# 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

Technical Report PCS-TR95-251

January 13, 1995

**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*), allowing the programmer to express a complex transfer in a single (perhaps collective) request. Collective, high-level requests permit techniques like *two-phase I/O* and *disk-directed I/O* to significantly improve performance over traditional file systems and interfaces. Neither of these techniques have been tested on anything other than simple benchmarks that read or write matrices. Many applications, however, intersperse computation and I/O to work with data sets that cannot fit in main memory. In this paper, we present the results of experiments with an "out-of-core" LU-decomposition program, comparing a traditional interface and file system with a system that has a high-level, collective interface and disk-directed I/O. We found that a collective interface was awkward in some places, and forced additional synchronization. Nonetheless, disk-directed I/O was able to obtain much better performance than the traditional system.

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.

# 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 [SGM86] 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 [FPD93, Nit92].

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 [Kot94] to improve performance by orders of magnitude. In [Kot94], 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 the context of a full program, one 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 detailed 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 Bridge [DSE88], Intel CFS [Pie89], Intel PFS [Roy93], IBM Vesta [CF94], nCUBE [DdR92], TMC sfs [LIN+93, BGST93], Hurricane File System [Kri94], and SPIFFI [FBD95]. There are also several systems intended for workload clusters, such as PIOUS [MS94] and VIP-FS [dHC94]. All of these systems decluster file data across many disks to provide parallel access to the data of any file.

**Workload.** The CHARISMA project traced production parallel scientific computing workloads on an Intel iPSC/860 [KN94] and on a TMC CM-5 [PEK+94] 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 [NK94], 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 CHARISMA project. Higher-level interfaces, such as specifying a strided series of requests [NK94, Cra94] or accessing data through a mapping function [CF94, DdR92, Kot93] 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, operating system, and hardware. The MPI message-passing interface may soon be extended to include I/O [CFH+94], including collective I/O. Finally, there are several libraries for collective matrix I/O [GGL93, BdC93, BBS+94, SW94].

**Two-phase I/O.** Two-phase I/O is a technique for optimizing data transfer given a high-level, collective interface [dBC93]. 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 two-phase writing, or 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 [Kot94]. 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 [Kot94].

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

One simple parallelization of this algorithm (although not the best; see [WGWR93] 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.

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 corresponding sets of slabs for all processors collectively represent a contiguous set of bytes in the file.

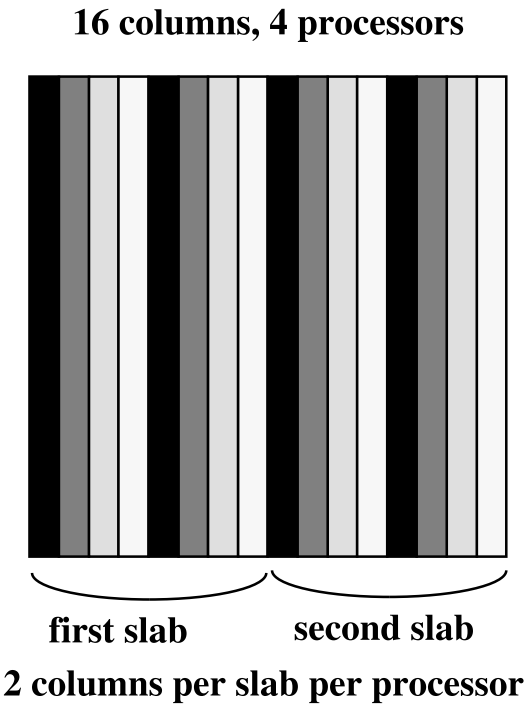


Figure 1: Example of column-cyclic distribution of 16 columns across four processors. Each processor is represented here by a different shade of gray. SLAB_COLS is 2 here, meaning each processor allocates space for two columns in main memory. The combined slab size is eight columns.

