8-1995

Disk-Directed I/O for an Out-Of-Core Computation

David Kotz
Dartmouth College, David.F.Kotz@Dartmouth.EDU

Disk-directed I/O for an Out-of-core Computation

David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
dfk@cs.dartmouth.edu

Abstract

New file systems are critical to obtain good I/O performance on large multiprocessors. Several researchers have suggested the use of collective *file-system operations, in which all processes in an application cooperate in each I/O request. Others have suggested that the traditional low-level interface (*`read, write, seek`*) be augmented with various higher-level requests (e.g.,* read matrix*). Collective, high-level requests permit a technique called* disk-directed I/O *to significantly improve performance over traditional file systems and interfaces, at least on simple I/O benchmarks. In this paper, we present the results of experiments with an "out-of-core" LU-decomposition program. Although its collective interface was awkward in some places, and forced additional synchronization, disk-directed I/O was able to obtain much better overall performance than the traditional system.*

1 Introduction

Although multiprocessor systems have increased their computational power dramatically in the last decade, the design of hardware and software for I/O has lagged and become an increasing bottleneck in the overall performance of parallel applications. The use of disk striping to access many disks in parallel has alleviated some of the hardware limitations by providing greater capacity, bandwidth, and throughput. Good parallel file-system software, however, is critical to a system's I/O performance, and early file systems often had disappointing performance [18].

Recent work shows that if an application could make high-level, collective I/O requests, the file system can optimize I/O transfers using disk-directed I/O [12] to improve performance by orders of magnitude. In [12], however, experiments were limited to simple benchmarks that read or wrote matrices. In this paper we evaluate the performance of disk-directed I/O on a much more complex program, an out-of-core LU-decomposition program. This program allows us to understand the performance benefits of disk-directed I/O in a full program that performs computation, reads and writes the same file (indeed, rereads and rewrites the same file many times), and has interprocess synchronization.

In the next section we provide more background information. Section 3 discusses the LU-decomposition program. In Section 4 we describe a set of experiments used to reinforce our discussion, and Section 5 provides the results. We conclude with commentary on the advantages and disadvantages of high-level, collective requests, and on the underlying technique of disk-directed I/O.

2 Background

File systems. There are many parallel file systems today, including Intel CFS and PFS [19], IBM Vesta [5], TMC [1], and HFS [15], to name a few. There are also several systems intended for workstation clusters, such as PIOUS [16], and VIP-FS [10]. All of these systems decluster file data across many disks to provide parallel access to the data of any file. A full characterization of these systems is not possible here due to space limitations, but [9] presents a reasonable summary.

Workload. The CHARISMA project traced production parallel scientific-computing workloads on an Intel iPSC/860 [14] and on a TMC CM-5 [20] to characterize their file-system activity. In both cases, applications accessed large files (megabytes or gigabytes in size) using surprisingly small requests (on the Intel, 96% of read requests were for less than 200 bytes). On further examination, we discovered that most of the files were accessed in complex yet highly regular patterns [17], most likely due to accessing multidimensional matrices.

Interfaces. Most parallel file systems present the traditional abstraction of a file as a sequence of bytes with Unix interface semantics, and add a few extensions to control the behavior of an implicit file pointer shared among the processes. This low-level interface, which restricts each request to a contiguous portion of the file, is one reason for the predominance of small requests found by the

This research was supported by Dartmouth College, by NSF under grant number CCR 9404919, and by NASA Ames Research Center under Agreement Number NCC 2-849.

CHARISMA project. Higher-level interfaces, such as specifying a strided series of requests [17], accessing data through a mapping function [5, 7, 4], or using an object-oriented interface [15, 11, 21], provide valuable semantic information to the file system, which can then be used for optimization purposes. Interfaces that allow the programmer to express *collective* I/O activity, in which all processes cooperate to make a single, large request, provide even more semantic information to the file system.

Unfortunately, few multiprocessor file systems provide a collective interface. CM-Fortran for the CM-5 does provide a collective-I/O interface, which leads to high performance through cooperation among the compiler, run-time system, operating system, and hardware. The MPI message-passing interface may soon be extended to include I/O [4], including collective I/O. Finally, there are several libraries for collective matrix I/O [23, 11, 21].

