Distributed Sparse Matrices for Very High Level Languages

John R. Gilbert
Department of Computer Science
UC Santa Barbara

Steve Reinhardt
Interactive Supercomputing

Viral B. Shah
Department of Computer Science
UC Santa Barbara

and

Interactive Supercomputing

November 27, 2007

Abstract

Sparse matrices are first class objects in many VHLLs (very high level languages) used for scientific computing. They are a basic building block for various numerical and combinatorial algorithms. Parallel computing is becoming ubiquitous, specifically due to the advent of multi-core architectures. As existing VHLLs are adapted to emerging architectures, and new ones are conceived, one must rethink tradeoffs in language design. We describe the design and implementation of a sparse matrix infrastructure for Star-P, a parallel implementation of the MATLAB® programming language. We demonstrate the versatility of our infrastructure by using it to implement a benchmark that creates and manipulates large graphs. Our design is by no means specific to Star-P— we hope it will influence the design of sparse matrix infrastructures in other languages.
Contents

1 Introduction 3

2 Sparse matrices: A user’s view 3

3 Data structures and storage 4

4 Operations on distributed sparse matrices 6
   4.1 Constructors ................................................. 6
   4.2 Element-wise matrix arithmetic ............................... 6
   4.3 Matrix multiplication ........................................ 7
      4.3.1 Sparse matrix dense vector multiplication ............. 7
      4.3.2 Sparse matrix sparse matrix multiplication ............. 7
   4.4 Sparse matrix indexing, assignment, and concatenation .... 10
   4.5 Sparse matrix transpose ..................................... 11
   4.6 Direct solvers for sparse linear systems .................... 11
   4.7 Iterative solvers for sparse linear systems ................. 12
   4.8 Eigenvalues and singular values .............................. 13
   4.9 Visualization of sparse matrices ............................ 13

5 SSCA #2 graph analysis benchmark 14
   5.1 Scalable data generator .................................... 15
   5.2 Kernel 1 ................................................... 15
   5.3 Kernel 2 ................................................... 16
   5.4 Kernel 3 ................................................... 16
   5.5 Kernel 4 ................................................... 17
   5.6 Visualization of large graphs ............................... 19
   5.7 Experimental Results ....................................... 20

6 Looking forward: A next generation parallel sparse library 22

7 Conclusion 24
1 Introduction

Two trends have emerged of late in scientific computing. The first one is the adoption of high level interactive programming environments such as MATLAB® [27], R [22] and Python [34]. This is largely due to diverse communities in physical sciences, engineering and social sciences using simulations to supplement results from theory and experiments.

Computations on graphs combined with numerical simulation is the other trend in scientific computing. High performance applications in data mining, computational biology, and multi-scale modeling, among others, combine numerics and combinatorics in a variety of ways. Relationships between individual elements in complex systems are typically modeled as graphs. Links between webpages, chemical bonds in complex molecules, and connectivity in social networks are some examples of such relationships.

Scientific programmers want to combine numerical and combinatorial techniques in interactive VHLLs, while keeping up with the increasing ubiquity of parallel computing. A distributed sparse matrix infrastructure is one way to address these challenges. We describe the design and implementation of distributed sparse matrices in STAR-P, a parallel implementation of the MATLAB® programming language.

Sparse matrix computations allow structured representation of irregular data structures and access patterns in parallel applications. Sparse matrices are also a convenient way to represent graphs. Since sparse matrices are first class citizens in modern programming languages for scientific computing, it is natural to take advantage of the duality between sparse matrices and graphs to develop a unified infrastructure for numerical and combinatorial computing.

The distributed sparse matrix implementation in STAR-P provides a set of well-tested primitives with which graph algorithms can be built. Parallelism is derived from operations on parallel sparse matrices. The efficiency of our graph algorithms depends upon the efficiency of the underlying sparse matrix infrastructure.

We restrict our discussion to the design and implementation of the sparse matrix infrastructure in STAR-P, trade-offs made, and lessons learnt. We also describe our implementation of a graph analysis benchmark, using Gilbert, Reinhardt and Shah’s “Graph and Pattern Discovery Toolbox (GAPDT)” [16, 29, 32, 33].

2 Sparse matrices: A user’s view

The basic design of STAR-P and operations on dense matrices have been discussed in earlier work [8, 20, 21]. In addition to MATLAB®’s sparse and dense matrices, STAR-P provides support for distributed sparse (dsparse) and distributed dense (ddense) matrices.