The code for parallel, out-of-core LU-decomposition (based on that in [TBC94]) 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` looks like that in Figure 3a. `LU_write` would look similar. Given a collective interface, these functions would be replaced as shown in Figure 3b. Note that no synchronization is necessary with the traditional

```
//  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
    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.

interface, whereas the collective interface must synchronize all processors for disk-directed I/O. This "extraneous" synchronization would in general accentuate temporary load imbalances, but it can often allow dramatically better I/O performance.

*a) the traditional read/write/seek interface:*

```
LU_read(array, first_col, ncols)
{
    int col_bytes = N * sizeof(float); //  size of a column
    int col = first_col;

    //  loop through the desired number of columns
    for (i = 1 to ncols) {
        seek((col-1) * col_bytes);    //  find this column in the file
        read(array, col_bytes);       //  read one column

        array += col_nbytes;  //  skip to next column
        col += P;             //  column cyclic
    }
}
```

*b) a collective interface with disk-directed I/O support:*

```
LU_read(array, first_col, ncols)
{
    barrier();

    first_col = min(first_col) over all processors;
    ncols = sum(ncols) over all processors;

    disk-directed read of (first_col) through (first_col + ncols - 1);

    barrier();
}
```

Figure 3: Pseudo-code for `LU_read` (`LU_write` is similar).

Finally, we note that code like that in Figure 2 could be written by hand, incorporated in a parallel matrix library [GGL93, BdC93, BBS+94, SW94], or generated by a smart compiler [CC94, TBC94, BTC94].

## 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 (Figure 3a) and the disk-directed file system (Figure 3b), on top of our parallel file-system simulator [Kot94]. This simulator ran on top of the

Proteus parallel-architecture simulator [BDCW91], which in turn ran on a DEC-5000 workstation. We configured Proteus as in [Kot94], except as noted below.

Simulation overhead limited our experiments to decomposing a $1024 \times 1024$ 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 [Kot94]. 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 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

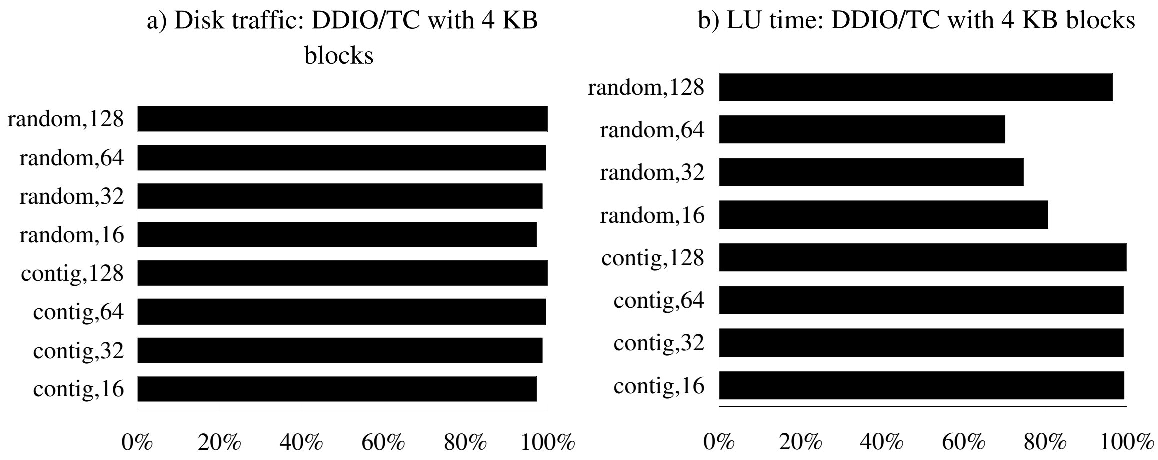


