Scalable Factorization and Classification

Using Sparsity, Using Multi-Core

Gary Howell, HPC/ITD

NC State University

gary_howell@ncsu.edu

Thanks to ..

The results on scalability of classification algorithms were co-authored by

- Atina Brooks – SAS
- Qianyi Zhang – Statistics Dept., NCSU

Chemical data sets as arrays

Chemical data sets are large and somewhat sparse, i.e., if they are considered as arrays, then perhaps ten per cent of the entries are nonzero. Data sets can consist of thousands or millions of chemicals.

Sparse array storage allows processing of large data sets be stored and minimizes the number of computations. But computation with sparse matrices is typically orders of magnitude slower (per flop) than with the dense matrix package LAPACK.

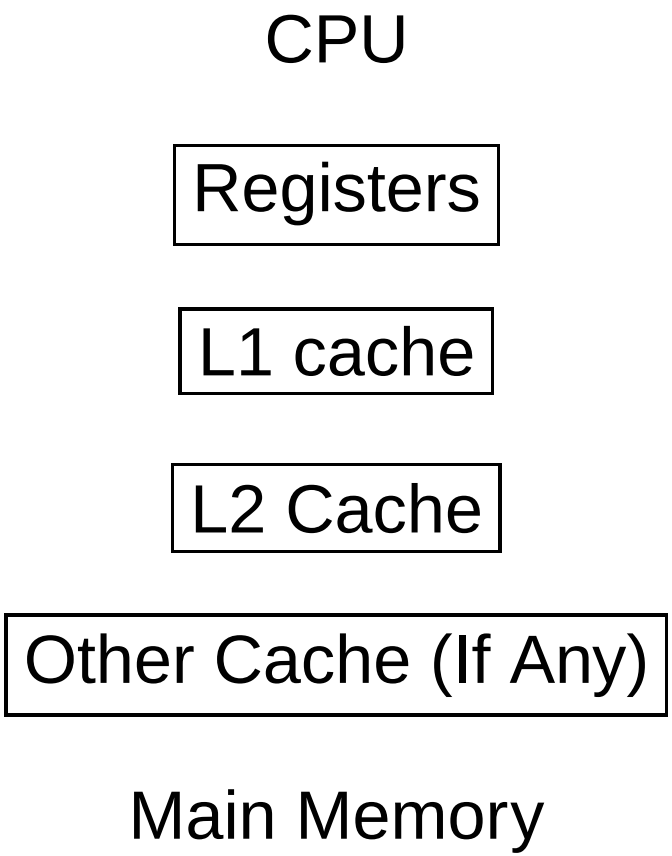
Sparse matrix techniques

Applying recent experience in improving computational efficiency in the LAPACK SVD algorithm, See LAPACK Working note 147, Howell, et al, we can solve sparse least squares problems with similar efficiency. Providing an efficient sparse SVD computation in the package R will be of wide use to the statistical community as well as aiding in handling the drug discovery data sets.

Similar techniques can be applied to the rSVD (robust singular value decomposition) and hopefully to other algorithms presented here.

Memory Hierarchy

Assumption: algorithm execution time is proportional to amount of data transferred from cache to main memory.



Clocks To CPU

CPU

Registers – 1 clock

L1 cache – 2-3 clocks

L2 Cache – 6-12 clocks

Other Cache (If Any)

Local RAM 20-200 clocks

Hard Drive 10^7 clocks

Distributed RAM 10^5 clocks

BLAS and LAPACK

use cache memory. When porting software packages, e.g. the statistical package R, one task is to find machine tuned BLAS and LAPACK packages and link R to them.
For dense matrix operations, using machine tuned BLAS and LAPACK libraries, can give dramatic speedups.

For example, a crystallography code solved $Ax = b$ by using a routine from the popular book *Numeric Recipes*. Replacing those calls by calls to LAPACK and BLAS reduced run times from ten hours to ten minutes.

60 fold speedup

In that case it was the order of accessing data. Processor speeds have gone up dramatically in the last few decades. Speed of accessing data plucked randomly form the Random Access Memory has increased much less dramatically. Accessing data from RAM can take a few hundred clock cycles.

By accessing data in order, throughput can be increased by an order of magnitude to perhaps ten per cent of theoretical peak speed.

By blocking a computation so that data fetched from RAM is stored in cache and re-used, $Ax = b$ can be solved at near processor peak speed.

Sparse Data Sets

If chemical data sets are represented as arrays, the arrays are usually sparse, i.e., most entries often all by a few per cent, are zero.
Storing array as sparse (only storing the nonzero entries and their indices) can save a good deal of storage so that larger jobs (perhaps ten times larger) times larger can be done with a given number of processors. Algorithms using the sparse array can save perhaps 90% of the computations.
If algorithms accessing the array can access data sequentially, we can expect to get about ten per cent of peak speed. So if we reduce the number of floating point operations by a factor of ten, we get about the same overall speed as using LAPACK.