Two-phase I/O. Two-phase I/O is a technique for optimizing data transfer given a high-level, collective interface [8]. A library implementing the interface breaks the request into two phases, an I/O phase and a redistribution phase. When reading, the compute processors cooperate to read a matrix in a "conforming distribution", chosen for best I/O performance, and then the data is redistributed to its ultimate destination. When writing, the data is first redistributed and then written in a conforming distribution. There are no published performance results for an out-of-core application using two-phase I/O.

Disk-directed I/O. Disk-directed I/O is a technique for optimizing data transfer given a high-level, collective interface [12]. In this scheme, the complete collective, high-level request is passed to the I/O processors, which examine the request, make a list of disk blocks to be transferred, sort the list, and then use double-buffering and special remote-memory "get" and "put" messages to pipeline the transfer of data between compute-processor memories and the disks. Compared to a traditional system with caches at the I/O processors, this strategy optimizes the disk accesses, uses less memory (no cache at the I/O processors), and has less CPU and message-passing overhead. In experiments with reading and writing one- and two-dimensional matrices, disk-directed I/O was as much as 18 times faster than traditional caching in some access patterns, and was never slower [12]. One implementation of a similar technique led to excellent I/O performance on an IBM SP-2 multiprocessor [21].

3 LU decomposition

LU decomposition represents the bulk of the effort in one technique for solving linear systems of equations. An $N \times N$ matrix M is decomposed into two matrices, a lower-triangular matrix L and an upper-triangular matrix U, such that $LU = M$. Typically, these two triangular matrices are stored in one $N \times N$ array, occupying disjoint elements of the array. Indeed, the decomposition can be done in place, overwriting M. A sequential algorithm (with no pivoting) looks like this:

```
for i = 1 to N-1
    // update rows i+1 .. N
    for j = i+1 to N
        mult(j) = M(j,i) / M(i,i)
        // row j: update cols i+1 .. N
        for k = i+1 to N
            M(j,k) -= mult(j) * M(i,k)
        end
    end
end
```

One simple parallelization of this algorithm (although not the best; see [24] for a better algorithm) is to distribute responsibility for columns of the matrix among P processors in a cyclic pattern; that is, column k is handled by processor $(k \bmod P)$ (see Figure 1). In iteration i, the multipliers (called $mult(j)$ above) are computed from column i by processor $(i \bmod P)$ and then broadcasted to the other processors. Then each processor updates the columns for which it is responsible; only in the last few iterations is any processor idle.

Figure 1: Example of column-cyclic distribution of 16 columns across four processors. Each processor is represented here by a different shade of gray. SLAB_COLS is 2 here, meaning each processor allocates space for two columns in main memory. The combined slab size is eight columns.

When the matrix is moderately large, that is, too large to fit in memory but small enough so that each processor's

memory can hold at least one column of the matrix, the processors repeatedly read a subset of their columns from the file, update those columns, and then write those columns back to the file. Thus, it makes sense to store the matrix in column-major order. We call each processor's subset of columns a "slab." Note that because of the cyclic distribution any one processor's slab is not contiguous in the file, but that the set of corresponding slabs for all processors collectively represent a contiguous set of bytes in the file.

The code for parallel, out-of-core LU-decomposition (based on that in [22]) is shown in Figure 2. There are several things to note about this program. First, note the optimization to split the outer loop into two loops, with the I/O pulled out of the second loop. The second loop begins once the remaining columns all fit in memory, eliminating many unnecessary I/O transfers; indeed, when the entire matrix fits in memory the first loop is ignored and we need only load and store the matrix once. Second, the nodes synchronize as part of the multiplier calculation, because one node computes the multipliers and broadcasts them to the other nodes. (In my implementation this broadcast involves a barrier synchronization.) Third, the code is written so that all processors make the same number of iterations through all loops, even though in the last few iterations some processors will have `ncols = 0`, so that collective communication and I/O routines can be used if desired. The performance cost of extra iterations is negligible, because those processors with fewer iterations eventually wait for those with more iterations anyway. Finally, the program explicitly waits for all pending writes to fully complete (`sync()`) before stopping the clock.