Figure 4: The ratio of disk-directed I/O (DDIO) to traditional caching (TC), in terms of bytes of disk traffic and seconds of execution time. The ratio is expressed as a percentage. Thus, less than 100% indicates that DDIO was better, i.e., did less I/O or took less time. There are several cases, using either contiguous or random layouts, and 16-, 32-, 64-, or 128-column slab size. All used a 4 KB block size.

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.[1]

Figure 4 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 4a focuses on the disk-I/O traffic. Note that the amount of file-system traffic generated by the LU-decomposition program depended only on the slab size, and 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 4b 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. 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 5 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

[1]See the Appendix for the raw data.

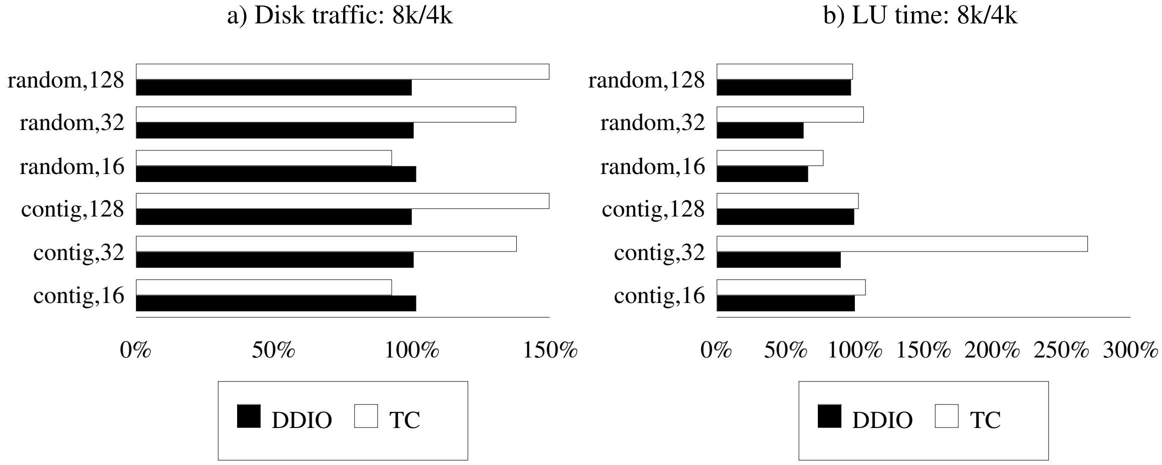


Figure 5: The ratio of LU-decomposition performance with 8 KB blocks to that with 4 KB blocks, in terms of bytes of disk traffic and seconds of execution time. The ratio is expressed as a percentage. Thus, less than 100% indicates that 8 KB was better, i.e., did less I/O or took less time. There are several cases, using either contiguous or random layouts, and 16-, 32-, or 128-column slab size. For each case there are two bars, one for traditional caching (TC) and one for disk-directed I/O (DDIO).

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 5b 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, and was counteracted in the random layout by the reduction in seeks needed to reach half as many blocks.

Figure 6 compares disk-directed I/O and traditional caching on 8 KB blocks, using the same data as Figure 5 and in the same style as Figure 4. 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

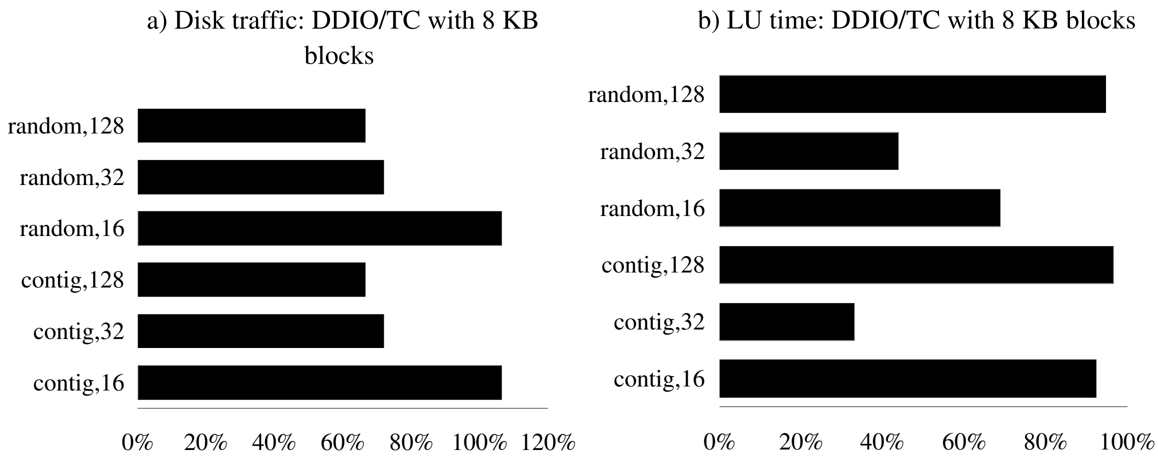


Figure 6: Just like Figure 4, but using an 8 KB block size.

limited to the cache size (16-column slabs), disk-directed I/O was 2–3 times faster than traditional caching.

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 7 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 the each CP was active in a slightly different region of the file. Ultimately, as shown in Figure 8, 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.

Figure 9 compares the traffic and execution speed of 32-column slabs with 16-column slabs. The LU-decomposition program causes less file-system traffic with 32-column slabs, as is reflected in Figure 9a. The chart again shows the cost of installation reads in traditional caching with 8 KB blocks. The contiguous layout accentuates the cost of the extraneous I/O, because the additional seeks remove many of the benefits of contiguous layout (Figure 9b).