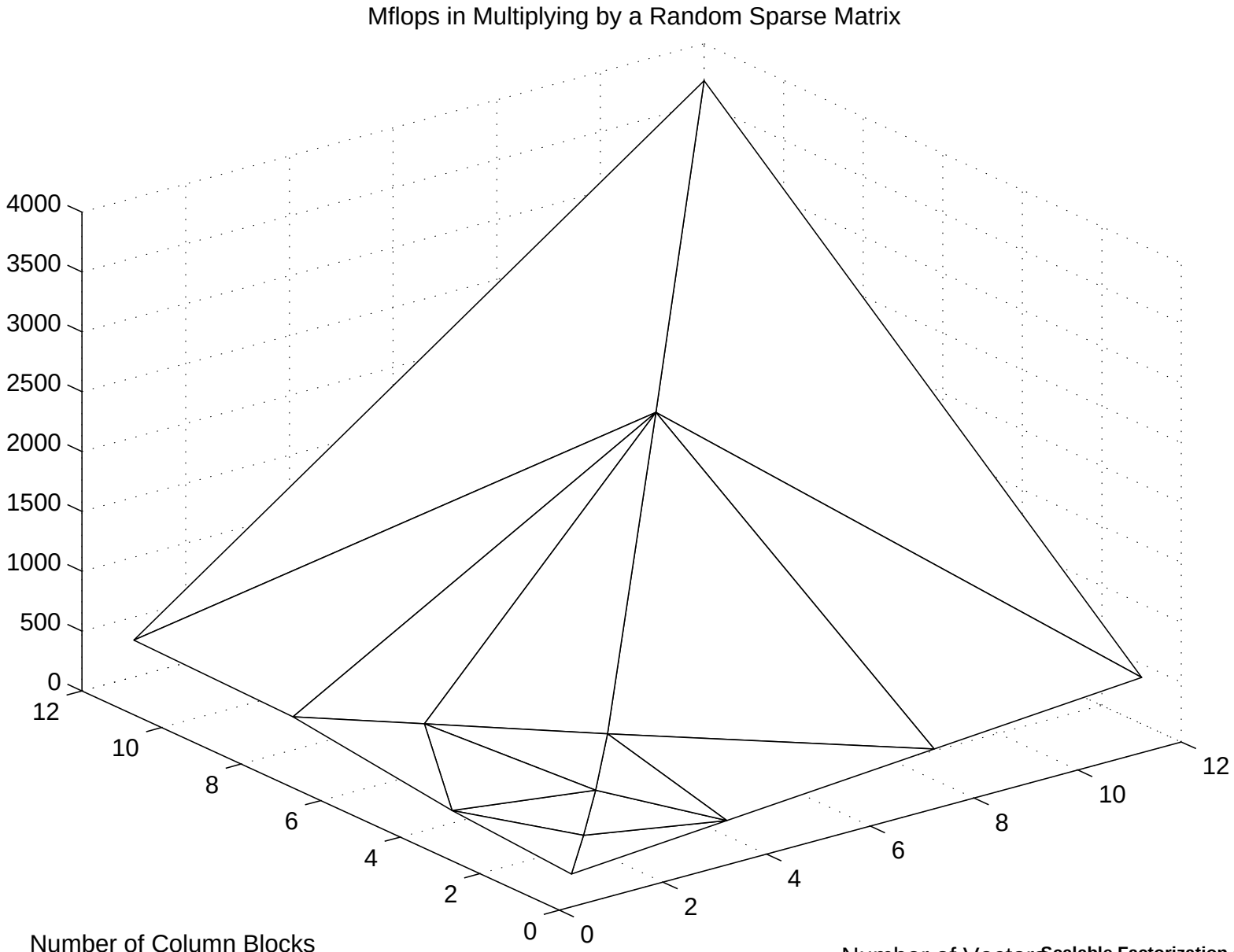
Dense Algorithms for Sparse

Advantages: we get good stability (full numeric accuracy). Flops go at almost the full theoretical peak speed.
Disadvantages: Suppose we get $A = U\Sigma V^T + E$ where $\|E\|/\|A\| = O(10^{-16}$. Then even if A is sparse and takes only a reasonable amount of storage, U and V are dense, so take a good deal of storage. Also computing with sparse A is slow per flop compared to computing with dense A.
To get around the storage issue represent

