High–performance graph algorithms from parallel sparse matrices

John R. Gilbert[1][*], Steven Reinhardt[2], and Viral Shah[1]

[1] UC Santa Barbara, ({`gilbert,viral`}`@cs.ucsb.edu`)[†]
[2] Silicon Graphics Inc., (`spr@sgi.com`)

Abstract. Large–scale computation on graphs and other discrete structures is becoming increasingly important in many applications, including computational biology, web search, and knowledge discovery. High–performance combinatorial computing is an infant field, in sharp contrast with numerical scientific computing.

We argue that many of the tools of high-performance numerical computing – in particular, parallel algorithms and data structures for computation with sparse matrices – can form the nucleus of a robust infrastructure for parallel computing on graphs. We demonstrate this with a graph analysis benchmark using the sparse matrix infrastructure in STARP, our parallel dialect of the matlab programming language.

1 Sparse matrices and graphs

Sparse matrix computations allow structured representation of irregular data structures and decompositions, and irregular access patterns in parallel applications. Every sparse matrix problem is a graph problem and every graph problem is a sparse matrix problem. We reiterate the basic principles that have to be considered while designing sparse matrix data structures and algorithms [2], which also result in efficient operations on graphs.

1. Storage for a sparse matrix should be $\theta(max(n, nnz))$
2. Operations on sparse matrices should take time proportional to the size of the data accessed and the number of nonzero arithmetic operations.

A graph consists of a set of vertices V, connected by edges E. A graph can then be specified by tuples (u, v, w) – this means that there exists a directed edge of weight w from vertex u to vertex v. This is the same as a nonzero w at location (u, v) in a sparse matrix. According to principle 1, the storage required is $\theta(|V| + |E|)$. An undirected graph has edges in both directions resulting in a corresponding symmetric sparse matrix. Special properties in graphs typically translate into a richer structure in the corresponding sparse matrix.

Sparse matrix operation	Graph operation
`G = sparse (U, V, W)`	Construct a graph from an edge list
`[U, V, W] = find (G)`	Obtain the edge list from a graph
`vtxdeg = sum (spones(G))`	Get vertex degrees for undirected graphs
`indeg = sum (spones(G))`	Indegrees for directed graphs
`outdeg = sum (spones(G), 2)`	Outdegrees for directed graphs
`N = G(i, :)`	Find all neighbours of vertex i
`Gsub = G(p, p)`	Extract a subgraph of G
`G(I, J) = 0`	Delete graph edges
`G(I, J) = W`	Add or modify graph edges
`G = G(label, label)`	Permute or relabel the vertices of a graph
`reach = G * start`	Breadth first search step

Table 1. Correspondence between some sparse matrix and graph operations.

A correspondence between sparse matrix operations and graph operations is listed in Table 1. Consider breadth first search (BFS). A BFS can be performed by multiplying a sparse matrix G with a sparse vector x. The simplest case is doing a BFS starting from vertex i. In this case, we set $x(i) = 1$, all other elements being zeros. $x = G * x$ simply picks out column i of G which contains the neighbours of vertex i. We can also do several independent BFS searches simultaneously, by using sparse matrix sparse matrix multiplication [6]. A detailed analaysis of sequential [2] and parallel [7] sparse matrix data structures and algorithms is available in earlier work by the authors.

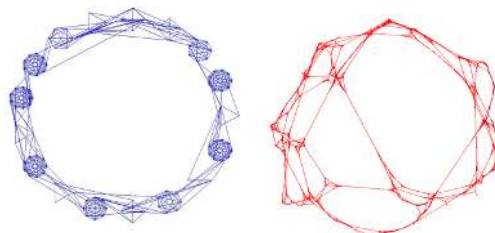## 2 An example: SSCA #2 graph analysis benchmark

Fig. 1. (a) SSCA #2 graph (b) SSCA #2 graph plotted with Fiedler co–ordinates.

