Annales UMCS Informatica AI 1 (2003) 191-199

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Shifting technique vs. pointer structures in unsymmetric sparse linear equations systems solver

Marek Stabrowski*

Department of Computer Science, Technical University of Lublin,
ul. Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The research reported in this paper presents a new idea of the storage structure of sparse matrices. This structure is used in the multi-option solver of linear equation systems with unsymmetrical sparse coefficient matrices. The new solver is compared comprehensively with the analogous (identical numerical method used) solver of the classical type. The tests of new solver have been performed on a quite broad spectrum of hardware platforms.

1. Introduction

This paper presents two basic approaches to development of sparse linear equation solver (C++ language [1]) for unsymmetric systems. The software is targeted at uniprocessor computers. Special attention has been paid to optimum memory usage and flexible pivoting. The Gauss method, with the option of partial or full pivoting [2], forms numerical basis of the solvers.

Storage structures of sparse matrices try to take into account both processing flexibility and economy of storage space (disk area, operational memory). Standard Harwell-Boeing storage format [3-4] will be presented in the next chapter.

Pivoting methods are usually classified as partial methods (pivot search limited to current leading column) and full ones (search in the whole submatrix to be eliminated). In the case of partial pivoting physical rows interchange is usually avoided and indirect row indexing is used. However, in the realm of sparse linear equation systems this elegant approach is impossible due to fill-in of nonzeros.

Pivoting methods are further divided into static and dynamic scheduling types. In static pivot scheduling preliminary symbolic forward elimination determines locations of fill-ins. Dynamic scheduling performs computations

* *E-mail address*: mmst@bravo.pol.lublin.pl

from the start of LU decomposition, determines and performs fill-ins on the fly. Static scheduling performs less shifting work, but numerical stability is inferior to the dynamic version. Dynamic pivot scheduling will be used in both solvers presented in this paper.

2. Monolithic matrix vs. pointer array structure

Classical sparse solver, developed for the comparison purposes, with monolithic coefficient matrix, uses the rowwise version of Harwell-Boeing sparse matrix storage format. The idea of rowwise storage [3-5] can be illustrated with an example of small unsymmetric 4x4 matrix which the in conventional mathematical notation has the form:

$$\begin{Bmatrix} 1. & -2. & 0 & -3. \\ 0 & -2. & 0 & 4. \\ 3. & 0 & -3. & 1. \\ 0 & 2. & 0 & 4. \end{Bmatrix}. \tag{1}$$

In the rowwise variant of Harwell-Boeing format this matrix is described with three arrays:

- `rowptr` with the pointers to row starts in the following two arrays;
- `colind` with the indices of columns with nonzero elements in the individual rows;
- `values` with numerical values of nonzero elements.

The contents of these three arrays for sample matrix (1) is given below in Table 1.

Table 1. The contents of Harwell-Boeing arrays for sample matrix (1).

`rowptr`	1	4	6	9	11					
`colind`	1	2	4	2	4	1	3	4	2	4
`values`	1.	-2.	-3.	-2.	4.	3.	-3.	1.	2.	4.

This format is quite compact but fill-in during LU decomposition results in large overhead due to massive shifts of data in operational memory. It is quite easy to observe that fill-in in any row results in shifting the rest of `colind` and `values` arrays (Fig. 1). In the later course this type of solver will be called shifting type one.

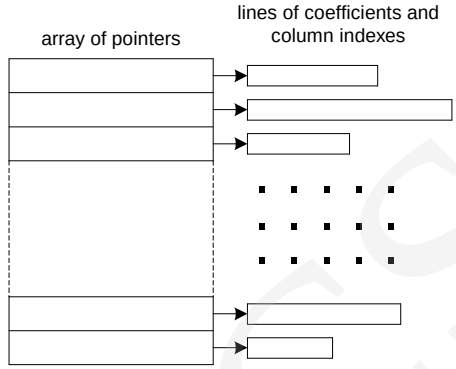

Fig. 1. New storage structure of sparse matrix using the array of pointers and dynamic lines

In has been decided to divide sparse matrix more strictly into separate rows in order to reduce the number of shifting operations. Gaussian elimination of a specific row affects only small part of matrix due to its sparsity. Therefore the shifts will occur only in the arrays corresponding to the rows with actual fill-in. Implementation of this idea makes use of the advanced data structures, replacing compressed and compacted coefficient matrix with the loser structure based on the pointer array (Fig. 2).