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

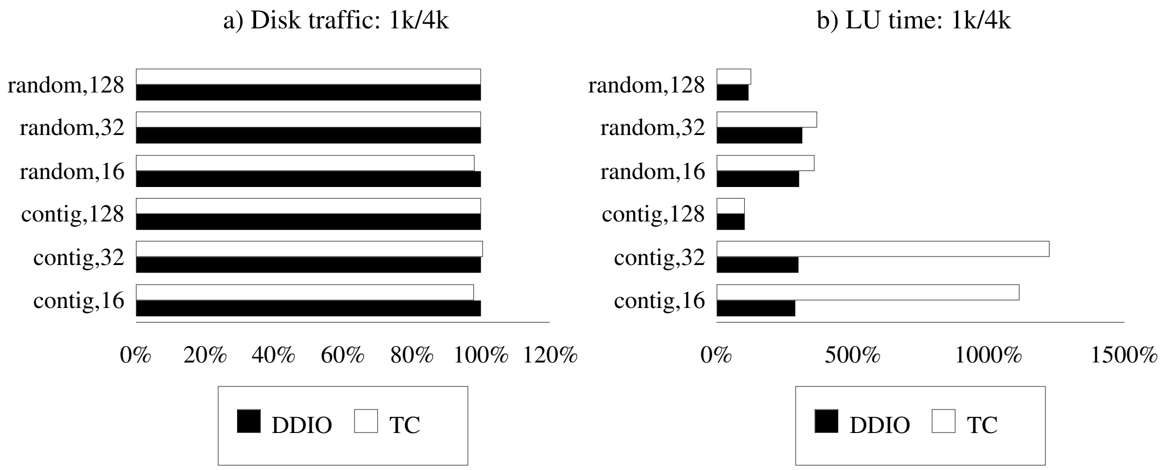


Figure 7: The ratio of LU-decomposition performance with 1 KB blocks to that with 4 KB blocks, in terms of bytes of disk traffic and seconds of execution time. The ratio is expressed as a percentage. Thus, less than 100% indicates that 1 KB was better, i.e., did less I/O or took less time. There are several cases, using either contiguous or random layouts, and 16-, 32-, or 128-column slab size. For each case there are two bars, one for traditional caching (TC) and one for disk-directed I/O (DDIO).

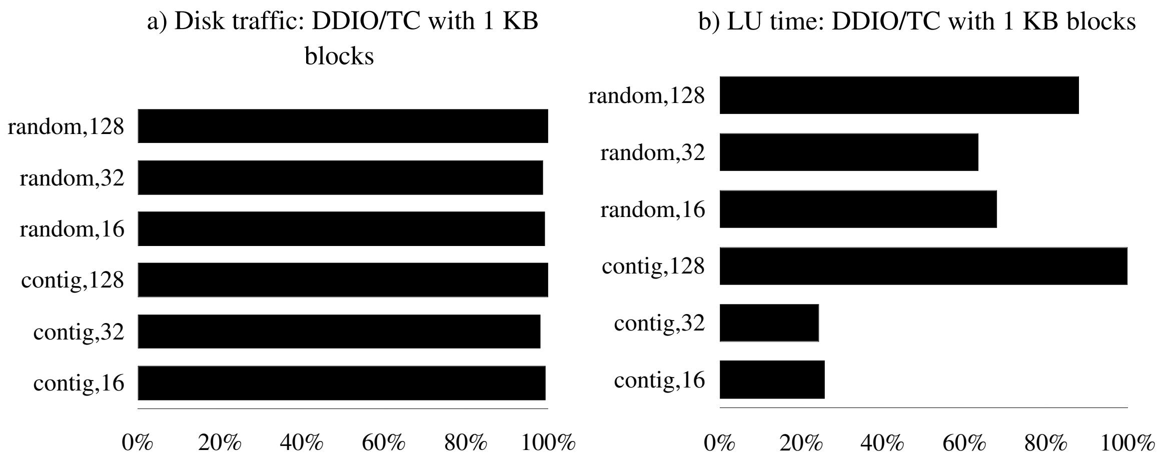


Figure 8: Just like Figure 4, but using a 1 KB block size.

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 [CK93]), and part is because the larger request sizes enable disk-directed I/O to better optimize the I/O.

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

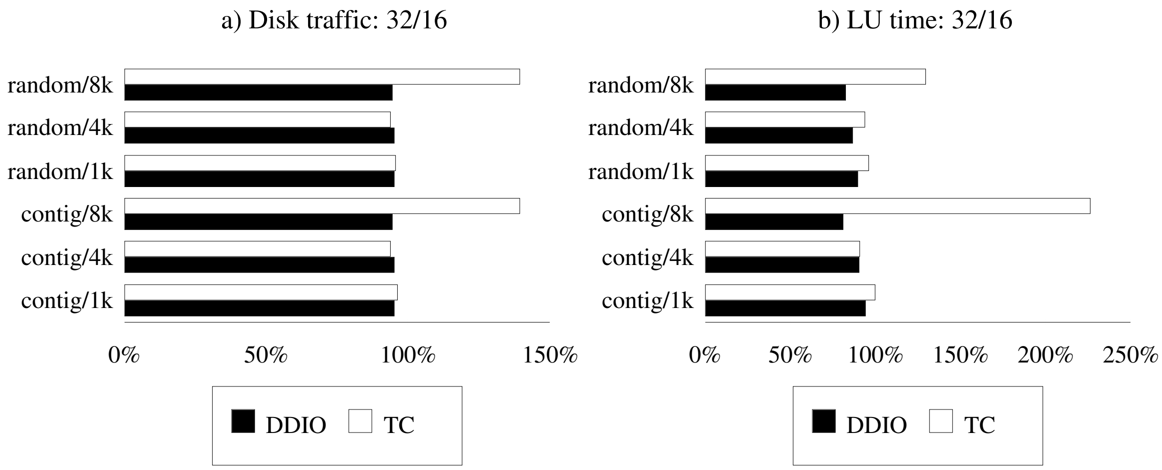


Figure 9: The ratio of 32-column slabs to 16-column slabs, in terms of bytes of disk traffic and seconds of execution time. Thus, less than 100% indicates that 32-column slabs were better, i.e., did less I/O or took less time. There are several cases, using either contiguous or random layouts, disk-directed I/O (DDIO) or traditional caching (TC), and 1 KB, 4 KB, or 8 KB block size.

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 [Kot94]. 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 [KN94]. Given this interface and the amount of interprocessor spatial locality arising from interleaving tiny requests from many processors, caching is essential for reasonable performance [KN94]. A file system based on traditional caching, however, can have terrible performance [Nit92] 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 [Kot94], 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.

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 [CFH+94] appears to have this support). Otherwise, any of the common collective matrix-I/O interfaces could be adapted for use. Ultimately, more cases need to be studied to determine an appropriate general-purpose interface.

Thus disk-directed I/O was successful for out-of-core computations, despite the additional synchronization of a collective interface. The next challenge is to define a specific interface and to experiment with real applications.

### Acknowledgements

Many thanks to Rajesh Bordawekar at Syracuse University for giving me his LU-decomposition program (in Fortran for the Intel Touchstone Delta) as a starting point. And thanks to the University of Virginia, where I was on sabbatical while doing this research.