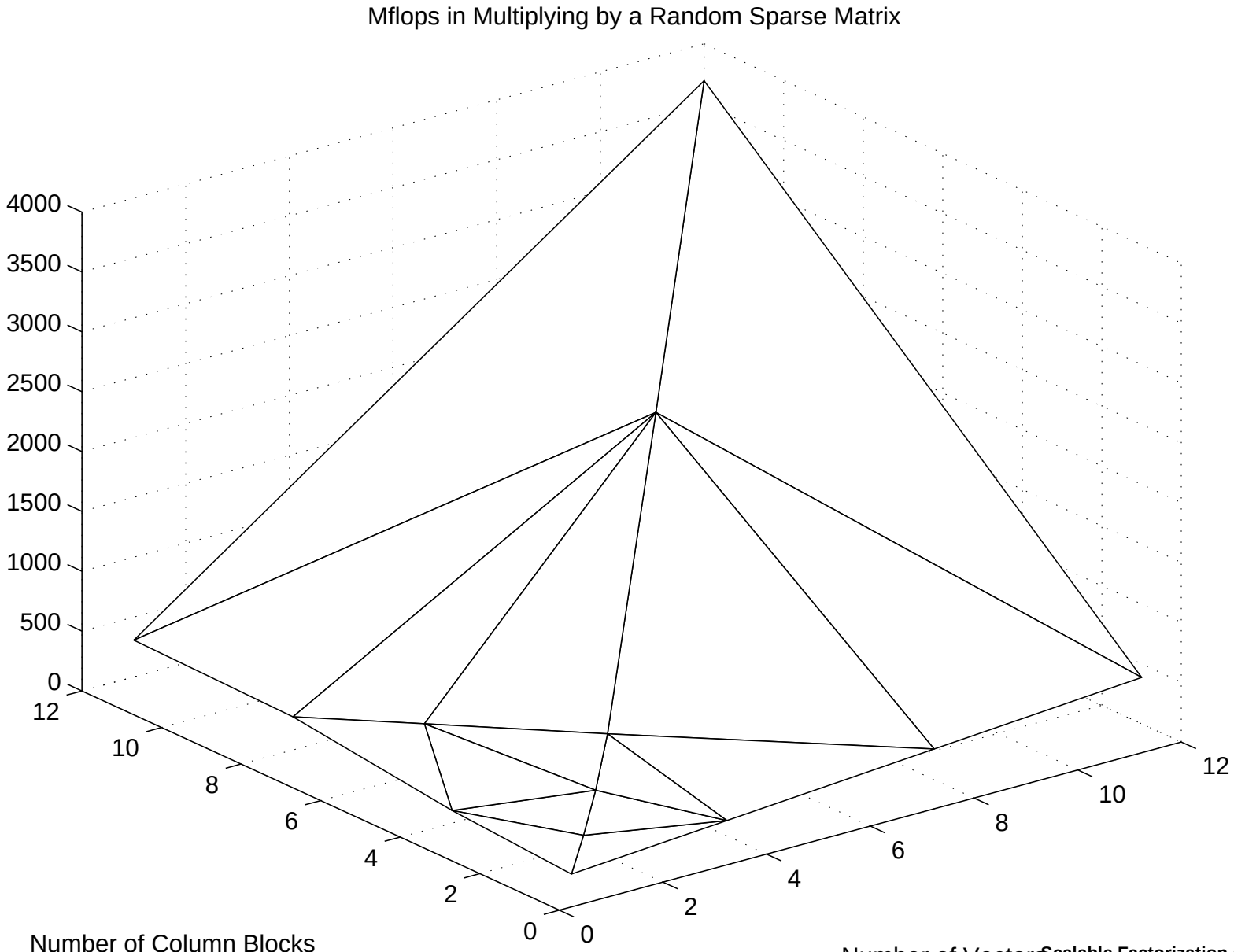
$$A = U_k \Sigma_k V_k^T + E_k$$

where where we choose $\|E_k\| \leq tol$. Typically in the sparse case, we may choose $tol = O(1.e-4/\|A\|$. The strategy is successful in terms of storage if k that achieves the tolerance is relatively small.

Sparse flops can be fast



Factor of 100 serial speedup



Dense Algorithms in Sparse Case

- All block update algorithms extend naturally to the sparse case. Because until the block update is done, you still are performing matrix vector multiplications by the original sparse matrix. So we can get the good stability properties of the dense algorithm ..
- A current improvement in the dense problem is in combining multiplications of $y^T A$ and Ax, thereby decreasing the number of transfers of A from RAM to cache. Similarly in the sparse case $y^T A$ and Ax are required for either the BLAS 2.5 - BLAS 3 algorithm or Lanczos methods.
- Grosser, Lang reduce to small band form by totally BLAS 3 operations, then carry on to bidiagonal via BLAS 1 operations. Sparse AX , A sparse X dense can be much faster than Ax. Some of the other tricks needed are block Householder transformations, similar

The Sparse Case

Golub and Van Loan, P. 498, 2nd Ed. “Unfortunately, if A is large and sparse, then we can expect large, dense submatrices to arise during the Householder bidiagonalization.” In context, this is a justification for using the unstable Lanczos procedure for bidiagonalizing sparse matrices, so they are probably not very startled to see an alternative approach.

Actually, the algorithms used to defer updates in the dense case, also allow sparse bidiagonalization without fill. That is to say, as long as we continue to defer matrix updates, still operating on the the original matrix, touching it only to do matrix vector multiplications, then the original matrix continues to be sparse.

Sparse Householder Bidiagonalization

Claim: if the Householder vectors require comparable storage to the original sparse A, Householder bidiagonalization is competitive to Lanczos in storage and overall computational costs.
For storage: If more than a few steps of Lanczos are used than the Lanczos vectors must be saved for use in re-orthogonalization. But this is as much storage as saving the Householder vectors.
For computational costs: the extra Householder flops are in dense computations, which are relatively fast compared to the predominant cost $y^T A$ and Ax.

All things being equal we should use the more stable algorithm?

Sparse Matrix Vector BLAS 2.5

For either sparse Householder bidiagonalization or for Lanczos bidiagonalization the main computational expense is in the multiplications $y^T A$ and Ax. Let's take the case of dense x, y, sparse A, A too large to fit in cache. Multiplication by sparse A is arranged so that either $y \leftarrow Ax$ or $w \leftarrow A^T y$ stream A from RAM. When A is banded or consists as in finite differences of several bands, then none of w, y, x suffer many cache misses in the multiplication. Combining the multiplications by A and A^T gives some speedup. For example, on a Xeon I saw (intel compiler -xW -tpp7 flags) 244 Mflops vs. 304 Mflops for the combined operations. (about 1/20 of the advertised peak speed). This trick should be orthogonal to other techniques such as getting dense subblocks, as for example from Vuduc, Demmel and Yelick .

x, y with many cache misses

For definiteness, take the artificially evil case that entries of A are randomly distributed, A strored in sparse row storage. That is, we read the entries of A as English speakers read text.