When based on a traditional file-system interface, the function `LU_read` is a loop over calls to `seek()` and `read()`, one iteration for each column in that processor's slab. No inter-process synchronization is necessary. `LU_write` would be similar. Given a collective file-system interface, these functions are rewritten to synchronize all processors at a barrier, and then call a disk-directed I/O routine to transfer the contiguous slab representing the combination of individual disjoint slabs (as in Figure 1). The "extraneous" synchronization of a collective interface would in general accentuate temporary load imbalances, but it can often allow dramatically better I/O performance.

Finally, we note that code like that in Figure 2 could be written by hand, incorporated in a parallel matrix library [2, 21], or generated by a smart compiler [6, 22].

4 Experiments

To gain a better understanding of the benefits of disk-directed I/O to an application like LU decomposition, we ran several experiments. In these experiments, we ran the program in Figure 2 with both the "traditional caching" file system and the disk-directed file system, on top of our parallel file-system simulator called STARFISH [12]. STARFISH is based on the Proteus parallel-architecture simulator [3], which runs on a DEC-5000 workstation. It does not model any particular multiprocessor architecture or operating system, but we configure it to behave like a machine of contemporary technology. It includes an extremely accurate disk-drive model, and uses Proteus to count instructions while executing the actual system code needed to implement the file system. As such, it accurately accounts for the overhead of system software. We configured Proteus as in [12], except as noted below.

Simulation overhead limited our experiments to decomposing a 1024×1024 matrix of single-precision numbers, using eight compute processors (CPs), eight I/O processors (IOPs), and eight disks (one on each IOP). This matrix only represented 4 MB of data, but when using the smallest slab size (16 columns per CP) the algorithm moved nearly 4 GB between disk and memory. Note that each column required 4 KB. Our file systems striped the file across all eight disks by 1 KB, 4 KB, or 8 KB blocks. The 4 KB blocks represent an "easy" case, where each full-column read and write operation touches precisely one block, and there are no shared blocks or partial-block requests. The 1 KB blocks represent a "likely" case, where each column requires several blocks. With 8 KB blocks a full-column transfer touches only half of a block, testing the ability of the cache to manage the subsequent spatial locality, and testing the effect of the extraneous disk reads needed when writing only half a block. Within each disk the blocks were laid out either randomly or contiguously, representing two interesting endpoints in the choice of block layouts.

We chose a slab size of 16, 32, or 128 columns per processor. With 8 CPs, these choices reflect total application memory sizes of 128, 256, or 1024 columns. In the last case, the matrix fit entirely in memory and so only one round of reading and writing was needed.

In the traditional-caching file system, the IOPs allocated two one-block buffers per compute processor per disk, or $2 \times 8 \times 8 = 128$ blocks of total cache, holding 32, 128, or 256 columns depending on the block size. While this cache may seem small, it is consistent with the size of the system and problem, and with our previous experiments [12]. In the disk-directed file system, the IOPs allocated two one-block buffers per disk (for double-buffering each disk), or 16 blocks of total buffer space. Note that disk-directed I/O's buffers used an asymptotic order-of-magnitude less memory than did traditional caching's cache.

5 Results

We concentrate on two primary metrics in our experiments: the amount of disk I/O (in bytes) and the total execution time (in seconds). Given our parameters, however, the values of these measures spanned several orders

```
// run simultaneously by all P processors
// file initially contains N × N matrix M in column major order
// SLAB_COLS is the number of columns per processor per slab
float M_local[N][SLAB_COLS];        // this processor's portion of a slab of M[N][N]
float multipliers[N];               // local copy of multipliers
int colsInMem = P * SLAB_COLS;      // number of columns in all P memories

barrier();  start clock;

for (i = 1 to N - colsInMem)  {
    my_first = the first column I will handle; // processor i mod P handles column i + 1
    ncols = the number of columns I will handle, usually SLAB_COLS;

    LU_read(M_local, my_first, ncols);      // get that slab from the file
    if (I am responsible for column i) {
        find the N-i multipliers;
        broadcast them to all other nodes;
    } else {
        receive the broadcasted multipliers;
    }
    update the ncols columns in M_local using multipliers;
    LU_write(M_local, my_first, ncols);      // and write the slab back

    // now update the rest of the columns
    leftmost = i;

    // everybody loop until everybody is done
    while ((leftmost += colsInMem) <= N) {
        my_first += colsInMem;
        ncols = the number of columns I will handle
                (usually SLAB_COLS, but could be fewer, or even 0);

        LU_read(M_local, my_first, ncols);   // get that slab from the file
        update the ncols columns in M_local using multipliers;
        LU_write(M_local, my_first, ncols);  // and write the slab back
    }
}

// ok, now do the colsInMem columns not handled above
my_first = the first column I will handle;
ncols = the number of columns I will handle (as few as 0);

LU_read(M_local, my_first, ncols);          // get that final slab from the file
for (i = i to N-1)  {
    if (I am responsible for column i) {
        find the N-i multipliers;
        broadcast them to all other nodes;
    } else {
        receive the broadcasted multipliers;
    }
    update the columns in M_local using multipliers;
}
LU_write(M_local, my_first, ncols);        // and write the slab back

sync();                                 // wait for all disk I/O to complete
barrier();  stop clock;
```

