CS 219: Sparse matrix algorithms: Homework 2

Assigned April 9, 2013

Due by class time Monday, April 16

Problem 1. A *complete subgraph* of an undirected graph is a subgraph in which every pair of vertices is joined by an edge.

Let G be the graph of the n-vertex model problem, that is, a k-by-k grid graph with $n = k^2$ vertices. Prove that there is some constant $c > 0$, independent of k or n, such that for *every* elimination ordering on G, the filled graph G^+ contains a complete subgraph with at least $c\sqrt{n}$ vertices.

Hint: Suppose you're given an ordering for the vertices of G. Think of playing the graph game in the given order, and consider the first time that you've either marked all the vertices in any single row of the entire grid or else marked all the vertices in any single column.

Problem 2. Let A and B be two n-by-n matrices. Prove that the number of nonzero scalar multiplications required to compute $C = AB$ is (using Matlab notation)

$$\sum_{i=1}^{n} \texttt{nnz}(\texttt{A}(:,\texttt{i})) * \texttt{nnz}(\texttt{B}(\texttt{i},:)).$$

Problem 3. (See Davis problem 2.20.) Nobody knows any way to predict the exact number of nonzeros in the product $C = AB$ that is asymptotically faster than actually computing C. (This is very different from Cholesky factorization, where it's much faster to compute the number of nonzeros in the Cholesky factor than to compute the factor itself.) Therefore, any sparse matrix multiplication routine has to include some kind of trial-and-error way of allocating memory for its results. The purpose of this problem is to experiment with different ways of doing this.

You will need a sparse matrix multiplication routine to start with. I recommend that you use `cs_multiply` from the Davis book, and modify it to do your experiments. Use a Matlab mex-file interface for testing. The Matlab interface to the Davis code is described in section 10.3, and is on the SuiteSparse web site along with the rest of the book's code.

Experiment with the three (optionally four) methods below on the following two classes of n-by-n matrices, for various values of n including the largest you can fit in your machine's memory: first, uniform random matrices created by Matlab's `sprand(n,n,8/n)`; second, power-law matrices created by the Matlab routine `rmat(k)` (see the course web page). Note that the dimension of an `rmat(k)` matrix is actually $n = 2^k$. Use Matlab's `tic` and `toc` to get wall-clock times for the various methods, and use the result of problem (2) above to get the number of flops for each multiplication.

Compute the number of flops per second for each method on each matrix, and make a scatter plot in Matlab showing your results. Can you beat Matlab's running time?

Turn in all your code, your plot, and also a Matlab transcript of the session that creates the plot and verifies that your output matrices agree with Matlab's.

First method: Guess and expand. This is the method both `cs_multiply` and the Matlab built-in matrix multiplication use. Guess an initial number of nonzeros to allocate for C (`cs_multiply` uses `nnz(A)+nnz(B)`), and then if you run out of space, allocate a larger space and copy the part of C you've computed so far into it. `cs_multiply` approximately doubles its guess each time.

Second method: Compute twice. This is the method suggested in Davis problem 2.20. Do the whole computation of $C = AB$ twice. The first time through, don't allocate any space for C; after computing each column of C in the SPA, discard it, but keep a count of the total number of nonzeros. The second time, allocate exactly the right amount of space for C before you start.

Third method: Cheat. Let the user pass in an upper bound on the number of nonzeros in C, and just fail if you exceed it. This isn't a good idea in practice, but you should time this method (using a big enough bound) just to see how much time the memory reallocation is costing.

Fourth method: Probabilistic estimate (optional extra credit, or part of a possible term project). If you're interested in digging deeper into this, you can read Edith Cohen's paper "Structure prediction and computation of sparse matrix products," *J. Combinatorial Optimization* 2:307–332, 1998, also at `http://www.springerlink.com/content/p328542122022748/`, which gives a fast probabilistic way of getting a good estimate of `nnz(A*B)`. The idea is to look at a graph whose vertices represent (separately) the rows and columns of A and B, and estimate the number of column vertices of B that have paths to each row vertex of A. Cohen gives a probabilistic estimate that uses repeated "rounds," each of which looks a lot like multiplying A and B by a dense vector. The more rounds, the better the estimate.