Then Ax is sparse ddots, relatively fast, which entail read misses in x.

For $y^T A$ we have no read misses, as the same entry $y(i)$ multiplies every entry in the ith row of A, but we update random entries in the product, so we have write misses.

For A in sparse row storage, Ax is typically faster than $y^t A$.

Not so much in coordinate storage, matrix ordered the same way (complete vector of row indices for A).

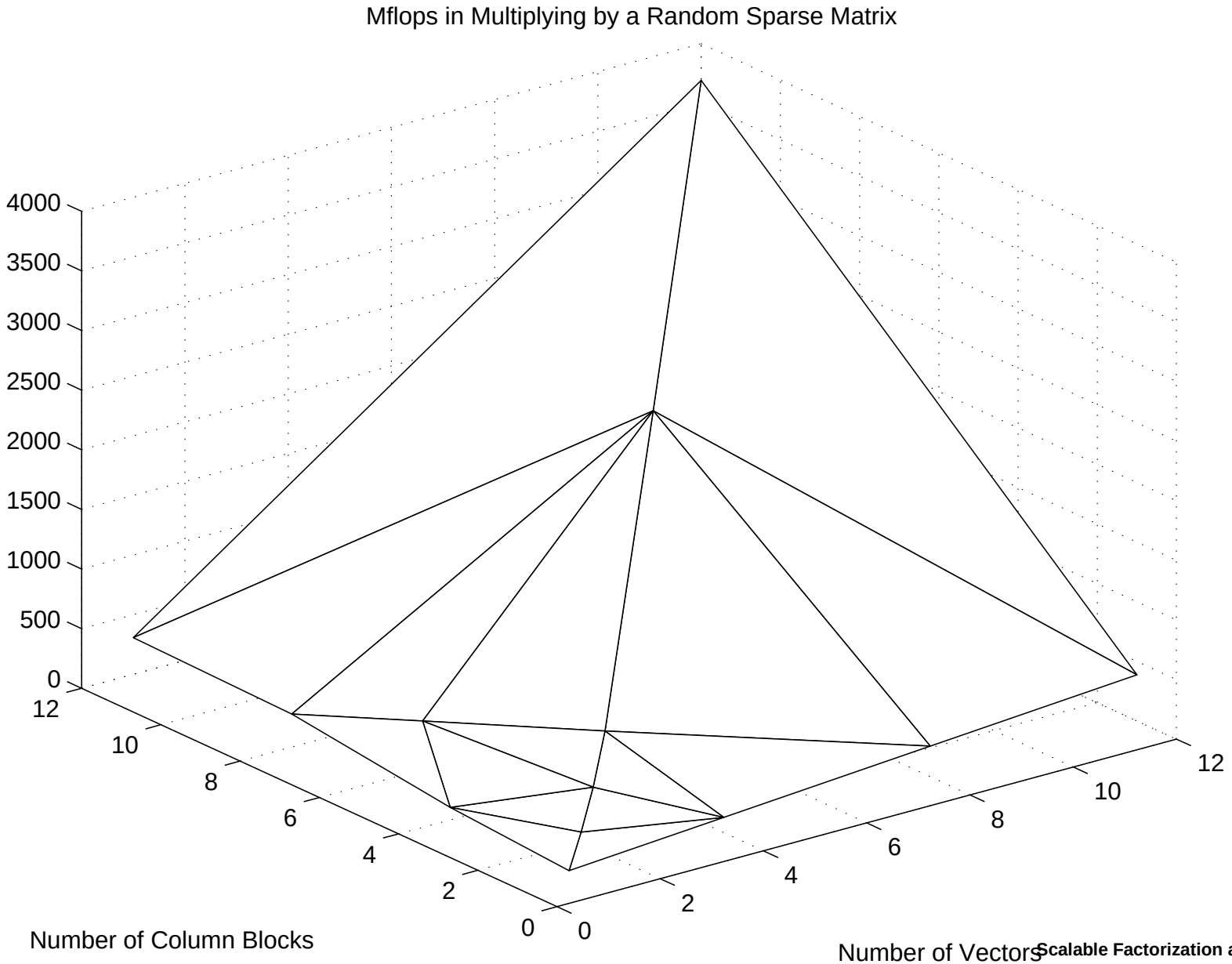
So column block A

One solution sometimes used is to keep two copies of A. If we want to save storage, we can instead use block storage. What I've been trying is to use row storage for column blocks. (like a reading a newspaper with multiple columns). The conversion routine first goes to coordinate storage, then sorts. In place quick sorts, so time goes like $O(nz\, log(nz))$.
This storage scheme appears to equalize the speeds of Ax and $y^t A$.

