*Anatomy of a Parallel Out-of-Core Dense Linear Solver

Kenneth Klimkowski
Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, Texas 78712
ken@ticam.utexas.edu

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
rvdg@cs.utexas.edu

Abstract – *In this paper, we describe the design and implementation of the Platform Independent Parallel Solver (PIPSolver) package for the out-of-core (OOC) solution of complex dense linear systems. Our approach is unique in that it allows essentially all of RAM to be filled with the current portion of the matrix (slab) to be updated and factored, thereby greatly improving the computation to I/O ratio over previous approaches. Experiences and performance are reported for the Cray T3D system.*

INTRODUCTION

Scalable, portable linear algebra libraries for distributed memory MIMD architectures that are currently being investigated put a limit on the size of the problem that can be solved, dictated by the aggregate RAM of the machine. A good example of this is the ScaLAPACK library for dense linear algebra [3]. This paper is concerned with the investigation of techniques for development of a dense linear solver package that is scalable both in machine and problem size.

A number of parallel out-of-core dense linear solver packages exist. These include Intel's Prosolver™ [4, 2] and a code developed at Sandia National Laboratories [9]. The current Prosolver™ package includes two OOC solvers: a block based solver and a slab based solver. The block solver brings essentially square blocks of the matrix incore, which improves the computation to I/O ratio, but limits pivoting to improve stability of the algorithm. Moreover, at any given time this approach juggles up to ten blocks in an effort to overlap computation and I/O. This limits the amount of memory that is dedicated to data on which useful computation is being performed. The current Prosolver slab solver works with column blocks and uses a full implementation of partial pivoting. It suffers from the fact that incore a linear data decomposition is used, which prevents scalability of the algorithm. Moreover, three to four slabs are being juggled at any given time, again in an effort to overlap computation and I/O. The Sandia approach is very similar to the Intel slab solver, except that it uses a two dimensional data decomposition for the incore computation, leading to a scalable incore algorithm. It also juggles four slabs simultaneously, with the same drawbacks.

Our code is based on a highly efficient incore solver, which was written for electromagnetics applications [5] and has also been used for acoustics applications [7]. We modified this code in a way that very naturally led to an OOC solver. This approach allows maximal code reuse, extending the robustness and manageability of the incore version to the OOC version. The incore computation uses a two dimensional data decomposition, yielding a scalable incore component, and does full partial pivoting, yielding the expected stability. While we also use a slab approach, we overlap computation and I/O at a finer level, which allows us to essential fill the entire incore memory with the current slab being updated and factored (the active slab). This gives us an edge by improving the computation to I/O ratio compared to previous approaches.

A PARALLEL INCORE DENSE LINEAR SOLVER

Our dense linear solver depends on the ability

*This work was performed in part at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration.

to perform an LU factorization of a given $n \times n$ matrix:

$$PA = LU$$

where L and U are unit lower and nonunit uppertriangular, respectively. Permutation matrix P represents the accumulation[1] of all pivots required for stability. Once this factorization is completed, the dense linear system $Ax = b$ can be solved by solving $Ly = Pb$ and $Ux = y$, the forwards and backward substitution phases.

Ignoring pivoting for now, a high performance version, known as a *blocked right-looking* variant of the algorithm can be generated by blocking the matrices:

$$\left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) \left(\begin{array}{c|c} U_{00} & U_{01} \\ \hline 0 & U_{11} \end{array}\right)$$

where A_{00}, L_{00}, and U_{00} are $k \times k$ matrices. Given this partitioning, the algorithm becomes: Factor the first *panel* of k columns, overwriting this panel with the result; solve the triangular system with multiple right-hand-sides $L_{00}U_{01} = A_{01}$, overwriting A_{01}; update $A_{11} \leftarrow A_{11} - L_{10}U_{01}$; recursively compute $A_{11} = L_{11}U_{11}$.

The algorithm is formulated so that the bulk of the computation is now in matrix-matrix operations like matrix multiplication, which can inherently achieve higher performance by overcoming the memory bandwidth bottleneck of many current generation microprocessors. Pivoting can be easily added.