Figure 2: Pseudo-code for parallel, out-of-core LU-decomposition program.

of magnitude (e.g., with 128-column slabs the matrix fits in memory and the program causes 8 MB of disk traffic over about one minute, whereas with smaller slabs the program moves the matrix in and out of memory and causes 3–4 GB of traffic lasting for nearly an hour). Furthermore, insights come by comparing the performance of two configurations, rather than from the absolute performance of any one configuration. Thus, we normalize and compare by charting the ratio of a measure between one configuration and another (see [13] for the raw data).

Figure 3 displays the ratio of disk-directed I/O's performance to traditional caching's performance, for a variety of configurations using 4 KB blocks. Figure 3a focuses on the disk-I/O traffic. The amount of *file-system* traffic generated by the LU-decomposition program depended only on the slab size, so by using the ratio we normalize for the difference between slab sizes so that any visible differences are due to differences in the way the file systems use the disks. Note that both file systems caused about the same amount of disk I/O, with the traditional caching system occasionally making mistakes that caused a little extra I/O. Figure 3b shows the total execution time, and paints a different picture. Disk-directed I/O was never slower, and was faster when using the random-blocks layout due to its ability to optimize disk-head movement (the average disk-access time was 23–35% faster). With the exception of 128-column slabs, the improvement of disk-directed I/O over traditional caching increased with slab size, because the larger disk-directed requests permitted sorting over a larger set of data. With 128-column slabs the entire matrix fit in memory, the application was compute-bound, and thus the improvements had little effect on execution time.

In Figure 4 we examine the performance when the block size was changed from 4 KB to 8 KB. This change increases the disk and network transfer unit, changes the striping unit, and doubles the size of traditional caching's cache. The larger block size hardly affected disk-directed I/O's disk traffic, but (despite the larger caches) dramatically increased the amount of traffic for traditional caching in some cases. (The 16-column slabs were an exception, because each slab fit entirely into the cache, and the necessary blocks remained in the cache between the read and write phases of each iteration.) The additional traffic was caused by the 4 KB (column) writes to 8 KB blocks, which caused a disk read when the block was not resident in the cache. Disk-directed I/O, with its higher-level perspective, recognized that the blocks were to be fully written and avoided these "installation reads."

Figure 4b shows the performance impact of traditional caching's excessive installation reads. Disk-directed I/O was able to make efficient use of 8 KB blocks to obtain better performance, despite a comparable amount of disk traffic. Traditional caching had mixed results. With 128-column slabs, the I/O time was only a small part of execution time, so the performance impact was small. With 32-column slabs, the effect was amplified in the contiguous layout because the extra I/O caused many costly seeks (the average access time increased by 145%!), and was counteracted in the random layout by the reduction in seeks needed to reach half as many blocks.

Figure 5 compares disk-directed I/O and traditional caching on 8 KB blocks, using the same data as Figure 4 and in the same style as Figure 3. Here we see the clear dominance of disk-directed I/O in terms of execution time, despite the extraneous synchronization and (in some cases) extra disk I/O. Indeed, unless the entire matrix fit in memory (128-column slabs) or the slab size was limited to the cache size (16-column slabs), disk-directed I/O was 2–3 times faster than traditional caching. Even when causing more disk traffic, disk-directed I/O had lower overhead and better seek behavior (its pre-sorted disk schedule reduced seek distance in random layouts and ensured sequential access in contiguous layouts).