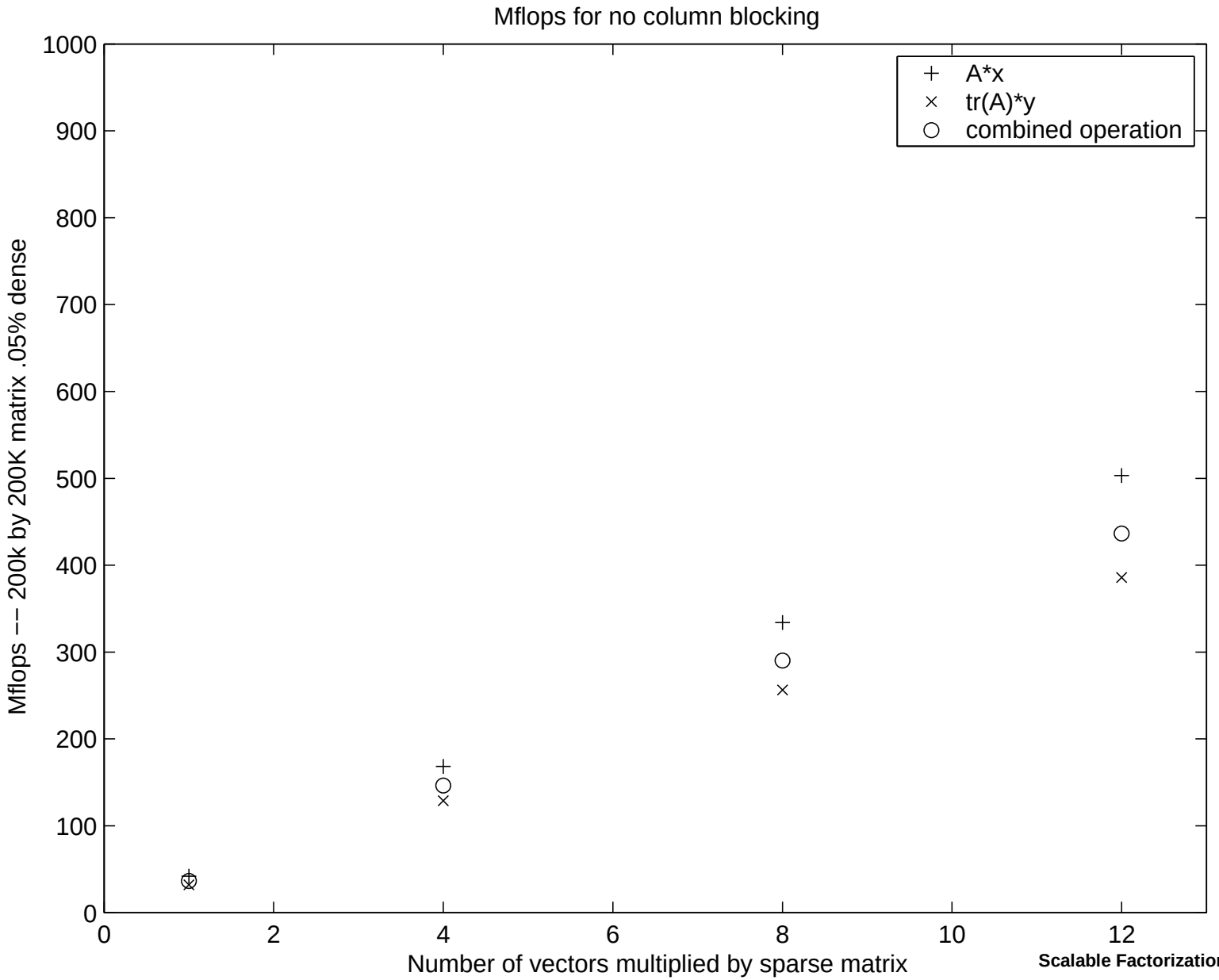
Multiplying by vector blocks

As any dense method that defers updates is really a sparse method (until the update), note that the Lang - Bischof - Sun type algorithms also extend to the sparse case.