The SSCAs (Scalable Synthetic Compact Applications) are a set of benchmarks designed to complement existing benchmarks such as the HPL and the NAS parallel benchmarks. Specifically, SSCA #2 [1] is a compact application that has multiple kernels accessing a single data structure representing a directed multigraph with weighted edges. The data generator generates an edge list in random order for a multigraph of sparsely connected cliques as shown in Figure 1. The four kernels are as follows:

1. Kernel 1: Create a data structure for further kernels.

2. Kernel 2: Search graph for a maximum weight edge.
3. Kernel 3: Perform breadth first searches from a set of start vertices.
4. Kernel 4: Recover the underlying clique structure from the undirected graph.
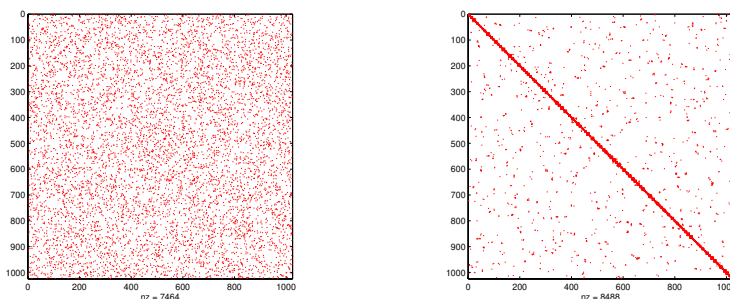

Fig. 2. (a) Input graph (b) After clustering, clusters are along the diagonal.

We implement only the integer version of the benchmark. The first three kernels are easily implemented using the infrastructure described in the earlier section. We focus our attention on kernel 4, which can be considered to be a partitioning problem or a clustering problem. We have several implementations of kernel 4 based on spectral partitioning (Figure 1) and seed growing techniques (Figure 2). The seed growing implementations scale better than the spectral methods, as expected. We will demonstrate how we use the infrastructure described above to implement kernel 4 in a few lines of matlab.

```
J = sparse(v,1:nseeds,1,n,nseeds);  % Sparse matrix, 1 seed per column.
J = G*J;       % Vertices reachable with 1 hop.
J = J + G*J;   % Vertices reachable with 1 or 2 hops.
J = J > 1;     % Vertices reachable with at least 2 paths of 1 or 2 hops.
```

Fig. 3. Breadth first parallel clustering by seed growing.

Our implementation starts out by picking out a set of seeds from the graph. These seeds may be chosen such that they form an independent set. One way to do this is to run one round of Luby's algorithm [4], which is part of our toolbox, or simply pick them randomly. Then, we grow the seeds so that each seed claims all vertices reachable by at least 2 paths of length 1 or 2. Since there may be some overlap, we use each vertex attaches itself to a cluster using a 'peer pressure' algorithm. Figure 3 describes the 'seed growing' and Figure 4 describes the 'peer pressure' algorithm.

Our implementation of SSCA #2 uses STARP [3], which is a parallel implementation of the matlab language with global array semantics. We are in the

```
% Each vertex chooses a random neighbour in the independent set.
neighbours = G * sparse(IndepSet, IndepSet, 1, n, n);
R = sprand (neighbours);
[ignore, vote] = max (R, [], 2);

% Collect neighbour votes and join the most popular cluster.
[I, J] = find (G);
S = sparse (IndepSet, vote(J), 1, n, n);
[ignore, cluster] = max (S, [], 2);
```

Fig. 4. Parallel clustering by peer pressure

process of porting it to MIT Lincoln Labs' PMATLAB [8] and the Mathworks Parallel MATLAB [5], when available.

3 Concluding remarks

We have run the full SSCA #2 benchmark in STARP on graphs with $2^{27} = 134$ million vertices on the SGI Altix, and we observe good scaling as we vary the problem size and number of processors. We have also manipulated graphs with 400 millon vertices and 4 billion edges. Note that the code in Figure 3 and Figure 4 are not pseudocode, but actual code from our implementation. The entire kernel 4 implementation is on the order of 100 lines of code. Although the code fragments look very simple and structured, they are anything but. All operations are on sparse matrices, resulting in highly irregular communication patterns on irregular data structures. We conclude that sparse matrices lend themselves as a natural data structure for operations on large graphs.

References

1. D. A. Bader, J. R. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. Hpcs ssca #2, graph analaysis. 2006.
2. J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIMAX*, 13(1):333–356, 1992.
3. P. Husbands and C. Isbell. MATLAB*P: A tool for interactive supercomputing. *SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
4. Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.
5. C. B. Moler. Parallel matlab. *Householder Symposium on Numerical Algebra*, 2005.
6. Christopher Robertson. Sparse parallel matrix multiplication. *M.S. Project, Department of Computer Science, UCSB*, 2005.
7. Viral Shah and John R. Gilbert. Sparse matrices in matlab*p: Design and implementation. In *HiPC*, pages 144–155, 2004.
8. Nadya Travinin and Jeremy Kepner. pMatlab parallel matlab library. *Submitted to International Journal of High Performance Computing Applications*, 2006.