Verkade: LU-Factorisation on Sparse Matrices

Micky Faas

October 11, 2018

1 Introduction

Verkade is an implementation of LU-factorisation on sparse matrices using the simple Gaussian elimination method with partial pivotting. Partial pivotting lends itself very well for sparse matrices as only row or column permutations are required but not both. Therefore the matrix data can be stored either by row or by column. Verkade stores matrices by row in *compressed row storage* (CRS)[1].

To load matrix data, the *matrix market* format[2] is used. The various test matrices described in this document are obtained from Suite Sparse[3].

2 Implementation

LU-factorisation is performed in situ on the CRS input data. Both the L and the U result matrix are stored at the same location: for a given row i all columns $0 \ldots j-1$ belong to the L matrix while all columns $j \ldots n-1$ belong to the U matrix.

2.1 Fill-ins

When transforming a sparse matrix using Gaussian elimination, it often happens that the density of the matrix increases. When pivot row i is subtracted from row k, $A(i,j)$ may be non-zero while $A(k,j)$ is zero such that the resulting operation $A(k,j) = A(k,j) - rA(i,h)$ is also non-zero. This means an extra non-zero value has to be added into row k, which expands the entire values-array in the CRS. In Table 3 we see that the factor in which the rows expand can be quite dramatic. To accommodate for these variations in length, rows i and k are often converted into dense representation first (scatter-gather). However, our experiments showed that this increased computation time by a factor of 10 compared to generating CRS immediate without scatter-gather. To this end we implement some simple form of memory management that supports heap-style allocation.

2.2 Memory Management

Verkade implements a simple memory management that supports allocating and freeing memory regions just like in C. It does this by maintaining a meta array of region descriptors separately from the CRS data. In the simplest form this

Matrix	Elements	Density	Elements after	Density after	Fill-in ratio
mcfe	24382	.042	87396	.15	3.58
c-21	32157	.0026	4915965	.4	152.87
flowmeter5	67391	.0007	1182851	.013	17.56
epb1	95053	.00044	3079993	.014	32.40
meg4	46842	.0014	746284	.022	15.93
cell1	34855	.0007	474506	.0095	13.61
nopoly	70842	.00061	15417152	.13	217.63
mhd4800b	27520	.0012	33490	.0015	1.22
ex10	54840	.0094	183234	.032	3.41
aft01	125567	.0019	1713317	.025	13.64

Table 1: Ratio for fill-in after LUP transformation.

Matrix	Original size (KiB)	Size after LUP (KiB)	Bytes lost (KiB)
mcfe	190	684	27
c-21	251	38405	3582
flowmeter5	526	9241	1010
epb1	742	24062	3188
meg4	365	5830	204
cell1	272	3707	18
nopoly	553	120446	4666
mhd4800b	215	261	5
ex10	428	1421	27
aft01	980	13385	726

Table 2: Memory footprint and loss due to fragmentation.

meta array only contains free regions: when a row is expanded it leaves a 'hole' in the array and the new data is moved to the back of the occupied space. A list of these free regions is later used to assign memory to other expanding rows.

A problem with this approach is the inherent fragmentation of the memory space. To counter fragmentation, we also need to store the occupied regions in the CRS array. This makes bookkeeping the meta array much more complex, but it enables us to move the row data around later to clear space when it is needed. In practice, however, the amount of data lost by fragmentation was so small that it did not outweigh the overhead caused by the more complex memory management. Table 3.1 shows the loss due to fragmentation while Table 3.2 shows the performance of the algorithm for both memory managers. These times were obtained by running LUP and the substitution to five different result vectors on a Core-i7 machine.

2.3 Numerical stability

Compared to complete pivotting, the method of partial pivotting can lead to much more numerical instability. In this case there is a trade-off to make because partial pivotting is much more suited for use with the simple CRS format for sparse matrices. In order to measure the effect of numerical instability on the test matrices, the Euclidian norm was used: $\frac{||\tilde{x}-x||}{x}$ was computed for each solu-

Matrix	Only free regions (s)	Full memory + defragmentation (s)
mcfe	0.054	0.12
c-21	19.16	43.91
flowmeter5	7.04	19.89
epb1	20.71	60.28
meg4	2.30	5.30
cell1	1.64	4.66
nopoly	131.21	376.22
mhd4800b	0.17	0.26
ex10	0.32	0.70
aft01	6.58	22.13

Table 3: Timings for LUP and substitutions for both memory models.

tion vector. Here x is the pre-computed vector and $||\tilde{x}||$ is the vector computed by the algorithm. $||$ is simply the square-root of the sum of squared elements of the vectors.

For almost all vectors the variance was exactly zero. This means no instability occurred. Only for the `nopoly` matrix any variance was measured (1, 0.656, 0.021, 0.012, 0.008 for vectors x_1 to x_5 respectively). The `nopoly` matrix was also a very hard matrix to factorise in terms of time and memory, which probably suggest it is not suitable to compute it with partial pivotting (it is not invertible).

References

[1] Compressed Row Storage. `http://www.netlib.org/linalg/html_templates/node91.html`

[2] The Matrix Market fileformat. `https://math.nist.gov/MatrixMarket/formats.html`

[3] SuiteSparse, Tim Davis et al. `https://sparse.tamu.edu/`