The $p$ operator provides for parallelism in STAR-P. For example, a random parallel dense matrix (ddense) distributed by rows across processors is created as follows:

\[
\text{>> } A = \text{rand} \left(1e4*p, 1e4\right)
\]
Similarly, a random parallel sparse matrix (dsparse) also distributed across processors by rows is created as follows: (The third argument specifies the density of non-zeros):

\[ S = \text{sprand}(1e6*p, 1e6, 1/1e6) \]

We use the overloading facilities in MATLAB\textsuperscript{®} to define a \textit{dsparse} object. The STAR-P language requires that almost all (meaningful) operations that can be performed in MATLAB\textsuperscript{®} be possible with STAR-P. Our implementation provides a working basis, but is not quite a drop-in replacement for existing MATLAB\textsuperscript{®} programs.

STAR-P achieves parallelism through polymorphism. Operations on ddense matrices produce ddense matrices. But, once initiated, sparsity propagates. Operations on dsparse matrices produce dsparse matrices. An operation on a mixture of dsparse and ddense matrices produces a dsparse matrix unless the operator destroys sparsity. The user can explicitly convert a ddense matrix to a dsparse matrix using \texttt{sparse(A)}. Similarly a dsparse matrix can be converted to a ddense matrix using \texttt{full(S)}. A dsparse matrix can also be converted into a frontend sparse matrix using \texttt{ppfront(S)}.

### 3 Data structures and storage

It is true in MATLAB\textsuperscript{®}, as well as in STAR-P, that many key operations are provided by public domain software (linear algebra, solvers, fft, etc.). Apart from simple operations such as array arithmetic, MATLAB\textsuperscript{®} allows matrix multiplication, array indexing, array assignment and concatenation of arrays, among other things. These operations form extremely powerful primitives upon which other functions, toolboxes, and libraries are built. The challenge in the implementation lies in selecting the right data structures and algorithms that implement all operations efficiently, allowing them to be combined in any number of ways.

Compressed row and column data structures have been shown to be efficient for sparse linear algebra \cite{18}. MATLAB\textsuperscript{®} stores sparse matrices on a single processor in a Compressed Sparse Column (CSC) data structure \cite{15}. The STAR-P language allows ddense matrices to be distributed by block rows or block columns \cite{8,20}. Our implementation supports only the block row distribution for dsparse matrices. This is a design choice to prevent the combinatorial explosion of argument types. Block layout by rows makes the Compressed Sparse Row data structure a logical choice to store the sparse matrix slice on each processor. The choice to use a block row layout was not arbitrary, but the reasoning was as follows:

- The iterative methods community largely uses row based storage. Since we believe that iterative methods will be the methods of choice for large sparse matrices, we want to ensure maximum compatibility with existing libraries.

- A row based data structure also allows efficient implementation of “matvec” (sparse matrix dense vector product), the workhorse of several iterative methods such as Conjugate Gradient and Generalized Minimal Residual.
Figure 1: The matrix is shown in its dense representation on the left, and its compressed sparse rows (CSR) representation on the right. In the CSR data structure, non-zeros are stored in three vectors. Two vectors of length $nnz$ store the non-zero elements and their column indices. A vector of row pointers marks the beginning of each new row in the non-zero and column index vectors.

For the expert user, storing sparse matrices by rows instead of by columns changes the programming model. For instance, high performance sparse matrix codes in MATLAB® are often carefully written so that all accesses into sparse matrices are by columns. When run in STAR-P, such codes may display different performance characteristics, since dsparse matrices are stored by rows. This may be considered by some to be a negative impact of our decision to use compressed sparse rows instead of compressed sparse columns.

This boils down to a question of design goals. We set out to design a high performance parallel sparse matrix infrastructure, and concluded that row based storage was the way to go. Had our goal been to ensure maximum performance on existing MATLAB® codes, we might have chosen a column based storage. Given all that we have learnt from our implementation, we might reconsider this decision in the light of 1D distributions. However, it is much more likely that a redesign will consider a 2D distribution, to allow scaling to thousands of processors. We describe some of these issues in detail in the “Looking forward” section.

The CSR data structure stores whole rows contiguously in a single array on each processor. If a processor has $nnz$ non-zeros, CSR uses an array of length $nnz$ to store the non-zeros and another array of length $nnz$ to store column indices, as shown in Figure 1. Row boundaries are specified by an array of length $m + 1$, where $m$ is the number of rows on that processor.

Using double precision floating point values for the non-zeros on 32-bit architectures, an $m \times n$ real sparse matrix with $nnz$ non-zero elements uses $12nnz + 4(m + 1)$ bytes of memory. On 64-bit architectures, it uses $16nnz + 8(m + 1)$ bytes. STAR-P supports complex sparse matrices as well. In the 32-bit case, the storage required is $20nnz + 4(m + 1)$ bytes, while it is $24nnz + 8(m + 1)$ bytes on 64-bit architectures.