In larger, more realistic problem sizes, that is, with larger matrices, the column size would be much larger than the block size, rather than smaller. In Figure 6 we examine the situation when the block size was 1 KB, so that each column spans four blocks (spread over four disks). The amount of disk traffic was nearly unchanged, but the execution times were remarkably different. The compute-bound 128-column slab cases were barely affected, but all other cases were drastically slower. Much of this slowdown was due to the increased overhead of a smaller transfer unit. In the contiguous layout the traditional caching system caused *much* more disk-head movement because each CP was active in a slightly different region of the file, more than tripling the average disk-access time. Ultimately, as shown in Figure 7, disk-directed I/O was much faster than traditional caching in the difficult, but realistic cases where the the matrix did not fit in memory and the column size was larger than the block size.

Finally, traditional caching uses more memory on each IOP than does disk-directed I/O. Indeed, with a 4 KB block size traditional caching with slab size 16 uses nearly the same total amount of memory (128 columns in the CPs and 128 columns in the IOP caches) as does disk-directed I/O with slab size 32 (256 columns in the CPs and 16 columns in the IOP buffers). Comparing these two configurations, the DDIO/TC execution-time ratio is 90% for contiguous layouts and 70% for random layouts. Part of this improvement is because the application could make better use of the memory to reduce I/O demands (many I/O algorithms do asymptotically less I/O given more memory), and part is because the larger request sizes enabled disk-directed I/O to better optimize the I/O.

In summary, disk-directed I/O often improved the per-

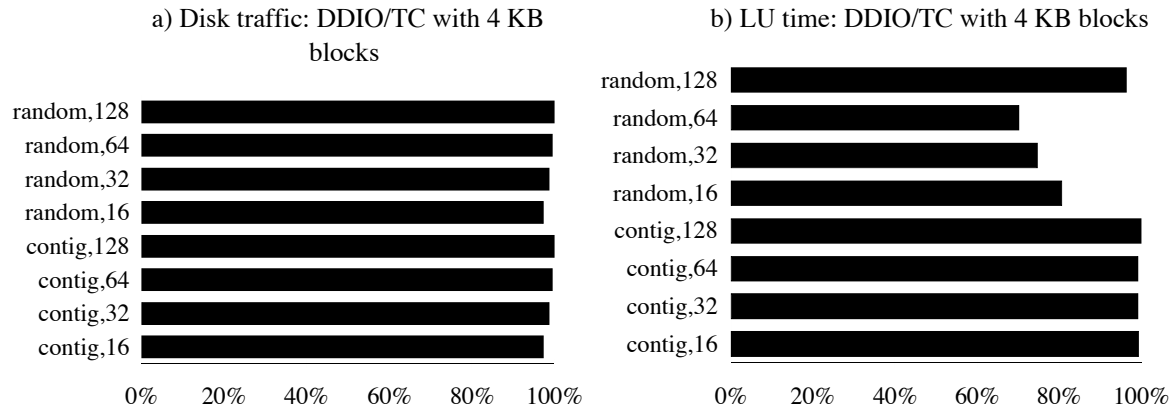

Figure 3: The ratio of disk-directed I/O (DDIO) to traditional caching (TC), in terms of bytes of disk traffic and seconds of execution time. The ratio is expressed as a percentage. Thus, less than 100% indicates that DDIO was better, i.e., did less I/O or took less time. All used a 4 KB block size.

