Solution of Out-of-Core Lower-Upper Decomposition for Complex Valued Matrices

Marianne Spurrier *and* **Joe Swartz**, *Lockheed Martin Corp.*
and **Bruce Black**, *Cray Inc.*

ABSTRACT: *Matrix decomposition and solution software is readily available for problems that fit into local memory. These software packages are highly optimized for the memory available, the caches, and CPUs involved. However, most real world problems, in particular the problems that are typically solved utilizing high performance computers, do not fit into local memory. Lockheed Martin, under government contract, is developing matrix decomposition and solution software that is usable for all sizes of matrix problems. The only mathematical assumption being made in the overall algorithm is that the matrix is non-singular. This paper explores algorithms based on two methods for storing the matrix on the disk: slabs of rows or columns (slab algorithm) and sub-matrix blocks (block algorithm.) The algorithm dynamically calculates slab or block sizes to maximize the amount of matrix in local memory to maximize computational time on the problem and minimize disk I/O time. The algorithms also utilizes asynchronous disk I/O and utilizes the higher inter-processor bandwidth by "cascading" slabs or blocks from the CPU currently working to the next CPU(s) that will use them for the next step. This approach eliminates a large amount of disk I/O that increases the average I/O rates that are seen in the programs. The computational cores of both algorithms are BLAS routines. The code development is being facilitated by the MOS Cray X1 located at the Army High Performance Computing Research Centre in Minneapolis, MN. The results are two software packages that correctly decompose the matrix into lower and upper triangular forms (stored in the same file) and solve the decompositions against sets of solution vectors.*

1. Introduction

The solution of the system of linear equations

$$Ax = b \qquad (1)$$

can be found by inverting the matrix *A* and multiplying it by *b*. Another approach that is less computationally expensive for a dense matrix is the lower-upper decomposition (LUD). The LUD solves equation (1) by decomposing the matrix *A* into a lower triangular matrix, *L*, and an upper triangular matrix, *U*, such that

$$Ax = LUx = b. \qquad (2)$$

Substituting $y = Ux$ yields the triangular system

$$Ly = b. \qquad (3)$$

Equation (3) can easily be solved for *y* and then the other triangular system, $Ux=y$, can be solved for *x*.

The decomposition of *A* into *L* and *U* poses an interesting I/O problem when the matrix becomes too large to fit in memory. Two out-of-core LUD storage algorithms have been developed for use on the X1. The first algorithm stores the matrix on disk in contiguous slabs and the second stores the matrix as contiguous blocks as shown below.

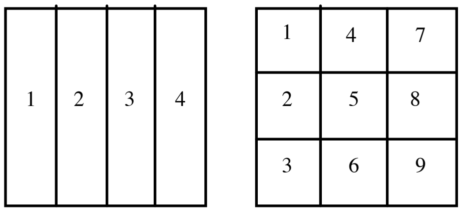

Figure 1

The key to an efficient algorithm is to use asynchronous I/O (AIO) to overlap the I/O with the computations, thus minimizing the amount of time lost to waiting for I/O to complete.

2. Slab Algorithm

The slab LUD and solve algorithms, both I/O and computational, are based on the work done by the benchmark group in the Cray Research, Inc., first completed around 1988 to support a government customer. The computational sections of the slab LUD and solve utilize the

BLAS matrix computations library. Subsequent versions of the slab algorithms were maintained by CRI until about 1998 when the Lockheed Martin Program, CSCF, started maintaining the code for internal use.

The code utilizes a triple buffer system with asynchronous disk I/O to overlap the disk I/O with computations. This algorithm approach was I/O bound until the C90 and SV1 systems were produced. The current version of this algorithm running on the SV1, utilizing 16 CPUs and 512 megawords of memory, for most sized complex valued matrices (up to approximately 120,000 unknowns) is almost perfectly balanced between I/O and computations. If real valued matrices were solved, the algorithm would be I/O bound due to the lack of computations needed to solve the problem.

Using this approach, most of the disk I/O is done via reads and each slab is written to disk only once when all of the computational work directly on it is done. The total number of slab reads required where N is the number of slabs in the matrix is:

$$\sum_{i=1}^{N-1} i = (N-2)(N-1)/2$$

and the total number is slab writes is N. The average I/O speed is that of the disk farm.