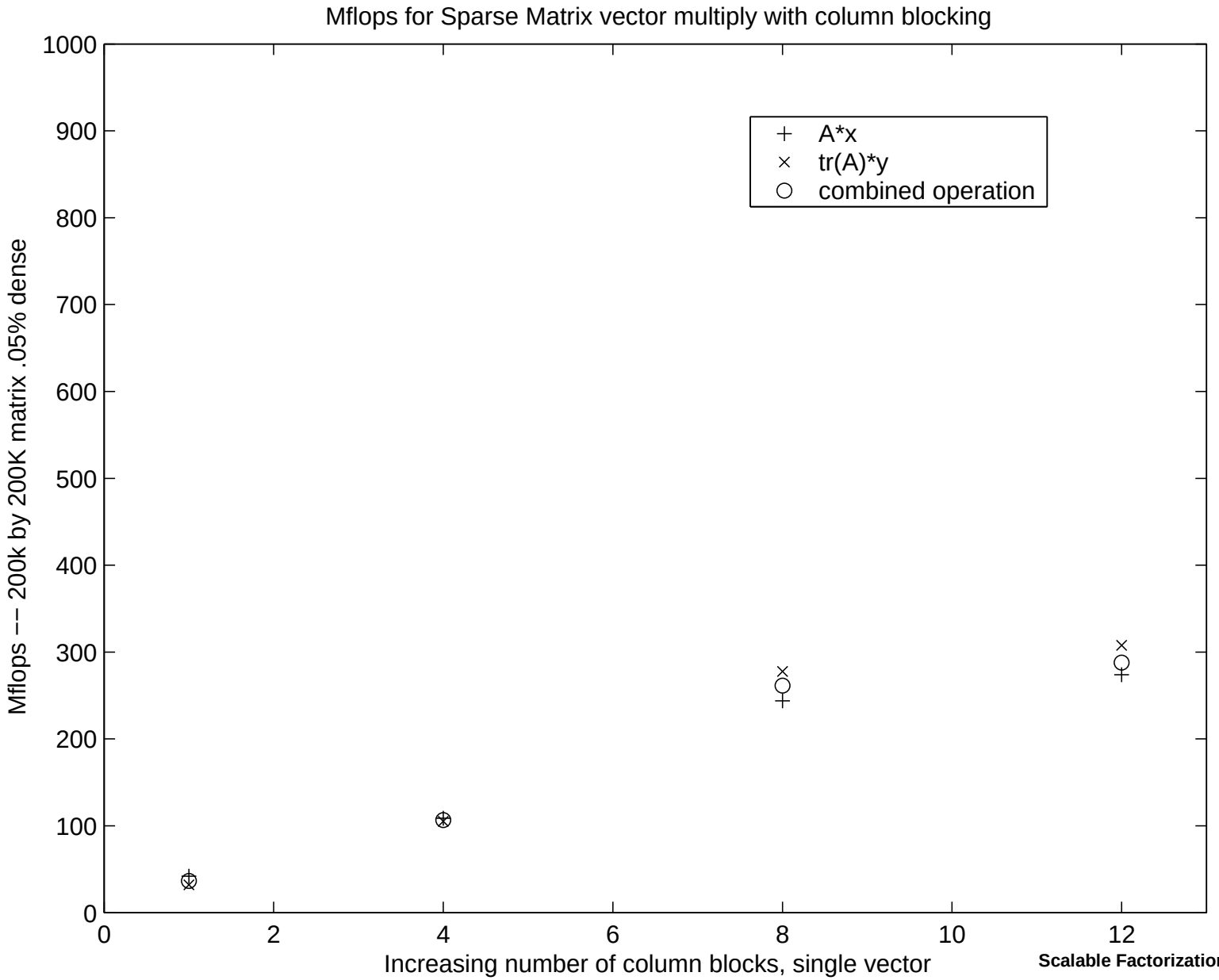
Yet Again


Mflops in Multiplying by a Random Sparse Matrix
4000
3500
3000
2500
2000
1500
1000
500
0
12
10
8
6
4
2
0
Number of Column Blocks
0
2
4
6
8
10
12
Number of Vectors

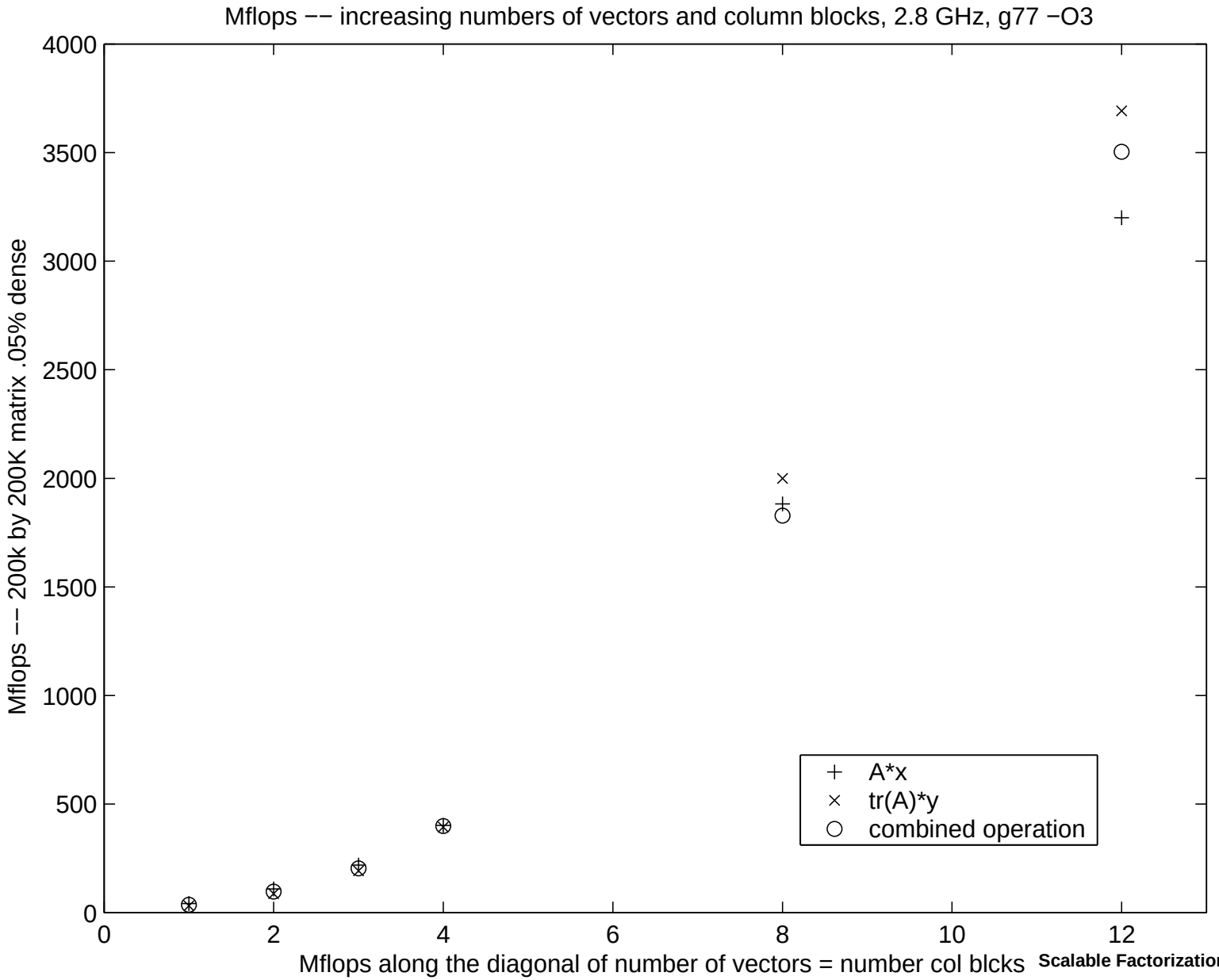
Multiplying by Multiple Vectors



Blocking the Sparse Matrix



Doing Both


Mflops −− increasing numbers of vectors and column blocks, 2.8 GHz, g77 −O3
Mflops −− 200k by 200K matrix .05% dense
Mflops along the diagonal of number of vectors = number col blcks
A*x
tr(A)*y
combined operation
0
500
1000
1500
2000
2500
3000
3500
4000
0
2
4
6
8
10
12