Consider the example described earlier. A sparse matrix with a million rows and columns, with a density of approximately one nonzero per row or column. The memory required for a dense representation would be $10^6 \times 10^6 \times 8$ bytes = 8 terabytes. The CSR data structure, on the other hand, would use $16 \times 10^6 + 8 \times 10^6$ bytes = 24 megabytes.
4 Operations on distributed sparse matrices

The design of sparse matrix algorithms in STAR-P follows the same design principles as in MATLAB® [15].

1. Storage required for a sparse matrix should be $O(nnz)$, proportional to the number of non-zero elements.

2. Running time for a sparse matrix algorithm should be $O(flops)$. It should be proportional to the number of floating point operations required to obtain the result.

The data structure described in the previous section satisfies the requirement for storage. The second principle is difficult to achieve exactly in practice. Typically, most implementations achieve running time close to $O(flops)$ for commonly used sparse matrix operations. For example, accessing a single element of a sparse matrix should be a constant time operation. With a CSR data structure, it typically takes time proportional to the logarithm of the length of the row to access a single element. Similarly, insertion of single elements into a CSR data structure generates extensive data movement. Such operations can be performed efficiently with the sparse/find routines (described next), which work with triples rather than individual elements.

4.1 Constructors

There are several ways to construct distributed sparse matrices in STAR-P:

1. ppback converts a sequential MATLAB® matrix to a distributed STAR-P matrix. If the input is a sparse matrix, the result is a dsparse matrix.

2. sparse creates a sparse matrix from dense vectors giving a list of non-zero values. A distributed sparse matrix is automatically created, if the dense vectors are distributed.

3. speye creates a sparse identity matrix.

4. spdiags constructs a sparse matrix by specifying the values on diagonals.

5. sprand and sprandn construct random sparse matrices with specified density.

6. spones creates a sparse matrix with the same non-zero structure as a given sparse matrix, where all the non-zero values are 1.

4.2 Element-wise matrix arithmetic

Sparse matrix arithmetic is implemented using a sparse accumulator (SPA). Gilbert, Moler and Schreiber [15] discuss the design of the SPA in detail. Briefly, a SPA uses a dense vector as intermediate storage. The key to making a SPA work is to maintain auxiliary data structures that allow direct ordered access to only the non-zero elements in the SPA. STAR-P uses a separate SPA for each processor.
4.3 Matrix multiplication

4.3.1 Sparse matrix dense vector multiplication

A sparse matrix can be multiplied by a dense vector either on the right or the left. The CSR data structure used in STAR-P is efficient for multiplying a sparse matrix by a dense vector: \( y = A \ast x \). It is efficient for communication and shows good cache behavior for the sequential part of the computation. Our choice of the CSR data structure was heavily influenced by our desire to have good matvec performance, since matvec forms the core computational kernel for many iterative methods.

The matrix \( A \) and vector \( x \) are distributed across processors by rows. The submatrix of \( A \) on each processor will need some subset of \( x \) depending upon its sparsity structure. When matvec is invoked for the first time on a dsparse matrix \( A \), STAR-P computes a communication schedule for \( A \) and caches it. When later matvecs are performed using the same \( A \), this communication schedule does not need to be recomputed, which saves some computing and communication overhead, at the cost of extra space required to save the schedule. We experimented with overlapping computation and communication in matvec. It turns out in many cases that this is less efficient than simply performing the communication first, followed by the computation. As computer architectures evolve, this decision may need to be revisited.

Communication in matvec can be reduced by graph partitioning. Lesser communication is required during matvec if fewer edges cross processor. STAR-P can use several of the available tools for graph partitioning [6, 17, 28]. However, STAR-P does not perform graph partitioning automatically during matvec. The philosophy behind this decision is similar to that in MATLAB: reorganizing data to make later operations more efficient should be possible for the user, but not automatic.

When multiplying from the left, \( y = x' \ast A \), instead of communicating just the required parts of the source vector, each processor computes its own destination vector. All partial destination vectors are then summed up into the final destination vector. This require \( O(n) \) communication. The choice of the CSR data structure which allows for efficient communication when multiplying from the right, makes it more difficult to multiply on the left.

Sparse matrix dense matrix multiplication in STAR-P is implemented as a series of matvecs. Such operations although not very common, do often show up in practice. It is tempting to simply convert the sparse matrix to a dense matrix and perform dense matrix multiplication; the reasoning being that the result will be dense in any case. Doing so, however, requires extra floating point operations. Such a scheme may also be inefficient in storage if the resulting matrix is smaller in dimensions than the sparse argument.