The basic slab solve algorithm also uses three buffers. One slab contains the right hand side (RHS) columns, one buffer is used to read in the next slab of the decomposed matrix, and the third buffer contains the slab for the current computation. The solve must read in the entire matrix a total of two times for a given set of RHSs. The forward elimination steps through the slabs from left to right and the forward elimination is performed with the RHS slab. The backwards substitution steps through the slabs of the matrix from right to left and finishes the solution of the RHS.

The parallelism for both the slab LUD and solve on the SV1 comes from the parallel BLAS routines available on that machine. When we began this work, a parallel version of the BLAS routines was not available so we started the development of distributed memory slab and block LUD and solve algorithms. A parallel version of the BLAS libraries was recently released and we plan on evaluating both the slab algorithms in the future.

2.1 X1 Slab LUD

For the X1 using a single MSP to decompose the matrix, the slab LUD algorithm is done exactly like the algorithm described in section 2.0.

For multiple MSPs on the X1, the slab LUD can either "cascade" the slabs from MSP 0 down the line of MSPs as a slab completes the computations that utilize a particular slab or the slab LUD can, in fairly "lock step," decompose M slabs where M is the number of MSPs by having each MSP other than MSP 0 copy the slab to their space as soon as MSP 0 has read the slab from disk. Either approach should significantly accelerate the decomposition process by utilizing the higher speed inter MSP communications. The average speed of the I/O gradually increases with the number of MSPs since the inter-MSP I/O is 40 gigabytes per second on board and 20 gigabytes per second off board. By the time there are 4 MSPs working on a problem, the total number of disk reads has been reduced by 75% and when 8 MSPs are used, the total number of disk reads is reduced by 87.5%. The algorithm only reads a slab one time from disk for every N-1 slabs that it is used against in computations where N is the number of MSPs.

The determination of the optimal algorithm will depend on the X1's approach to CAF copies. If computations are blocked on an MSP when a different MSP is copying data via a CAF Fortran 90 array construct, then the best approach will be the "cascade" even though there will be start up and stopping idleness in MSPs. If there is no block on the computations as other MSPs copy data from an MSP, then the "lock step" approach will be best by eliminating start up and stop idles of MSPs since all of the MSPs will be starting and ending pretty much at the same time.

2.2 X1 Slab Solve

The slab solve utilizes the same slab solve algorithm described in section 2 for the single MSP version. The multiple MSP version of the algorithm will utilize either the "cascade" approach or the "lock step" approach.

2.3 Benchmark Times

The slab version of this algorithm has only been run on one MSP at this point and in a contended environment. The data is for single MSPs using 2.7 gigabytes for the main buffer space on complex valued problems:

Unknowns	Problems	MSPs	WC Hours
77,500	8	8	35
133,000	4	4	208

3. Block Algorithm

The block algorithm was originally developed using OpenMP for a Compaq GS-80 and an SGI Origin 3800. The algorithm was also partially based on the slab LUD developed by CRI. Since OpenMP only allows one to use one node of the X1, the algorithm was converted to CAF.

3.1 Block LUD

3.1.1 Computational Algorithm

The block LUD first decomposes the first diagonal block against itself (block 1 in Figure 1). Next, the blocks to the right of (blocks 4 and 7 in Figure 1) and below (blocks 2 and 3 in Figure 1) the first diagonal block are decomposed against the diagonal block. All the work on this set of blocks can be done in parallel. The decomposed blocks to the right of and below the diagonal are then stored in co-arrays. The remaining blocks in the matrix can then be updated in parallel using the appropriate blocks stored in the co-arrays. For example, block six in Figure 1 would need blocks 3 and 4 and block 8 would need blocks 2 and 7. Once this pass through the matrix is complete, the algorithm decomposes the next diagonal block (block 5 in Figure 1) against itself and continues on in the manner described above. This process continues until the last diagonal block is decomposed against itself. All of the computations are done using the BLAS routines CGEMM, CTRSM, CSCAL, CTRSV, and CGEMV.