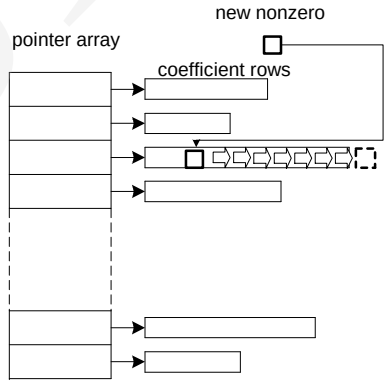

Fig. 2. Fill-in in pointer type solver affects only single row

The pointers are used in dynamic allocation of memory for individual sparse matrix rows. The pointer nodes (Fig. 2) contain information on dynamically allocated space for column indexes and on coefficients values located in the single matrix row. Moreover, the pointer node contains also the information on matrix row length, i.e. on the number of nonzeros. The solver using such data structure will be called pointer solver in the further course.

Row interchange may be handled locally due to the pointer array introduction. In this case mere interchange of pointer nodes contents is sufficient. Appropriate C++ code, shown below, is very concise and elegant:

```
ONEROW TempStruct;
TempStruct = *(AllRows + irow1);
*(AllRows +irow1) = *(AllRows + irow2);
*(AllRows + irow2) = TempStruct;
```

The pointer `AllRows` points to dynamic array of all `ONEROW` pointer nodes. The variables `irow1` and `irow2`, quite self-explanatory, are the indices of the rows to be swapped.

3. Comparison of shifting and pointer type solvers

Replacing of massive data shifting with local data manipulation in the pointer solver should lead to CPU time consumption reduction. As a test basis several standard unsymmetrical sparse matrices from the Harwell-Boeing and University of Florida collections [3, 5] have been used. The matrices have been used in the original uncompressed form and in the compressed one. Matrix compression in the case of sparse matrices concentrates more tightly the nonzeros around the main diagonal. It is frequently asserted that compression reduces LU decomposition time. Both these theses have been tested experimentally.

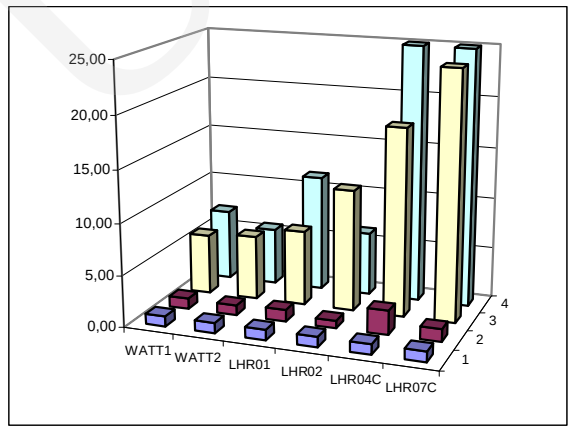

Fig. 3. Relative LU decomposition time: pointer solver 1) matrix compressed, 2) matrix uncompressed, shifting solver 3) matrix compressed, 4) matrix uncompressed

In two test series the size of matrices ranged from 1000 to more than 13000 and timing of pointer solver (full pivoting) operating on compressed matrices has been used as the reference. Most spectacular is the performance difference of pointer and shifting solver. For the first set of matrices (Fig. 3) the time ratio is 5

to about 30 in favor of pointer solver. Similar observation can be made for the second set of examples (Fig. 4) encompassing extremely unsymmetric and sparse matrices. Timing ratio is a bit lower than for the first series but the difference is most spectacular for the largest Gru35 matrix.

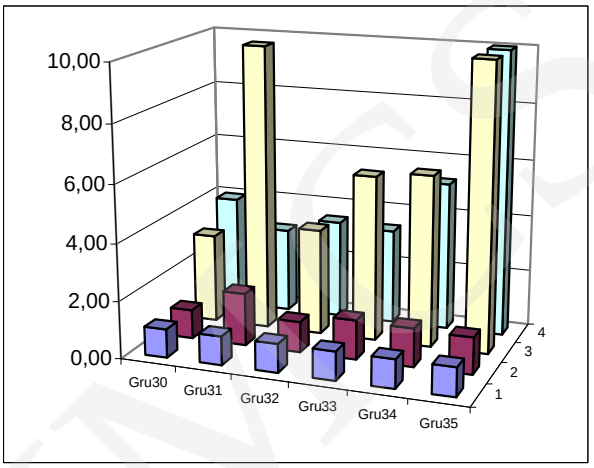

Fig. 4. Relative LU decomposition time: pointer solver 1) matrix compressed, 2) matrix uncompressed, shifting solver 3) matrix compressed, 4) matrix uncompressed

Matrix compression (first and third row in the plots) influences the timing of LU decomposition. For pointer solver matrix compression speeds up this process but the difference is far from spectacular. Moreover there is one evident exception of LHR02C matrix with compression slowing the LU decomposition. For the conventional shifting type solver the situation is far from clear. The only conclusion making sense is that compression benefits depend on the problem at hand.