To parallelize the above algorithm, one must first decide upon a data distribution. For scalability reasons, a two dimensional data decomposition is required, as is described in [8, 6]: Given p processors, we view these as *logically* forming a $r \times c$ mesh, where $p = rc$. In this mesh, $\mathbf{P}_{ij}$ denotes the (i, j)th processor. The matrix is partitioned like:

$$A = \left(\begin{array}{c|c|c|c} \tilde{A}_{00} & \tilde{A}_{01} & \cdots & \tilde{A}_{0(N-1)} \\ \hline \tilde{A}_{10} & \tilde{A}_{11} & \cdots & \tilde{A}_{1(N-1)} \\ \hline \tilde{A}_{20} & \tilde{A}_{21} & \cdots & \tilde{A}_{2(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline \tilde{A}_{(N-1)0} & \tilde{A}_{(N-1)1} & \cdots & \tilde{A}_{(N-1)(N-1)} \end{array}\right) \quad (1)$$

Here all blocks are of dimension $k \times k$, except those in the last row and column, which may be smaller. These blocks are now wrapped onto the logical mesh so that $\tilde{A}_{ij}$ is assigned to processor $\mathbf{P}_{(i \bmod r),(j \bmod c)}$.

Given this partitioning, the parallel algorithm becomes: Factor the first *panel* of k columns on the first column of nodes; broadcast the result within rows of nodes; solve the triangular system with multiple right-hand-sides $L_{00}U_{01} = A_{01}$; broadcast the result within columns of nodes; in parallel update $A_{11} \leftarrow A_{11} - L_{10}U_{01}$; recursively compute $A_{11} = L_{11}U_{11}$.

While the wrap mapping described above is necessary to maintain reasonable load balance throughout the computation, it is unreasonable to require the user to build the matrix, or the OOC solver to read it, as prescribed by that mapping. However, the matrix can be blocked into the alternative partitioning

$$A = \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \\ \hline B_{10} & B_{11} & \cdots & B_{1(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{(\mathbf{r-1})0} & B_{(\mathbf{r-1})1} & \cdots & B_{(\mathbf{r-1})(N-1)} \end{array}\right) \quad (2)$$

where B_{ij} is a matrix of dimensions *approximately* n/r *by* k. The matrix is now distributed among the processors (e.g. by reading it from a disk) so that all blocks B_{ij} are assigned to processor $\mathbf{P}_{i(j \bmod c)}$. I.e., the distribution uses blocking in the row direction, and wrapping in the column direction. It would appear that if the matrix is assigned to processors in this fashion, a redistribution of the matrix is necessary to be able to use the parallel incore solver routine that expects wrapping in both directions. However, notice that if each processor $\mathbf{P}_{ij}$ calls the incore solver without redistribution, a factorization and corresponding solve of a permuted matrix will be performed, which will not change the final answer if minimal care is taken. We can view this distribution as being physically blocked and logically wrapped in the row dimension, and physically and logically wrapped in the column dimension.

FROM IN-CORE TO OUT-OF-CORE

In this section, we describe how an incore factorization algorithm can be converted to an OOC version, motivating our implementation.

An alternative to the above described blocked right-looking LU factorization scheme is the

[1] The permutations are computed and applied as the algorithm progresses, but can logically be viewed as being applied upfront.

blocked left-looking variant. For this, one must perform an alternative partitioning of the matrix:

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) =$$
$$\left(\begin{array}{c|c|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline & U_{11} & U_{12} \\ \hline & & U_{22} \end{array}\right)$$

where A_{00}, L_{00}, and U_{00} are $n_1 \times n_1$ matrices, and A_{11}, L_{11}, and U_{11} are $k \times k$ matrices. The assumption is that the first n_1 columns have already been factored, i.e. L_{00}, L_{10}, L_{20}, and U_{00} are available, and have overwritten the corresponding parts of matrix A. From this, we derive the equalities

$$A_{01} = L_{00}U_{01} \quad (3)$$
$$\left(\begin{array}{c} A_{11} \\ A_{21} \end{array}\right) - \left(\begin{array}{c} L_{10} \\ L_{20} \end{array}\right) U_{01} = \left(\begin{array}{c} L_{11} \\ L_{21} \end{array}\right) U_{11} (4)$$

The algorithm then becomes: If $n_1 > 0$ compute U_{01} from (3). and update according to the LHS of (4); factor the result to obtain the RHS of (4), overwriting the corresponding parts of A; repartition the matrix so that $n_1 \leftarrow n_1 + k$ and repeat. Initially, $n_1 = 0$.

The bulk of the computation is still in matrix-matrix operations. The left-looking variant has two advantages over the right-looking variant for OOC computation: it requires less I/O: each element of a matrix is only written to disk once, and in case of a system failure, it is easier to recover while salvaging computation already performed.

Assuming that the factorization of

$$\left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline A_{20} \end{array}\right) \quad (5)$$

was obtained using a right-looking variant, the *updating* of

$$\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) \quad (6)$$

w.r.t. that factorization is a matter of *recreating the history of the right-looking algorithm*. I.e., the same pivots and updates that were applied to factor (5) must be applied to (6). We will call the data in (6) a *slab*, which will typically be the data to be kept in-core during the current stage of the computation.