3.1.2 I/O and Memory Requirements

The algorithm uses a triple buffer system to hide as much I/O as possible. The three buffers allow one buffer to be reading in the next block, one buffer to be writing the previous block and one buffer is available for the current computation. The total number of read requests for this algorithm is

$$N_B^{1/2} + 2\sum_{i=1}^{N_B^{1/2}-1} i + \sum_{j=1}^{N_B^{1/2}} (j-1)^2 = \frac{2N_B^{3/2} + 3N_B + N_B^{1/2}}{6}, \tag{4}$$

where N_B is the total number of blocks in the matrix. Thus, as the number of blocks increases the *total* amount of I/O done increases. So, if more memory were available, a larger block size would be beneficial because it reduces the total number of blocks and, therefore, the total amount of I/O that needs to be done.

The total amount of memory that the algorithm uses is

$$(6 + \frac{2N_B^{1/2}}{N_{CPU}}) * N_{CPU} * N_{rows} * N_{cols} * 16, \tag{5}$$

where N_B is the total number of blocks in the matrix , N_{CPU} is the number of MSPs being used, N_{rows} is the number of rows in a single block and N_{cols} is the number of columns in a single block. The total number of accesses to blocks stored in co-arrays is

$$N_B^{1/2} \times N_{CPU} + 2\sum_{k=1}^{N_B^{1/2}-1} k + 2\sum_{l=1}^{N_B^{1/2}} (l-1)^2 = \frac{2N_B^{3/2} + N_B^{1/2}(3 \times N_{CPU} - 1)}{3}, \tag{6}$$

where N_B and N_{CPU} are as defined above.

3.2 Block Solve

3.2.1 Computational Algorithm

Once the entire matrix has been decomposed into the lower and upper triangular matrices, one can solve equation (3) for *y*. To do this, the block solve algorithm reads in the first diagonal block and does the forward elimination with the first chunk of right hand sides (RHS) denoted by rectangle 1 in the diagram of the RHS below.

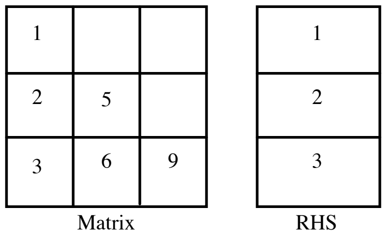

Figure 6

Next, each of the blocks below the first diagonal block are read in and the appropriate chunk of the right hand side matrix gets updated. For example, block 2 in the matrix will be used to update rectangle 2 of the RHS and block 3 of the matrix will be used to update rectangle 3 of the RHS. All of these updates can be done in parallel, and the right hand side matrix is stored as a co-array so each MSP has access to it. Following the completion of these updates, the next diagonal block to the right is read in and the same sequence of calculations occurs as in the previous step. This process continues until the last diagonal block is read in and the forward elimination is completed.

As soon as the forward elimination is complete, the backwards substitution is performed to solve *Ux=y* for *x*. The last diagonal block is read in and the backwards substitution is performed with the last chunk of right hand sides. Then all of the blocks above the last diagonal block are read in and the appropriate chunks of the right hand side matrix are updated. Again, all of these updates can be done in parallel. The next diagonal block to the left is then read in and the same process is followed as for the last diagonal block. This cycle continues until the first diagonal block has been read in and the backwards substitution is complete. One now has the solution, *x*, to equation (1). All of the computations are done using the BLAS routines CTRSM and CGEMM.

3.2.2 I/O and Memory Requirements

Unlike the block LUD, the solve algorithm only needs a double buffer system because a write buffer is not necessary. One buffer is used to read in the next block of *A* and one is for the current computations. For the current algorithm, the entire RHS matrix must fit in memory. An algorithm where the RHS matrix is out of core is currently under development. For the current algorithm with the RHS matrix in core, the total number of read requests is

$$2\sum_{m=1}^{N_B^{1/2}} m = N_B^{1/2}(N_B^{1/2}+1)\,, \qquad (7)$$

where N_B is the total number of blocks in the matrix. The total number of accesses to the RHS co-array is three times the number in (7).

3.2.3 Comparison of I/O Requirements for Slab vs. Block Solves

The block solve has an advantage over the slab solve in that it does about half the I/O that the slab solve does for moderate to large problems. For the forward elimination, the lower triangular part of the matrix is all that is used in the calculations. In Figure 7, the black portions of both matrices are the pieces of the matrix that are read in and actually needed for the calculation. The grey sections are the parts of the matrix that are read in and not needed for the computation. The white blocks are blocks that are not read in and not needed. It is clear from this that the block disk storage format does less I/O than the slab disk storage format. The backwards substitution is similar except that the upper triangular part of the matrix is needed instead of the lower triangular part.