4. Full and partial pivoting in sparse matrices

The pivoting in the pointer solver has been implemented in both partial and full versions with dynamic scheduling. It is quite obvious that partial pivoting incurs less overhead during both search of pivot (only current column is scanned and not whole not fully processed submatrix) and row interchange (no column interchange is performed).

This observation is confirmed by numerical experiment including also investigation of compression influence (Fig. 5). As with the results already reported the LU decomposition times are related to the partial pivoting solver operating on the compressed matrix (reference time = 1.0).

Numerical experiment confirms the thesis that partial pivoting is less time-consuming than full pivoting. The timing difference depends on the matrix

processed but generally it grows with the matrix size. Matrix compression speeds-up LU decomposition in the case of partial pivoting except WATT2 matrix. Speed-up is larger for larger matrices. Situation is different from that already mentioned, for full pivoting.

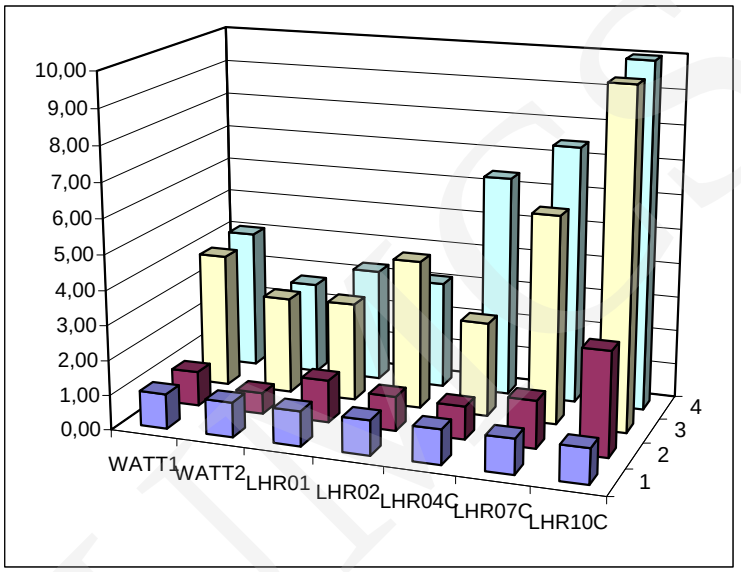

Fig. 5. Relative LU decomposition time by the pointer solver: partial pivoting: 1) matrix compressed, 2) matrix uncompressed, full pivoting: 3) matrix compressed, 4) matrix uncompressed

5. Dense solver vs. new pointer structures solver

A stranger new to the field of sparse matrices may be deterred by the structures used in this field, by special preparation and assembly of input data and at least by proprietary character of some part of the software. Moreover, dropping prices of the hardware may encourage resorting to brute force approach. Such an approach relies on simple application of classical dense solvers on new computer with larger operational memory and faster processor. It works almost satisfactorily in the case of small matrices - perhaps the limit is at 1000 rows/columns. With a rising size of the matrix the performance of dense solver spectacularly deteriorates and finally it fails to do the job.

These theses have been verified experimentally with the dense solver derived from the classical academic source [6]. The dense solver uses partial pivoting and therefore it has been compared primarily with the pointer solver in the partial pivoting version. However, the data on the full pivoting pointer solver have been also included in Tab. 2.

The data on processing time in Tab. 2 show that the pointer solver in the partial pivoting version is typically 2 to 3 times faster than the dense counterpart. Even the full pivoting pointer solver outperforms (timing) in most cases the dense solver.

Table 2. Comparison of the dense solver [6] and the sparse pointer solver (partial and full pivoting) with respect to operational memory usage and processing time

matrix			largest array size			CPU time in sec		
	size n	initial nonzeros	dense solver = n^2	nonzeros part.piv.	nonzeros full piv.	dense solver	point s. part.piv.	point s. full piv.
WATT1	1856	11360	3444736	135518	546862	59	19	69
WATT2	1856	11550	3444736	145045	548594	60	19	70
Gru30	3268	27836	10679824	913763	253388	337	165	54
Gru31	3008	27576	9048064	803459	547028	279	137	128
Gru32	3268	27836	10679824	849531	264078	335	150	58
Gru33	3008	27576	9048064	762225	408260	280	134	102
Gru34	3083	21216	9504889	396076	521168	275	64	119
LHR01	1477	18592	2181529	217295	338874	38	17	40
LHR02	2954	37206	8726116	541529	1300180	249	73	297
LHR04C	4101	82682	16818201	1066297	2350628	793	239	781