### Availability

Many of the references below are available via the WWW at URL

`http://www.cs.dartmouth.edu/pario.html`

The simulator source code will be available there in early 1995.

## References

[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 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[BdC93] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR–516, MIT, September 1991.

[BGST93] Michael L. Best, Adam Greenberg, Craig Stanfill, and Lewis W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[BTC94] Rajesh Bordawekar, Rajeev Thakur, and Alok Choudhary. Efficient compilation of out-of-core data parallel programs. Technical Report SCCS-622, NPAC, April 1994.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[CF94] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[CFH+94] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report RC 19841 (87784), IBM T.J. Watson Research Center, November 1994. Version 0.2.

[CK93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[Cra94] Cray Research. listio *manual page*, 1994. Publication SR-2012.

[dBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[DdR92] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, April 1992.

[dHC94] Juan Miguel del Rosario, Michael Harry, and Alok Choudhary. The design of VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, NPAC, Syracuse, NY 13244, May 1994.

[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.

[FBD95] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. Submitted to IEEE TPDS.

[FPD93] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1–2):115–121, January and February 1993.

[GGL93] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

[Kot93] David Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kri94] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 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.

[Nit92] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[NK94] Nils Nieuwejaar and David Kotz. A multiprocessor extension to the conventional file system interface. Technical Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, September 1994.

[PEK+94] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. Technical Report CS-1994-33, Dept. of Computer Science, Duke University, October 1994. To appear in IPPS '95.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160.

[Roy93] Paul J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

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

[SW94] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[TBC94] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, July 1994.

