the design and performance of a prefetching scheme for global sequential reads. Our current implementation suggests that high-level software striping systems can deliver high performance without sacrificing portability.

In future work, we plan to experiment with adaptive prefetching techniques, using TOPs as a testbed. We also anticipate examining custom striping strategies for heirarchical scientific data. As TOPs continues to develop, we plan to integrate further database functionality, such as concurrency control. Finally, we are connecting TOPs to an actual database system and experimenting with performance gains for databases using a TOPs system (in a distributed or client-server configuration).

References

[ACPtNt95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for NOW (networks of workstations). *To appear in IEEE Micro*, 1995.

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[BLL92] Allan Bricker, Michael Litzkow, and Miron Livny. Condor technical summary. Technical Report CS-TR-92-1069, University of Wisconsin - Madison, 1992.

[BP94] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of worstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, August 1994.

[CBF93] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 7–14.

[CLD+93] Peter M. Chen, Edward K. Lee, Ann L. Drapeau, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, Ken Shirriff, David A. Patterson, and Randy H. Katz. Performance and design evaluation of the RAID-II storage server. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 110–120, 1993.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[DK93] Ann L. Drapeau and Randy H. Katz. Striping in large tape libraries. In *Proceedings of Supercomputing '93*, pages 378–387, 1993.

[DS89] Peter C. Dibble and Michael L. Scott. Beyond striping: The Bridge multiprocessor file system. *Computer Architecture News*, 19(5), September 1989.

[DSE88] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[FBD94] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. Submitted to IEEE TPDS, 1994.

[Gra94] Jim Gray. Parallelism: The new imperative in computer architecture. Slides from talk at VLDB 1994, 1994.

[HO93] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 1993.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994. Revised November 8, 1994.

[Mon93] Bruce R. Montague. The Swift/RAID distributed transaction driver. Technical Report UCSC-CRL-93-99, UC Santa Cruz, January 1993.

[MS94] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: a RISC machine sort. In *Proceedings of the 1994 ACM SIGMOD*. ACM Press, 1994.

[SC94] Tarvinder Pal Singh and Alok Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[Sto91] Michael Stonebraker. An overview of the Sequoia 2000 project. Technical Report 91-5, University of California, Berkeley, 1991.

[TBC$^+$94] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.