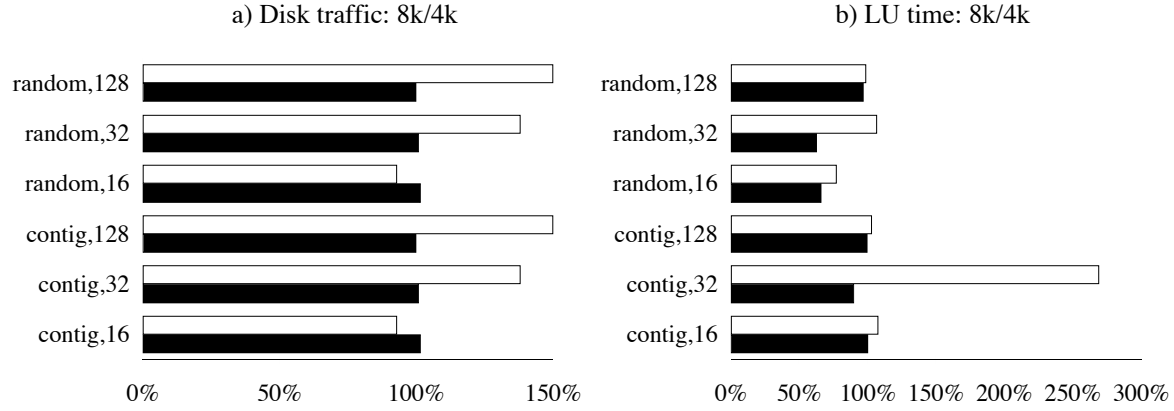

Figure 4: The ratio of LU-decomposition performance with 8 KB blocks to that with 4 KB blocks, in terms of bytes of disk traffic and seconds of execution time. Less than 100% indicates that 8 KB was better. For each case there are two bars, one for traditional caching (white) and one for disk-directed I/O (black).

formance of the LU-decomposition program. In a random layout, it was able to optimize the order of disk access within each disk-directed request. This benefit should be even larger in larger problem sizes with larger slab sizes. It also used less memory — memory that the application could use to reduce I/O demands. Furthermore, it avoided the extraneous installation reads, unnecessary prefetches, and occasional cache mistakes caused by traditional caching. Finally, although disk-directed I/O never made performance worse, despite the extraneous synchronization, it had little benefit for 4 KB blocks on contiguous layouts. There, traditional caching was able to maintain the same performance as disk-directed I/O largely because the I/O-request size (1 column) was the same as the caching unit (1 block). In a larger problem, the request size would be larger, and either the caching unit (block size) must also be larger or each request must span many blocks. The former would require a very large cache, and the latter would have the effect of spreading out simultaneous multi-block requests into multiple localities, counteracting the benefits of the contiguous layout [12]. The results of experiments with 1 KB blocks support this statement. Overall, the disk-directed file system would be the faster choice.

6 Conclusions

Until recently most multiprocessor file systems have provided the programmer with a familiar Unix-like interface, consisting of read, write, and seek calls, and various "modes" to control the semantics of a shared file pointer. While this interface is comfortable to parallel programmers familiar with sequential programming, it is inadequate for expressing their needs [14]. Given this interface and the amount of interprocessor spatial locality arising from interleaving tiny requests from many processors, caching is

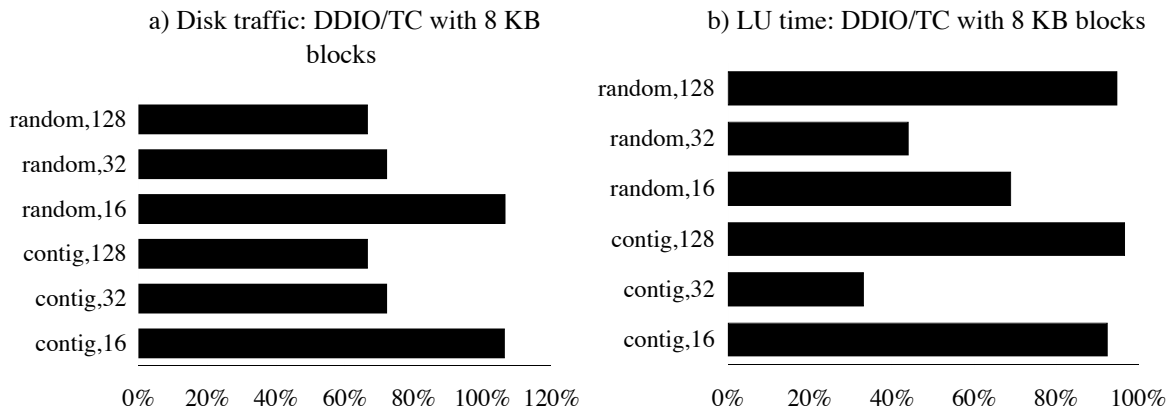

Figure 5: Just like Figure 3, but using an 8 KB block size.