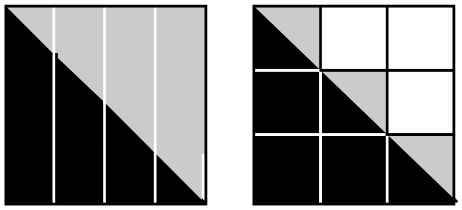

Figure 7

3.3 Benchmark Times

3.3.1 Block LUD with Buffered AIO

The block LUD was run with problem sizes of 36,864, 61,440 and 122,880 unknowns. Each run used buffered AIO and used a block size of 2048x2048 complex elements. Table 1 shows the CPU time, wall clock time and scaling for the 36k case run that used 324 total blocks. The one MSP run used 2.82 GB of memory and the 12 MSP run used 7.25 GB. The addition of an MSP adds about 400 MB to the total memory size of the single MSP run. Table 2 shows the CPU time, wall clock time and scaling for the 61k case that used 900 total blocks. The single MSP runs used 4.43 GB of memory and the 12 MSP run used 8.86 GB. Table 3 shows the CPU time, wall clock time and scaling for the 122k case that used 3600 total blocks. The single MSP run used 8.46 GB of memory and the 12 MSP run used 12.88 GB. One can see that the parallel scaling becomes much better as the problem size, and hence, the number of blocks, increases. We believe that the scaling drops off after 10 MSPs for the 122k case because we have reached a point where there is so much I/O that more MSPs will not improve the run time. We would need to increase the block size, thereby reducing the total amount of I/O, in order to improve the runtime on more than 10 MSPs (see section 3.3.3). Having more bandwidth will just push this point out to a higher number of MSPs. This point will be reached for any combination of problem size and block size eventually provided you have enough MSPs on your machine.

The 36k and 61k cases were run with CrayPat to see what the average FLOP rate was. For a single MSP run, the FLOP rate was about 10.9 GFLOPS for the 36k case and about 11 GFLOPS for the 61k case! This translates to 87% of the theoretical peak FLOP rate for a single MSP! The fact that the FLOP rate is so high shows that for a single MSP, the I/O is all effectively hidden and the algorithm is not I/O bound.

MSPs	CPU (hrs)	Wall (hrs)	Scaling
1	3.33	3.43	1.00
2	3.38	1.75	1.96
3	3.45	1.19	2.87
4	3.49	0.94	3.65
5	3.56	0.76	4.52
6	3.65	0.65	5.25
7	3.73	0.58	5.95
8	3.72	0.51	6.79
12	4.03	0.38	8.97

Table 1: Buffered AIO - 36,864 unknowns (324 total blocks)

MSPs	CPU (hrs)	Wall (hrs)	Scaling
1	15.36	15.82	1.00
2	15.49	7.94	1.99
3	15.57	5.34	2.96
4	15.69	4.04	3.92
5	15.81	3.29	4.82
6	15.92	2.78	5.70
7	16.07	2.42	6.54
8	16.07	2.10	7.55
12	16.59	1.54	10.27

Table 2: Buffered AIO - 61,440 unknowns (900 total blocks)

MSPs	CPU (hrs)	Wall (hrs)	Scaling
1	122.44	128.16	1.00
10	124.87	12.93	9.91
11	125.86	13.58	9.44
12	125.52	13.24	9.68

Table 3: Buffered AIO - 122,880 unknowns (3600 total blocks)

3.3.2 Comparison of Buffered C I/O, Buffered AIO, and Direct AIO

We had to spend some of our development time to determine what kind of I/O would be best suited for the block algorithm. Early on in the development we used direct Fortran I/O. There were a few bugs to work through, such as *sync_file* didn't work and if the I/O size was too small it looked like only one MSP was doing any I/O. Eventually it worked pretty well and we moved on to try using C I/O. We found that for the small cases we were testing that there was not too much difference in the run times so we continued development with C I/O because the AIO routines we eventually wanted to use were C routines. Once the AIO bugs were all fixed, we modified the code to overlap I/O with the computations and we began doing some timing tests.