Blocking

Column Blocking (or more generally blocking) helps when the product and/or multiplicand vectors get too large for storage. It's cost is $O(log(n_z)n_z)$ and additional storage for storing a complete and column vectors. (About 33% more storage for matrix compared to compressed row storage). Multiplying by more vectors at a time "always" improves performance. Alas involves a rethink of whatever algorithm you're interested in and that can be tricky. You can introduce instability, e.g., block Lanczos. The easy write of the algorithm is in Matlab, but then there's an extra hurdle converting to C or Fortran.

I've got a Fortran Householder reduction of a banded matrix

to small band form.

Upper banded vs. block iteration

In block subspace iteration, singular values of the block form converge slowly and erratically. Wasteful in terms of number of floating point operations.

The banded singular values converge monotonically. Each new batch of flops improves the computation. Also there are some well-known error bounds.

Approximate A by $J_k = U_k B_k V_k^T$, I

The following results are adapted from Zhang, Zha, and Simon . Let B_k be the $m \times n$ bidiagonal matrix with diagonal entries $\alpha_1, \alpha_2, \ldots \alpha_k$ and superdiagonal $\beta_1, \beta_2, \ldots \beta_{k-1}$ If U_k and V_k are orthogonal, then the Frobenius norm $\|A - J_{k-1}\| = \omega_{k-1}$, then $\omega_k^2 + \alpha_k^2 + \beta_k^2 = \omega_{k-1}^2$.
More generally, denote $\|I - U_k^T U_k\|_2 = \eta(U_k)$ and $\|I - V_k^T V_k\|_2 = \eta(V_k)$. Then in floating point arithmetic

$$\omega_k^2+\alpha_k^2+\beta_k^2 = \omega_{k-1}^2+O(\|A\|^2\eta(U_k)(1+\eta(U_k)(1+m\epsilon))+O(\|A\|^2+\eta(U$$

Here ϵ is the largest number such that $fl(1+\epsilon) = 1$.

Approximate A by $J_k = U_k B_k V_k^T$, II

In exact arithmetic, $U_k A V_k^T = A_k$ where U_k and V_k are orthogonal. Due to the orthogonality of U_k and V_k

$$\|A_k\|_F = \|A\|_F \, .$$

Simplify the partitioning as

$$A_k = \left[\begin{array}{c|c} R_k & L_k \\ \hline 0 & \hat{A}_k \end{array}\right] \tag{1}$$

Approximate A by $J_k = U_k B_k V_k^T$, III

In practice, $\hat{A}_k$ is not computed as it would be dense and large and likely to overflow the RAM. Since

$$\|A\|_F^2 = \|A_k\|_F^2 = \|R_k\|_F^2 + \|L\|_F^2 + \|\hat{A}_k\|_F^2,$$

$$\|\hat{A}_k\|_F^2 = \|A\|_F^2 - \|R_k\|_F^2 - \|L\|_F^2$$

where all the quantities on the right hand side are easily computed.

Adapting LSQR

One of the better-known sparse least squares algorithms is LSQR. due to Paige and Saunders. It's interesting to translate LSQR, based on the Lanczos method of constructing locally biorthogonal, globally singular basis vectors. Would it actually help to have the orthogonal basis? I'm currently at the stage of constructing a Matlab alorithm.

SVD Summary

For a particular well-known algorithm (SVD), it appears we can extend the standard dense algorithm to get comparable efficiency (time per flop) in the sparse case.
Using block algorithms may enable other sparse algorithms to have similarly improved efficiency. Success would improve algorithm scalability and applicability to large data sets.
Nevertheless, improving sparse algorithm scalability is laborious and can only be justified for algorithms that will be widely used.
Except of course, that many fairly standard libraries already exist and can be easily ported. Sparse packages devloped by DOE include PETsc, ARPACK, SuperLU, and Trilinos (all have parallel versions).

Wny not a SVD for a sparse matrix?

- We want a sparse decomposition
- We want a positive decomposition
- We want a local decomposition
- Sometimes we don’t have a matrix (missing data for some entries).
- SVD may not scale well.
- SVD may not actually be competitive as a classification algorithm.

Scaling Classification Algorithms

Consider a set of compounds as a sparse matrix. Each row corresponds to a compound, each column to a descriptor. We want to discover likely compounds for further screening. And we want algorithms that can be applied to large data sets.

By using the standard set of modules developed for the R program, we can test (and allow users to test) a variety of possible algorithms.

Least Squares is Hard to Scale

Statisticians and Numerical Analysts like least squares and we want to put our eggs in that basket.
Thus far the ECCR project has not found least squares based algorithms to be effective – also they scale badly.
In R implementations, PCR appears to scale like $O(m^2 n)$ in time and often classifies no better than would a random choice. Of course, we can plug in ARPACK and the methods discussed above .. or other factorization methods.