Once one has a parallel in-core blocked right-looking LU factorization, the implementation of an OOC version now becomes: Read in the current slab; create a virtual factorization by calling the in-core factorization, except that during the factorization of the part that is to the left of the current slab, the factored portions are read from disk, before being applied to the current slab; once the factorization reaches the current slab, the algorithm proceeds exactly like the in-core factorization; write the current slab; continue with next slab. We call this approach a *virtual* factorization.

DETAILS

Due to space limitations, we limit ourselves here to an enumeration of the issues.

The constraints in designing the I/O interface for the Cray T3D include the fact that each I/O request generates a system call to the Cray Y-MP with a comparatively large startup overhead, in addition to the cost per data item transferred. Reading of large blocks is hence essential. In addition, open files use resources on the Cray Y-MP agent, so they should be avoided to reduce communication traffic between the T3D and the Y-MP. Other parallel systems can be expected to have similar constraints.

We earlier made the observation that although wrapping requires small blocks, the matrix can be generated or read from disk in large blocks. This naturally accommodates the requirement that large blocks be read. In our implementation, we store the matrix so that each *row of nodes* shares one file. This limits the number of open files required.

Starting with N panels, the number of panels that comprise a slab, K, is solely determined by available memory. Naturally, K is an integer multiple of the number of columns of processors. In general, we will have $\lfloor N/K \rfloor$ full slabs, and one, possibly empty, slab of width $n - \lfloor N/K \rfloor Kk$ columns. A simple approach to managing these slabs will read a slab, perform a virtual factorization as described above, and write the slab. Notice that this leaves the reading and writing of slabs totally exposed (not overlapped with computation). Since this constitutes essentially the only exposed I/O, and is of lower order complexity, good efficiency can be achieved. It should be noted that even this I/O could be almost completely overlapped, requiring only slight modification of the current code.

After performing a factorization of a matrix, applications usually solve for one or several right hand sides (RHS) by forward and backward substitution. Since the forward substitution is similiar to a virtual factorization, we can treat the RHS as another slab logically appended to the matrix. For the backward substitution, we apply panels of the factored matrix to the RHS in the reverse order. Thus, forward and backward substitutions are created by modifying the virtual factorization.

The triangular solve is an $\mathbf{O}(n^2)$ computation on $\mathbf{O}(n^2)$ data items. Thus the forward and backward solves are inherently I/O bound. Where in the factorization we used $p = r \times c$ processors, we can reduce the size of the processor grid during these triangular solutions as long as we maintain the row dimension, r, of the blocked matrix. Explicit column wrapping and the row blocking facilitates this choice. As a tradeoff, larger numbers of processors can improve I/O by reading more of the matrix from disk.

Checkpointing is the process of monitoring the progress of a computation such that the computation could be restarted after system crash. In our implementation, the factored matrix is sequentially read, factored, and written, and hence our checkpointing requires only knowledge of how far we have progressed. For simplicity, we currently checkpoint after writing each slab of the matrix to disk. However, our granularity of I/O operations is a panel, so we could checkpoint after each panel when writing out a slab.

Checkpointing provides all the necessary information for restarting. Restarting is the ability to continue a computation after crashes such as system failure, exceeding time limits, or exceeding disk quotas. Our approach adds two features to checkpointing: restarting and the ability to continue with a different number of processors.

The user interface for our package is simple. An application builds the matrix piecewise and stores it on disk. The application builds a single logical file, which has properties much like a matrix: row and column dimensions and a leading dimension. To fill parts of this logical file, we provide an I/O interface, called the Platform Independent Parallel I/O (PIPIO) interface, which hides all details of how the logical file is physically mapped to disk(s), as well as system dependent details. The interface automatically splits the logical file into the required r physical files. The RHS is handled similarly.

An advantage of this approach is that during the building of the matrix and RHS, the processors need not be configured as the logical mesh required for the parallel OOC solver. For our primary application, an electromagnetics application developed by the Jet Propulsion Laboratory, it is more convenient and efficient to generate the matrix by columns.

RESULTS

This section describes preliminary results obtained using the PIPSolver on the Jet Propulsion Laboratory's Cray T3D system.

The performance of the virtual LU factorization is very respectable because much of the I/O during that portion of the code is overlapped with computation. The observed I/O rates can be greatly improved, which would improve overall performance to very acceptable levels. The I/O rates are highly influenced by the load on the Y-MP frontend. This can greatly affect the observed I/O rate from one run to the next.