[WGWR93] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

## Appendix: Raw data

Figure 10 shows the raw data from which the charts are derived. The columns are as follows, left to right:

**Layout:** contiguous or random

**Slab:** the number of columns in each processor's slab

**Block:** file-system block size (and striping unit) in bytes

**FS:** DD (disk-directed I/O) or TC (traditional caching)

**App read:** total read requests to the file system from the application, in megabytes

**App wrote:** total write requests to the file system from the application, in megabytes

**App total:** the sum of App read and App wrote

**Disk read:** total read requests to the disks from the file system, in megabytes

**Disk wrote:** total write requests to the disks from the file system, in megabytes

**Disk total:** the sum of Disk read and Disk wrote

**inflation:** the ratio of Disk total to App total, expressed as a percentage

**DD/TC:** the ratio of Disk total for that DD case to the corresponding TC case, expressed as a percentage

**8k/4k:** the ratio of Disk total for that 8k case to the corresponding 4k case, expressed as a percentage

**1k/4k:** the ratio of Disk total for that 1k case to the corresponding 4k case, expressed as a percentage

**slab ratio:** the ratio of Disk total for that case to the corresponding case with half the slab size (i.e., 32/16 or 64/32), expressed as a percentage

**Seconds:** total LU-decomposition time in seconds

**DD/TC:** the ratio of Seconds for that DD case to the corresponding TC case, expressed as a percentage

**8k/4k:** the ratio of Seconds for that 8k case to the corresponding 4k case, expressed as a percentage

**1k/4k:** the ratio of Seconds for that 1k case to the corresponding 4k case, expressed as a percentage

**slab ratio:** the ratio of Seconds for that case to the corresponding case with half the slab size (i.e., 32/16 or 64/32), expressed as a percentage

| Layout | Slab | Block | FS | App read | App wrote | App total | Disk read | Disk wrote | Disk total | inflation | DD/TC | 8k/4k | 1k/4k | slab ratio | **Seconds** | DD/TC | 8k/4k | 1k/4k | slab ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| contig | 16 | 1024 | TC | 2018.3 | 2018.3 | 4036.5 | 2043.6 | 2018.3 | 4061.9 | 100.6% | | | | | 5189.8 | | | | |
| contig | 16 | 1024 | DD | 2018.3 | 2018.3 | 4036.5 | 2018.3 | 2018.3 | 4036.5 | 100.0% | 99.4% | | | | 1339.4 | 25.8% | | | |
| contig | 16 | 4096 | TC | 2018.3 | 2018.3 | 4036.5 | 2127.2 | 2018.3 | 4145.5 | 102.7% | | | 98.0% | | 466.0 | | | 1113.8% | |