But the best methods we have seen so far are KNN and tree based algorithms.

KNN

KNN (K nearest neighbors) is an unsupervised clustering algorithm. Given m compounds, we compute a dense $m \times m$ collection of distances d_{ij} of distances between compounds i and j. If there are an average of k nonzero descriptors per compounds, we have an algorithm requiring $m^2/2$ storage and $O(m^2 k)$ computations to get the distances d_{ij}. Also there are $O(m^2 \, log \, m)$ operations in sorting distances to find nearest neigbors.

Works fairly well, easy to parallelize. If we can do $m = 50K$ on one node, then with a hundred nodes, we could do $m = 500K$ (problem size goes as $p^{1/2}$ for p processors.)

Tree algorithms

Getting a best tree for m compounds is an NP complete problem, ... nevertheless we observe tree algorithms to be highly efficient.
Ms. Zhang has observed that the tree based algorithms, Rpart, Tree, and the ensemble method Random Forest all scale as m or $O(m\,log\,m)$ in time.

For these problems, we can scale to very large problems $m = O(10^6)$ on a single CPU or set of cores.

Multi-core Processors

Using the R packages has allowed the ECCR projects easy access to a number of algorithms.
Recently, we've found that some algorithms require 64 bit computations. Recompiling R for 64 bit is relatively easy in Linux, as 64 bit Intel CPUs have become standard this year.
A next architectural change is "multi-core". Dual core this year, quad core this summer, 8 cores next year ..
Codes do not automatically take advantage of the multi-core machines. Recompilation of codes is required.

Clocks To Core

core

Registers – 1 clock

L1 cache – 2-3 clocks

L2 Cache – 6-12 clocks

core

Registers – 1 clock

L1 cache – 2-3 clocks

L2 Cache – 6-12 clocks

More clocks depending on which core and register

Local RAM 20-200 clocks

Hard Drive 10^7 clocks

Distributed RAM 10^5 clocks

Compiler Generated Parallelism?

As with cache architectures and distributed memory computing, multi-core architectures have a long history in high performance computing.
Multi-core (shared memory architectures) have several standard libraries. The main current two are OpenMP and pThreads. pThreads libraries are available from C and C++ programs. OpenMP directives can be embedded also in Fortran codes.

Intel, IBM, and PGI compilers (and others) can automatically embed OpenMP directives in C or Fortran programs. The HPC experience is that the automatically inserted directives can be improved by manual editing, so that some parallel speed-up is achieved.

OpenMP?

Is OpenMP adequate for good parallelization? After editing the OpenMP pragmas, got speedup of two with four processors?
Workers on the LAPACK project have not found OpenMP directives adequate for good multi-core performance in the case of LU decomposition. They have turned to using the pThreads library.
The UPC project may be an easier route for programmers. It is also a standard.

Adapting R Modules to Multi-Core

Part of the continued ECCR project should be adapting R modules to a multi-core environment. The adapted algorithms will be widely useful.

References

- M. BERRY, *Large scale singular value computations*, Internat. J. Supercomputer Appl., 6:13-49, 1992.
- C. H. BISCHOF AND C. F. VAN LOAN, *The WY Representaion of Products of Householder Matrices*, SIAM J. Sci. Stat. Comput, 8:s2-s13, 1987.
- BLAS TECHNICAL FORUM,*www.netlib.org/utk/papers/blast-forum.html*, 1999.
- G. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, 3rd Ed., The Johns Hopkins University Press, Baltimore, 1996.

References-2

- B. GRÖSSER AND B. LANG, *Efficient Parallel Reduction to Bidiagonal Form*, Preprint BUGHW-SC 98/2 (Available from http://www.math.uni-wuppertal/)
- G.W. HOWELL, J.W. DEMMEL, C.T. FULTON, S. HAMMARLING,K. MARMOL *Cache Efficient Bidigaonlization Using BLAS 2.5 Operations*. (LAPACK Working Note # 174). http://www.netlib.org/lapack/lawnspdf/lawn174.pdf. Related work to appear in ACM TOMS.
- B. LANG, *Parallel reduction of banded matrices to bidiagonal form* Parallel Comput., 22 (1996), 1-18.
- C. PAIGE AND M. SAUNDERS. *An Algorithm for Sparse Linear Equations and Sparse Least Squares*. ACM Trans. on Math. Software, 8(1), 43–71, 1982.

References-3

- B. PARLETT AND I. DHILLON, *Fernando's solution to Wilkinson's problem: An application of double factorization* , Lin. Alg. Appl., 267:247–279, 1997.
- R. SCHREIBER AND C. F. VAN LOAN, *A Storage-Efficient WY Representation for Products of Householder Transformations*, SIAM Scientific and Statistical Computing, 10:53-57, 1989.
- R. VUDUC, J. DEMMEL, K. YELICK, *OSKI: A library of automatically tuned sparse matrix kernels*, Proceedings of SciDAC 2005, Journal of Physics: Conferences Series, Jun 2005, http://bebop.cs.berkeley.edu/#pubs

References-4

- Z. ZHANG, H. ZHA, AND H. SIMON, *Low-rank Approximations with Sparse Factors I: Basic Algorithms and Error Analysis*, SIAM Journal of Matrix Analysis and Applications, 23, pp. 706-727,2002.