The T3D provided reliable high performance. To achieve this performance, it is essential to use highly optimized compute kernels like the Basic Linear Algebra Subroutines. Moreover, although we achieved reasonable I/O rates, the very high I/O rates (200 MBPS) promised by the Cray T3D continue to elude us.

CONCLUSION

We have described a simple approach to converting a highly efficient and scalable incore complex dense linear solver to an OOC implementation. In doing so, we designed an OOC version which has an arguably better computation to I/O ratio than other implementations with similar stability properties. The simplicity of the approach inherently yields a more maintainable and flexible code. The simple application interface is another attractive feature.

ACKNOWLEDGEMENTS

We are indebted to Tom Cwik of the Jet Propulsion Laboratory and Dan Katz of Cray Research for their expertise with the Cray T3D. In addition, we would like to acknowledge the Jet Propulsion Laboratory for providing generous access to their 256 node Cray T3D during the development and evaluation of the codes.

Table 1: Performance data on Cray T3D. For each run, we report timing data for the factorization only, giving the exposed reading and writing of slabs, as well as the virtual factorization (which includes some I/O). We grouped the timings so that the *number of slabs* are approximately constant. This gives a better insight into the scalability of the method. MBPS are aggregate I/O rates, reported in megabytes per second. MFLOPS reported are *per processor*.

n	p	$r \times c$	# of	Read		Write		Virtual LU		All	
			Slabs	Time	MBPS	Time	MBPS	Time	MFLOPS	Time	MFLOPS
5000	8	2×4	0.96	7.4	54.4	18.1	22.1	4436.5	95.5	62.0	90.2
7000	16	4×4	0.94	16.7	47.0	37.9	20.7	598.0	95.6	652.6	87.6
10000	32	4×8	0.96	46.9	34.1	198.0	8.1	872.3	95.5	1117.2	74.6
14000	64	8×8	0.94	100.3	31.3	237.2	13.2	1210.3	94.5	1547.8	73.9
6000	8	2×4	1.39	10.7	54.0	21.2	27.2	762.0	94.5	793.9	90.7
8400	16	4×4	1.35	21.1	53.6	69.9	16.1	1067.5	92.5	1158.5	85.3
12000	32	4×8	1.39	68.7	33.5	139.0	16.6	1614.8	89.2	1822.5	79.0
16800	64	8×8	1.35	108.5	41.6	257.8	17.5	2311.4	85.5	2677.7	73.8
8000	8	2×4	2.50	43.0	23.8	55.7	18.4	1820.8	93.7	1919.5	88.9
11200	16	4×4	2.41	92.7	21.6	173.0	11.6	2679.0	87.4	2944.7	79.5
16000	32	4×8	2.50	141.7	28.9	267.9	15.3	3991.6	85.5	4401.2	77.6
22400	64	8×8	2.41	179.5	44.7	484.0	16.6	5855.8	80.0	6519.3	71.8

References

[1] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide,* SIAM, Philadelphia, 1992.

[2] E. Castro-Leon and M. L. Barton, "The Prosolver(TM) Libraries : Parallel Solvers for Intel High Performance Scalable Computers", Internal Document, Intel Scalable Systems Division.

[3] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation.* IEEE Comput. Soc. Press, 1992, pp. 120-127.

[4] T. Cwik, J. Patterson, and D. Scott, "Electomagnetic Scattering Calculations on the Intel Touchstone Delta," *Proceedings of Supercomputing '92,* IEEE Comput. Soc. Press, 1992, pp. 538–542.

[5] Tom Cwik, Robert van de Geijn, and Jean Patterson, "Application of Massively Parallel Computation to Integral Equation Models of Electromagnetic Scattering," *Journal of the Optical Society of America A,* Vol. 11, No. 4, April 1994, pp. 1538–1545

[6] Jack. J. Dongarra, Robert A. van de Geijn, and David W. Walker, "Scalability Issues Affecting the Design of a Dense Linear Algebra Library," *Journal of Parallel and Distributed Computing,* Vol. 22, No. 3, Sept. 1994, pp. 523–537.

[7] P. Geng, J. T. Oden, and R. A. van de Geijn, "Massively Parallel Computation for Acoustics Scattering Problems using Boundary Element Methods," *Journal of Sound and Vibration,* (to appear).

[8] B. A. Hendrickson and D. E. Womble, "The Torus-wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers," *SIAM J. Sci. Comput.*, vol. 15 (1994), no. 5, pp. 1201-1226.

[9] D. E. Womble, D. S. Greenberg, R. E. Riesen, and S. R. Wheat, "Out of Core, Out of Mind: Practical Parallel I/O," *Proceedings of the Scalable Libraries Conference,* Oct. 6-8, 1993, Mississippi State University, pp.10–16.