Sparse Data Structures for Weighted Bipartite Matching

E. Jason Riedy* Dr. James Demmel
UC Berkeley

October 31, 2003

1 Introduction

Inspired by the success of blocking to improve the performance of algorithms for sparse matrix vector multiplication [5] and sparse direct factorization [1], we explore the benefits of blocking in related sparse graph algorithms. A natural question is whether the benefits of local blocking extend to other sparse graph algorithms. Here we examine algorithms for finding a maximum-weight complete matching in a sparse bipartite graph, otherwise known as the linear sum assignment problem, or LSAP. Like sparse matrix-vector products, algorithms for LSAP combine data from two sources. But rather than computing a dot product, the use a complicated max reduction or depth-first search (DFS).

Maximum weight matchings find uses in sparse matrix algorithms which could otherwise benefit from blocked matrix representations. A matching provides statically chosen pivots for parallel sparse *LU* factorization [6]. The *LU* factorization is followed by matrix-vector products in iterative refinement, so we would like take a blocked matrix structure as input. Preliminary work shows that the blocked representations do not decrease performance significantly although they do increase the number of operations performed, and so we can use a matrix data structure that improves the iterative refinement performance.

2 Weighted Matching and the Auction Algorithm

The square matching problem (square LSAP) is to maximize the trace of $B^T X$ overall permutation matrices X:

$$\textbf{Primal:}\quad \begin{array}{ll} \underset{X \in \Re^{n\times n}}{\text{maximize}} & \operatorname{Tr} B^T X \\ \text{subject to} & X 1_c = 1_r, \\ & X^T 1_r = 1_c, \text{ and} \\ & X \geq 0; \text{ and} \end{array} \qquad \textbf{Dual:}\quad \underset{p_c}{\text{minimize}} \quad 1_r^T p_c + \sum_j \max_{i \in \mathcal{R}} (B(i,j) - p_c(j))$$

Column-vectors 1_r and 1_c are unit entry vectors of length n. Dual variable p_c is a column vector of length n. Subscripts are used for descriptive purposes; $p_c(j)$ is the "price" for column j of B. The interesting optimality condition for the LSAP is its complementary slackness (CS) condition $x(i,j) \cdot (p_c(j) + \pi_r(i) - B(i,j)) = 0$ for all i, j where the "profit" $\pi_r(i) = \max_{j \in \mathcal{C}} B(i,j) - p_c(j)$. An edge can be in the matching only when its price is balanced with its profit.

Our applications have a sparse B. However, entries not explicitly stored in B are considered to be $-\infty$ rather than zero. A finite maximum value means all the entries on the diagonal of $X^T B$ are finite entries of B. We derive B from a sparse matrix A by either taking, element-wise, magnitudes to maximize the diagonal's sum or the log of magnitudes for the product.

Algorithms for solving the LSAP add edges to the matching in X according to current values of the dual variable p_c. In essence, they are primal-infeasible / dual-feasible optimization algorithms. Many algorithms exist for solving LSAPs; see [3]. We focus on two: an explicit depth-first search (DFS) algorithm implemented in MC64 [4] and Bertsekas's auction algorithm[2]. For a given entry range, both have asymptotic bounds of $O(n\tau \log n)$, where τ is the number of explicitly stored entries, but experimental running times grow much more slowly. Explicit DFS algorithms are traditionally used for sparse problems. The explicit search has little to gain from blocked representations; the search jumps from row to row rather than scanning entire rows.

Bertsekas's auction algorithm performs an *implicit* DFS. Each currently unmatched row i places a bid for the column of greatest profit $j = \operatorname{indmax}_k B(i,k) - p_c(k)$. The bid offers to increase the price to satisfy a relaxed CS condition

*DOE SCIDAC: Grant No. DE-FC02-01ER25478

$x(i,j) \cdot (p_c(j) + \pi_r(i) - B(i,j)) \leq \mu$. The relaxation ensures progress and produces an LSAP solution within an additive factor of $\mu(n-1)$ of the optimum. Each row is scanned in its entirety, and the algorithm allows great flexibility in the order of its basic operations (finding a bid, placing a bid, outbidding an existing match). The literature includes many algorithmic optimizations for auctions [2], but the only one found useful on the test problems was adaptively scaling μ to its final value.

3 Selected Results

On average, our adaptive scaling auction implementation in C++ performs as well as MC64 (Fortran77). Each travels a different path through the optimization space, but both arrive at matchings with the same weight. Relative performance varies from problem to problem.

The following table provides a few selected results somewhat characteristic of a larger body of test matrices[1]. The implementations were run on nodes of `seaborg.nersc.gov`, with 375MHz IBM POWER3 processors and 8MB of L2 cache. Times are given for MC64, the auction on a compressed sparse row representation, and the auction on a compressed sparse block-row representation. Elements have been transformed by $b(i,j) = \lfloor \log_2 a(i,j) \rfloor$ and then by shifting the finite entries to be positive.

Name	n	# entries	MC64 time (s)	Unblocked Auction time (s)	Blocked 2×2 Auction time (s)
e40r5000	17281	553562	0.83	1.04 (1.2× MC64)	1.05 (1.3×)
av41092	41092	1683902	6.31	0.67 (.11×)	0.72 (.12×)
kim1	38415	933195	0.079	0.056 (.71×)	0.059 (.75×)

For each of the above matrices, forming 2×2 dense blocks to encompass the elements of each element increases the matrix fill by about 20%. However, the total memory number of words loaded by the auction algorithm increases by at most 2%.

The block sizes larger than 2×2, preferred for matrix-vector multiplications, increase the cost of auctions from 25% to 50%. This increase is somewhat expected. Matrix-vector products amortize overheads while scanning entire block rows, but the matcher only scans one row within each block. Also, iterating over 1×1 blocks imposes about a 5% overhead when compared to the unblocked form, and the compiler used is not scheduling the block loops well.

On a handful of matrices, using 1×2 or 1×3 blocks improves performance by 10% to 15%. This leads us to believe that more tuning work or modified data structures could decrease the general overheads

Besides preprocessing for sparse *LU* factorization, matchings are also used as core tests in travelling salesman algorithms,for specialized database queries, and in other roles. Overall, if the matching is only one piece of a larger algorithm, it appears that optimizing sparse matrix data structures for other, more time-consuming operations could improve the algorithm's performance.

References

[1] Patrick Amestoy, Iain Duff, Jean-Yves L'Excellent, and Xiaoye Li. Analysis and comparison of two general sparse solvers for distributed memory computers. Technical Report LBNL–45992, Lawrence Berkeley National Laboratory, July 2000.

[2] Dimitri Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1:7–66, 1992.

[3] Rainer Burkard and Erand Çela. Linear assignment problems and extensions. In Panos M. Pardalos and Ding-Zhu Du, editors, *Handbook of Combinatorial Optimization - Supplement Volume A*, pages 75–149. Kluwer Academic Publishers, October 1999.

[4] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[5] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 2003. (to appear).

[6] Xiaoye S. Li and James W. Demmel. Superlu-dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), June 2003.

[1] Matrices acquired from the UF collection (`http://www.cise.ufl.edu/research/sparse/matrices/`) and the MatrixMarket (`http://math.nist.gov/MatrixMarket/`).