| contig | 16 | 4096 | DD | 2018.3 | 2018.3 | 4036.5 | 2018.3 | 2018.3 | 4036.5 | 100.0% | 97.4% | | 100.0% | | 462.9 | 99.3% | | 289.3% | |
| contig | 16 | 8192 | TC | 2018.3 | 2018.3 | 4036.5 | 1916.0 | 1933.3 | 3849.3 | 95.4% | | 92.9% | | | 502.6 | | 107.9% | | |
| contig | 16 | 8192 | DD | 2018.3 | 2018.3 | 4036.5 | 2065.5 | 2034.0 | 4099.5 | 101.6% | 106.5% | 101.6% | | | 464.5 | 92.4% | 100.4% | | |
| | | | | | | | | | | | | | | | | | | | |
| contig | 32 | 1024 | TC | 1922.5 | 1922.5 | 3845.0 | 1992.6 | 1922.5 | 3915.1 | 101.8% | | | | 96.4% | 5181.0 | | | | 99.8% |
| contig | 32 | 1024 | DD | 1922.5 | 1922.5 | 3845.0 | 1922.5 | 1922.5 | 3845.0 | 100.0% | 98.2% | | | 95.3% | 1264.4 | 24.4% | | | 94.4% |
| contig | 32 | 4096 | TC | 1922.5 | 1922.5 | 3845.0 | 1970.2 | 1922.5 | 3892.7 | 101.2% | | | 100.6% | 93.9% | 423.3 | | | 1224.1% | 90.8% |
| contig | 32 | 4096 | DD | 1922.5 | 1922.5 | 3845.0 | 1922.5 | 1922.5 | 3845.0 | 100.0% | 98.8% | | 100.0% | 95.3% | 419.7 | 99.2% | | 301.2% | 90.7% |
| contig | 32 | 8192 | TC | 1922.5 | 1922.5 | 3845.0 | 3442.8 | 1927.7 | 5370.5 | 139.7% | | 138.0% | | 139.5% | 1139.0 | | 269.1% | | 226.6% |
| contig | 32 | 8192 | DD | 1922.5 | 1922.5 | 3845.0 | 1945.0 | 1930.0 | 3875.0 | 100.8% | 72.2% | 100.8% | | 94.5% | 377.8 | 33.2% | 90.0% | | 81.3% |
| | | | | | | | | | | | | | | | | | | | |
| contig | 64 | 4096 | TC | 1539.0 | 1539.0 | 3078.0 | 1554.9 | 1539.0 | 3093.9 | 100.5% | | | | 79.5% | 339.0 | | | | 80.1% |
| contig | 64 | 4096 | DD | 1539.0 | 1539.0 | 3078.0 | 1539.0 | 1539.0 | 3078.0 | 100.0% | 99.5% | | | 80.1% | 336.4 | 99.2% | | | 80.2% |
| | | | | | | | | | | | | | | | | | | | |
| contig | 128 | 1024 | TC | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | | | | | 57.1 | | | | |
| contig | 128 | 1024 | DD | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | 100.0% | | | | 57.1 | 100.0% | | | |
| contig | 128 | 4096 | TC | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | | | 100.0% | 0.3% | 55.4 | | | 103.2% | 16.3% |
| contig | 128 | 4096 | DD | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | 100.0% | | 100.0% | 0.3% | 55.4 | 100.0% | | 103.2% | 16.5% |
| contig | 128 | 8192 | TC | 4.0 | 4.0 | 8.0 | 8.0 | 4.0 | 12.0 | 150.0% | | 150.0% | | | 57.1 | | 103.1% | | |
| contig | 128 | 8192 | DD | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | 66.7% | 100.0% | | | 55.1 | 96.6% | 99.6% | | |
| | | | | | | | | | | | | | | | | | | | |
| random | 16 | 1024 | TC | 2018.3 | 2018.3 | 4036.5 | 2049.8 | 2018.3 | 4068.1 | 100.8% | | | | | 11722.0 | | | | |
| random | 16 | 1024 | DD | 2018.3 | 2018.3 | 4036.5 | 2018.3 | 2018.3 | 4036.5 | 100.0% | 99.2% | | | | 7973.8 | 68.0% | | | |