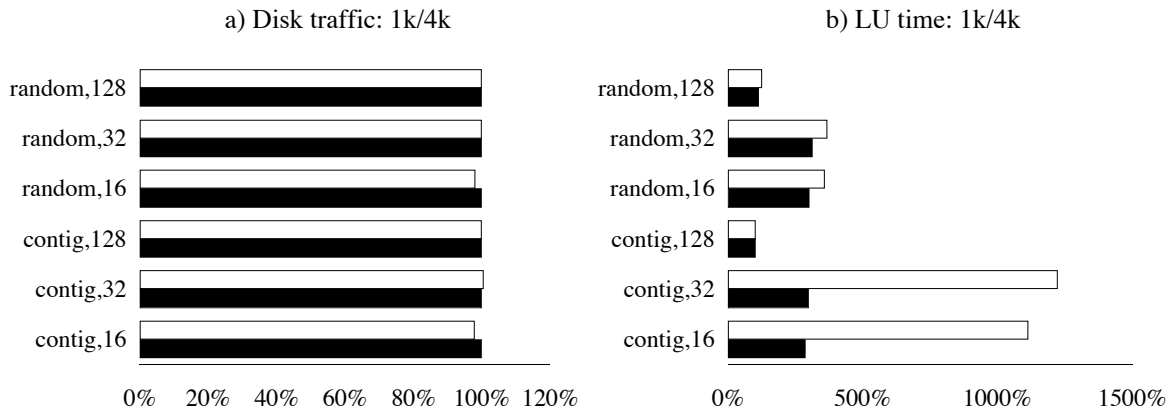

Figure 6: The ratio of LU-decomposition performance with 1 KB blocks to that with 4 KB blocks, in terms of bytes of disk traffic and seconds of execution time. Less than 100% indicates that 1 KB was better. For each case there are two bars, one for traditional caching (white) and one for disk-directed I/O (black).

essential for reasonable performance [14]. A file system based on traditional caching, however, can have terrible performance [18] and, as we show in this paper, can have counter-intuitive performance characteristics (increasing the block size from 4 KB to 8 KB, or increasing the slab size from 16 to 32 columns, sometimes *decreased* performance).

As we show here and in [12], disk-directed I/O can lead to much better performance than traditional caching. This paper shows that disk-directed I/O, using a collective, high-level interface, could be used effectively for an out-of-core LU-decomposition computation. The additional synchronization of the collective interface appeared not to be a significant factor here.

Of course, some types of applications may not benefit from disk-directed I/O. In particular, those where small amounts of I/O necessarily alternate with computation, making large, high-level requests impossible, would be better served by a cache-based system. Furthermore, some irregular problems may make collective I/O extremely difficult or inefficient, reducing the usefulness of disk-directed I/O.

In our LU-decomposition example the code needed some careful structuring to ensure that all processes participated in all I/O requests. Clearly, a collective interface that supported *subsets* of processes would reduce the need to structure the code this way (the MPI-IO proposal [4] appears to have this support). Otherwise, any of the common collective matrix-I/O interfaces could be adapted for use.

The next challenge is to define a specific interface and to experiment with real applications, such as the computational fluid dynamics we encountered in our tracing efforts [14].

References

[1] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proc. of the 7th IPPS*, pp. 489–495, 1993.

[2] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and evaluation of primitives for parallel I/O. In *Proc. of Supercomp. '93*, pp. 452–461, 1993.