For the 36k case, we ran the code using buffered C I/O, buffered AIO and direct AIO to compare the different I/O methods. Table 4 shows the difference in the wall clock time depending on whether buffered C I/O or buffered AIO is used. We again used blocks with 2048x2048 complex elements in them. For the 36k case, one can see that running a code that overlaps the I/O with computations is about 20% faster.

MSPs	C I/O (hrs)	AIO (hrs)	% Speedup
8	0.65	0.51	22
12	0.48	0.38	21

Table 4: Comparison of buffered C I/O and AIO - 36,864 unknowns (324 total blocks)

In Table 5, we show the difference between using direct AIO and buffered AIO. In order to use any form of direct I/O on the X1, one can only have a maximum of a 16MB I/O transfer size. Thus, we had to reduce the size of the blocks to be 1024x1024 complex elements, which increases the total number of blocks for the 36k case to 1296. One can see that for this problem, the two forms of I/O are comparable when using one and four MSPs, but once more MSPs are used, direct AIO is faster for a 16MB block size. However, in Table 1 we can see that using a bigger block size with buffered AIO produces faster runs than using direct AIO with a smaller block size when more than 4 MSPs are used. Of course it would be best to be able to use direct AIO with a bigger block size, but we are currently restricted by the current limits on the X1.

MSPs	Direct (hrs)	Buffered (hrs)
1	3.50	3.46
4	0.93	0.93
8	0.57	0.76
12	0.52	0.66

Table 5: Comparison of direct and buffered AIO - 36,864 unknowns (1296 total blocks)

3.3.3 Comparison of Different Block Sizes

We also varied the size of the blocks used for the 36k case and the 122k case to see what effect this has on the total run time.

For the 36k case, we ran the code with block sizes of 1024x1024, 2048x2048 and 4096x4096 complex elements. These sizes will be referred to as small, medium and large, respectively. The small case used 1296 total blocks, the medium case used 324 total blocks and the large case used 81 total blocks. Table 6 shows the differences in the timings for these different block sizes. On more than four MSPs, the medium block size gives the best performance. The large block size does not do as well because there are fewer blocks (only 81) and thus not as much parallel work to do.

MSPs	Small (hrs)	Medium (hrs)	Large (hrs)
1	3.46	3.43	3.50
4	0.93	0.94	1.02
8	0.76	0.51	0.64
12	0.66	0.38	0.54

Table 6: Comparison of different block sizes for 36,864 unknowns

The 122k case was run with block sizes of 2048x2048 and 4096x4096 complex elements, referred to as medium and large, respectively. We did not run this case with the small block size because we already know that the small block size in inefficient. The large case was not run on 1 MSP because it required about 18 GB of memory, which is more memory than it has available to it. The 12 MSP runs used about 13 GB for the medium case and 35 GB for the large case. Table 7 shows the differences in the times for the different block sizes. One can see that using the larger block size makes a significant difference in the wall clock time for the 12 MSP runs. This is because by increasing the block size we have reduced the total number of blocks and thus the total amount of I/O being performed. This decrease in the number of blocks does not hurt the runtime on multiple MSPs for this case because there are still enough blocks to keep all the processors busy.

MSPs	Medium (hrs)	Large (hrs)
1	128.16	n/a
10	12.93	13.32
11	13.58	--
12	13.24	11.35

Table 7: Comparison of different block sizes for 122,880 unknowns

3.3.4 Block Solve

We ran the block solve for the 36,864 unknowns test case with 900 right hand side columns. We used a block size of 2048x2048 complex elements for the matrix and the algorithm uses buffered AIO. Table 8 shows the times for these runs.

MSPs	CPU (min)	Wall (min)	Scaling
1	14.78	15.17	1.00
2	54.00	8.08	1.88
3	22.40	5.60	2.71
4	17.23	4.50	3.37
5	17.98	3.75	4.04
6	18.83	3.28	4.62
7	20.37	3.07	4.95
8	21.85	2.88	5.26
9	22.95	2.70	5.62
10	25.12	2.67	5.69
11	27.07	2.62	5.80
12	29.03	2.58	5.87

Table 8: RHS solve for 36,864 unknowns and 900 RHS columns

The scaling for the solve is not very good, but we expect the scaling to be better for the larger test cases because there are more blocks to work on in parallel. Larger test cases are currently being tested.