| random | 16 | 4096 | TC | 2018.3 | 2018.3 | 4036.5 | 2127.2 | 2018.3 | 4145.5 | 102.7% | | | 98.1% | | 3254.3 | | | 360.2% | |
| random | 16 | 4096 | DD | 2018.3 | 2018.3 | 4036.5 | 2018.3 | 2018.3 | 4036.5 | 100.0% | 97.4% | | 100.0% | | 2624.6 | 80.7% | | 303.8% | |
| random | 16 | 8192 | TC | 2018.3 | 2018.3 | 4036.5 | 1916.0 | 1931.4 | 3847.3 | 95.3% | | 92.8% | | | 2513.9 | | 77.2% | | |
| random | 16 | 8192 | DD | 2018.3 | 2018.3 | 4036.5 | 2065.5 | 2034.0 | 4099.5 | 101.6% | 106.6% | 101.6% | | | 1732.2 | 68.9% | 66.0% | | |
| | | | | | | | | | | | | | | | | | | | |
| random | 32 | 1024 | TC | 1922.5 | 1922.5 | 3845.0 | 1970.3 | 1922.5 | 3892.8 | 101.2% | | | | 95.7% | 11290.6 | | | | 96.3% |
| random | 32 | 1024 | DD | 1922.5 | 1922.5 | 3845.0 | 1922.5 | 1922.5 | 3845.0 | 100.0% | 98.8% | | | 95.3% | 7159.6 | 63.4% | | | 89.8% |
| random | 32 | 4096 | TC | 1922.5 | 1922.5 | 3845.0 | 1970.2 | 1922.5 | 3892.7 | 101.2% | | | 100.0% | 93.9% | 3054.9 | | | 369.6% | 93.9% |
| random | 32 | 4096 | DD | 1922.5 | 1922.5 | 3845.0 | 1922.5 | 1922.5 | 3845.0 | 100.0% | 98.8% | | 100.0% | 95.3% | 2280.6 | 74.7% | | 313.9% | 86.9% |
| random | 32 | 8192 | TC | 1922.5 | 1922.5 | 3845.0 | 3442.8 | 1927.1 | 5369.9 | 139.7% | | 137.9% | | 139.6% | 3262.7 | | 106.8% | | 129.8% |
| random | 32 | 8192 | DD | 1922.5 | 1922.5 | 3845.0 | 1945.0 | 1930.0 | 3875.0 | 100.8% | 72.2% | 100.8% | | 94.5% | 1434.1 | 44.0% | 62.9% | | 82.8% |
| | | | | | | | | | | | | | | | | | | | |
| random | 64 | 4096 | TC | 1539.0 | 1539.0 | 3078.0 | 1554.9 | 1539.0 | 3093.9 | 100.5% | | | | 79.5% | 2451.9 | | | | 80.3% |
| random | 64 | 4096 | DD | 1539.0 | 1539.0 | 3078.0 | 1539.0 | 1539.0 | 3078.0 | 100.0% | 99.5% | | | 80.1% | 1719.4 | 70.1% | | | 75.4% |
| | | | | | | | | | | | | | | | | | | | |
| random | 128 | 1024 | TC | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | | | | | 77.4 | | | | |
| random | 128 | 1024 | DD | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | 100.0% | | | | 68.2 | 88.0% | | | |
| random | 128 | 4096 | TC | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | | | 100.0% | 0.3% | 60.9 | | | 127.2% | 2.5% |
| random | 128 | 4096 | DD | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | 100.0% | | 100.0% | 0.3% | 58.7 | 96.4% | | 116.1% | 3.4% |
| random | 128 | 8192 | TC | 4.0 | 4.0 | 8.0 | 8.0 | 4.0 | 12.0 | 150.0% | | 150.0% | | | 60.2 | | 98.9% | | |
| random | 128 | 8192 | DD | 4.0 | 4.0 | 8.0 | 4.0 | 4.0 | 8.0 | 100.0% | 66.7% | 100.0% | | | 57.0 | 94.8% | 97.2% | | |

Figure 10: Raw data for LU-decomposition experiments. See the text for an explanation of columns.