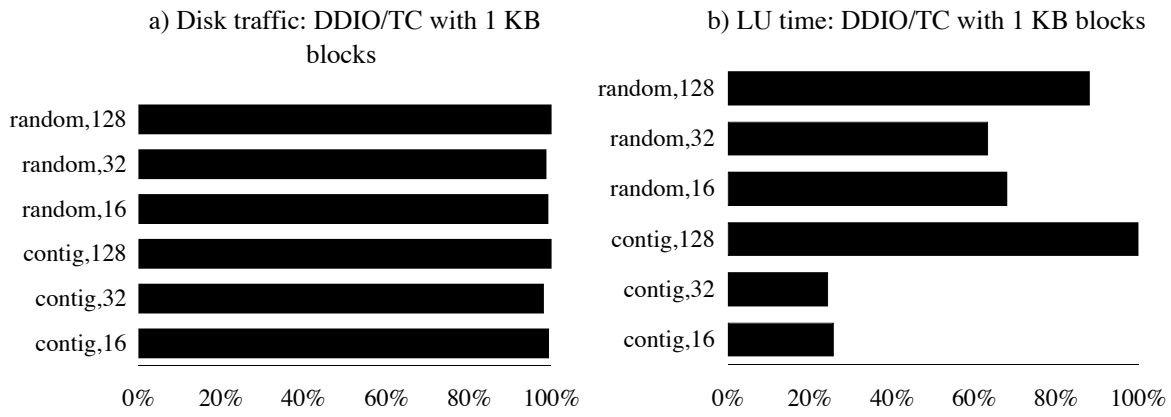

Figure 7: Just like Figure 3, but using a 1 KB block size.

[3] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. MIT TR LCS/TR–516, 1991.

[4] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: a parallel file I/O interface for MPI. Tech. Rep. NAS-95-002, NASA Ames Research Center, 1995. v. 0.3.

[5] P. F. Corbett and D. G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proc. of the Scalable High-Perf. Comp. Conf.*, pp. 63–70, 1994.

[6] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Dartmouth College TR PCS-TR94-243.

[7] E. DeBenedictis and J. M. del Rosario. nCUBE parallel I/O software. In *Proc. of the 11th IPCCC*, pp. 0117–0124, 1992.

[8] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on I/O in Par. Comp. Sys.*, pp. 56–70.

[9] D. G. Feitelson, P. F. Corbett, Y. Hsu, and J.-P. Prost. Parallel I/O systems and interfaces for parallel computers. In C.-L. Wu, editor, *Multiprocessor Systems — Design and Integration*. World Scientific, 1995. To appear.

[10] M. Harry, J. M. del Rosario, and A. Choudhary. VIP-FS: A VIrtual, Parallel File System for high performance parallel and distributed computing. In *Proc. of the 9th IPPS*, pp. 159–164, Apr. 1995.

[11] J. F. Karpovich, A. S. Grimshaw, and J. C. French. Extensible file systems ELFS: An object-oriented approach to high performance file I/O. In *Proc. of the 9th OOPSLA*, pp. 191–204, Oct. 1994.

[12] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. of the 1994 Symp. on OS Design and Impl.*, pp. 61–74, Nov. 1994. Updated as Dartmouth TR PCS-TR94-226 on Nov. 8, 1994.

[13] D. Kotz. Disk-directed I/O for an out-of-core computation. Technical Report PCS-TR95-251, Dept. of Computer Science, Dartmouth College, Jan. 1995.

[14] D. Kotz and N. Nieuwejaar. File-system workload on a scientific multiprocessor. *IEEE Par. and Dist. Tech.*, pp. 51–60, Spring 1995.

[15] O. Krieger. *HFS: A flexible file system for shared-memory multiprocessors*. PhD thesis, University of Toronto, Oct. 1994.

[16] S. A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proc. of the Scalable High-Perf. Comp. Conf.*, pp. 71–78, 1994.

[17] N. Nieuwejaar and D. Kotz. Low-level interfaces for high-level parallel I/O. In *IPPS '95 Workshop on I/O in Par. and Dist. Sys.*, pp. 47–62, 1995.

[18] B. Nitzberg. Performance of the iPSC/860 Concurrent File System. Tech. Rep. RND-92-020, NAS Systems Division, NASA Ames, 1992.

[19] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proc. of the Fourth Conf. on Hypercube Concurrent Comp. and Appl.*, pp. 155–160. 1989.

[20] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proc. of the 9th IPPS*, pp. 165–172, 1995.

[21] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. Submitted to Supercomputing '95, 1995.

[22] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proc. of the 8th ACM Int'l Conf. on Supercomp.*, pp. 382–391, 1994.

[23] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proc. of the Scalable Par. Libraries Conf.*, pp. 119–128, 1994.

[24] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proc. of the 1993 DAGS/PC Symposium*, pp. 56–63, Hanover, NH, 1993.

Many of the above references are available via URL `http://www.cs.dartmouth.edu/pario.html`