4. Problems Encountered

There were several problems we encountered as we developed the LUD algorithms. Most of the problems were software related, but there were also some hardware issues that slowed down the development.

4.1 Software Problems

A number of the problems we had were found when trying to do I/O from multiple MSPs using CAF. First, with Fortran I/O, if the I/O size was too small, it looked like only one MSP did I/O. This was fixed using an assign statement. Another problem was that the *sync_file()* call did not work. We need to close and reopen the file from all MSPs to ensure file coherency. Also, we had problems opening a file in C from multiple MSPs from a CAF program. This was resolved by setting the open and mode flags for the first MSP to create the file for read/write access and then open the file. All the other MSPs need to wait for this to complete and then each MSP can open the file for read/write access. The last problem we ran into was that the AIO routines would not run on multiple MSPs if you went off node. This has been fixed.

There are two other I/O issues that we have run into, but they are not related to doing I/O from multiple MSPs. First, is the 16 MB maximum limit for the I/O transfer size when using direct I/O. Direct I/O gives us much better I/O rates, but because the transfer size is so small, the overall performance of the code is worse than using buffered I/O with a larger transfer size as was seen in section 3.3. Being able to use direct I/O with a much larger transfer size would help greatly. Second, we are only getting about 50% of our peak I/O bandwidth. Our Cray support people expected that the more recent I/O drivers would fix this. Our I/O drivers were updated, but the performance didn't improve. Cray is still looking into this problem.

We ran into a strange problem on the AHPCRC's X1 where the code would run fine on 1, 2, 3, 4 and 6 MSPs but not on 5, 7 or 8 MSPs. Different sized test cases showed similar behaviour, but the number of MSPs it would work ok on would change. We sent this to Cray and it ran fine on the internal Cray X1. It was finally determined that the version of MPT that was on the AHPCRC's X1 at the time was causing the problem. Once MPT version 2.2.0.0 was installed, the code ran fine.

Another peculiar problem we have run into on our X1 is that occasionally when we run our codes (sometimes the LUD, sometimes other codes) using PBS, the job will stop due to a BPT Trace error. No other information is provided. Cray is currently working on this issue.

4.2 Hardware Problems

The biggest problem we had early in the development of the LUD code was that we had a limited amount of time in which we could log onto the AHPCRC's X1, and the X1 was down quite a lot. While this is expected with a new architecture, it really hindered early development because of the limited opportunities we had to log on. Some of the problems we ran into while the machine was available were leaking memory (when trying to run a code, the machine would say that resources weren't available and it looked like our code was trying to use 50 GB of memory even though it was really using less than 1 GB) and the CPES server was down a lot so we couldn't compile code.

Conclusion

The slab LUD shows every sign of working as proficiently on the X1 as it has on the C90 and SV1 architectures. The inter-processor I/O rates provide ample bandwidth to make up for the distributed memory aspects of the machine. Once the CAF and F90 interactions are well understood, the best algorithm will fall out and the software and algorithm can be tweaked to attain maximum speeds.

The block LUD performs very well on the X1, achieving 87% of the peak theoretical FLOP rate on a single MSP. It also scales fairly well with the number of MSPs for large problems. The block solve does not scale very well with the number of MSPs. We believe the scaling will improve for the larger test cases.

Acknowledgments

Thanks to AHPCRC for allowing us to use their X1 for development and testing of the LUD algorithms. The CSCF would also like to thank Bruce Black, Nathan Wichmann, John Kaitschuck, Kelly Lippard and others from Cray for all the help they gave us tracking down and resolving problems and for their suggestions during the development of the algorithms.

About the Authors

Marianne Spurrier is a Systems Engineer with Lockheed Martin and has been working with CSCF for three years. She can be reached via email at marianne.spurrier@lmco.com. Joe Swartz has been the manager of CSCF for seven years and has been actively involved in the supercomputing community for many years. He can be reached via email at joseph.h.swartz@lmco.com. Bruce Black is a Sr. Systems Engineer with Cray Inc. and is one of the several Cray employees working in support of CSCF. He can be reached via email at bdblack@cray.com.