More decisive is the operational memory usage. For the dense solver, the largest array size is simply equal to n^2, where n is the rows/columns number. If such a array should be located in operational memory only, then for double precision and memory size of 256 MB, the limit of n is approximately 5500. The matrix LHR04C (tab. 2 – last row) occupies in the dense version about 134 MB. The sparse solver stores only nonzeros and it follows from tab. 2 that that memory consumption, even after inevitable fill-in, is still lower by one order of magnitude than in the case of dense solver. In the case of matrix Gru32 this ratio reaches even 40 in favor of sparse solver with full pivoting. even with no pivoting, the situation is different due to inevitable fill-in. The matrix LHR04C in the sparse solver, after expansion due to fill-in occupies only 12 MB of memory – not 134 MB. One may argue that the virtual memory technique enables dense processing of large matrices but timing overhead is prohibitive.

As a final conclusion it should be stated that the sparse solver offers distinct timing advantages over the dense one. For larger equation systems, the sparse technique is the only reasonable solution, if the penalty of virtual memory should be avoided.

6. Hardware dependence of pointer solver performance

The tests of the software developed in the course of present research have been performed on the assorted Unix-type platforms. The hardware encompassed 32-bit Pentium processor and several 64-bit processors (older and modern SUN products, new Itanium of Intel).

Table 3. Hardware dependence of LU decomposition time
for the assorted sparse matrices – time in seconds

computer & operating system	compiler	LHR01	LHR02	LHR04C	LHR07C
Pentium 800MHz + Linux	gcc	42	303	789	4263
Itanium 900 MHz + HP UX	HP	19	143	438	2072
Ultra10 300 MHz + Solaris8	gcc	135	1000	2958	14556
Ultra10 300 MHz + Solaris8	SUN	132	641	2779	14945
Blade100 500 MHz + Solaris8	gcc	61	450	1291	6060

The best results in the sense of CPU time have been achieved on Pentium and Itanium platforms. For approximately the same clock frequency Itanium times are roughly one half of Pentium times. It coincides inversely with the length of the processor word. One may expect reduction of Pentium times for higher clock frequency. It is reasonable to suppose that quadrupling of clock will result in outperforming of Itanium.

The tests on SUN's Ultra platform have shown that GNU gcc compiler is practically equivalent to the hardware vendor compiler. Modern and most expensive, among computers tested, Blade100 is slower than the old and cheap Pentium.

7. Conclusions

Detailed observations and conclusions have been already presented in the preceding chapters. Summing up, two major facts should be considered.

First of all, the tests using assorted matrices have proved that spectacular benefits are gained through introduction of advanced pointer type structures.

The second major observation is related to the compression of sparse matrices. It follows from numerical experiments that reduction of LU decomposition time depends on the matrix structure and quite frequently matrix compression affects adversely this time.

Seemingly naïve, but pragmatically sound, question of the advantages of sparse approach vs. dense one, has been resolved in favor of sparse solver. Sparse solver in the partial pivoting version offers distinct processing time reduction. But more convincing is the capability to process far larger equation systems – by at least one order of magnitude. No resorting to virtual memory usage is necessary and no accompanying processing time expansion occurs.

Optimum hardware selection for this class of problems depends on the problems size and price considerations. If 64-bit addressing capability is required, the best choice with respect to performance (and price) is the Itanium processor. In other cases modern and relatively cheap Pentium is an evident winner.

References

[1] Gajewski R. R., Lompies P., *Object-oriented approach to the reduction of matrix bandwidth, profile and wavefront*, Advances in Engineering Software, 30 (1999) 783.
[2] Demmel J.W., *Applied Numerical Linear Algebra*, SIAM, Philadelphia, (1997).
[3] Duff I.S., Grimes R.G., Lewis J.G., *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989) 1.
[4] Engeln-Muellges G., Uhlig F., *Numerical Algorithms with C*, Springer Verlag, Berlin (1996).
[5] Zitney S.E., Mallya J.U., Davis T.A., Stadherr M.A., *Multifrontal vs. frontal techniques for chemical process simulation on supercomputers*, Computers in Chemical Engineering, 20 (1996) 614.
[6] Forsythe G., Malcolm M.A., Moler C.B., *Computer Methods for Mathematical Computations*, Prentice Hall, Englewood Cliffs, (1977).
[7] Pandit S., Soman s. A., Khaparde S. A., *Design of generic direct sparse Linear System Solver in C++ for power system analysis*, IEEE Transactions on Power Systems, 16 (2001) 647.