4.3.2 Sparse matrix sparse matrix multiplication

The multiplication of two sparse matrices is an important operation in STAR-P. It is a common operation for operating on large graphs. Its application to graph manipulation and numerical solvers is described by Shah [33]. Our implementation of sparse matrix matrix multiplication is described by Robertson [30] and Shah [33].
function C = mult_inner_prod (A, B)
% Inner product formulation of matrix multiplication
for i = 1:n
% For each row of A
    for j = 1:n
% For each col of B
        C(i, j) = A(i, :) * B(:, j);
    end
end
end

Figure 2: Inner product formulation of matrix multiplication. Every element of $C$ is computed as a dot product of a row of $A$ and a column of $B$

function C = mult_outer_prod (A, B)
% Outer product formulation of matrix multiplication
for k = 1:n
    C = C + A(:, k) * B(k, :);
end
end

Figure 3: Outer product formulation of matrix multiplication. $C$ is computed as a sum of $n$ rank one matrices.

The computation for matrix multiplication can be organized in several ways, leading to different formulations. One common formulation is the inner product formulation, as shown in code fragment 2. In this case, every element of the product $C_{ij}$ is computed as a dot product of a row $i$ in $A$ and a column $j$ in $B$.

Another formulation of matrix multiplication is the outer product formulation (code fragment 3). The product is computed as a sum of $n$ rank one matrices. Each rank one matrix is computed as the outer product of column $k$ of $A$ and row $k$ of $B$.

MATLAB® stores its matrices in the CSC format. Clearly, computing inner products (code fragment 2) is inefficient, since rows of $A$ cannot be efficiently accessed without searching. Similarly, in the case of computing outer products (code fragment 3), rows of $B$ have to be extracted. The process of accumulating successive rank one updates is also inefficient, as the structure of the result changes with each successive update.

The computation can be setup so that $A$ and $B$ are accessed by columns, computing one column of the product $C$ at a time. Code fragment 4 shows how column $j$ of $C$ is computed as a linear combination of the columns of $A$ as specified by the nonzeros in column $j$ of $B$. Figure 5 shows the same concept graphically.

STAR-P stores its matrices in CSR form. As a result, the computation is setup so that only rows of $A$ and $B$ are accessed, producing a row of $C$ at a time. Each row $i$ of $C$ is computed as a linear combination of the rows of $B$ specified by non-zeros in row $i$ of $A$ (code fragment 6).
function C = mult_csc(A, B)

% Multiply matrices stored in compressed sparse column format

for j = 1:n
    for k where B(k,j) ~= 0
        C(:, j) = C(:, j) + A(:, k) * B(k, j);
    end
end

Figure 4: The column-wise formulation of matrix multiplication accesses all matrices A, B and C by columns only

Figure 5: Multiplication of sparse matrices stored by columns. Columns of A are accumulated as specified by the non-zero entries in a column of B using a SPA. The contents of the SPA are stored in a column of C once all required columns are accumulated.
function C = mult_csr (A, B)
% Multiply matrices stored in compressed sparse row format
for i = 1:n
    for k where A(i,k) ~= 0
        C(i, :) = C(i, :) + A(i, k) * B(k, :);
    end
end

Figure 6: The row-wise formulation of matrix multiplication accesses all matrices A, B and C by rows only.

The performance of sparse matrix multiplication in parallel depends upon the non-zero structures of A and B. A well-tuned implementation may use a polyalgorithm. Such a polyalgorithm may use different communication schemes for different matrices. For example, it may be efficient to broadcast the local part of a matrix to all processors, but in other cases, it may be efficient to send only the required rows. On large clusters, it may be efficient to interleave communication and computation. On shared memory architectures, however, most of the time is spent accumulating updates, rather than in communication. In such cases, it may be more efficient to schedule the communication before the computation. In the general case, the space required to store C cannot be determined quickly, and Cohen’s algorithm [9] may be used in such cases.

4.4 Sparse matrix indexing, assignment, and concatenation

Several choices are available to the implementor to design primitives upon which a sparse matrix library is built. One has to decide early on in the design phase which operations will form the primitives and how other operations will be derived from them.

The syntax of matrix indexing in Star-P is the same as in MATLAB®. It is of the form A(p,q), where p and q are vectors of indices.

>> B = A(p,q)
In this case, the indexing is done on the right side of “=”, which specifies that B is assigned a submatrix of A. This is the subsref operation in MATLAB®.

>> B(p,q) = A

On the other hand, indexing on the left side of “=” specifies that A should be stored as a submatrix of B. This is the subsasgn operation in MATLAB®. Repeated indices in subsref cause replication of rows and columns. However, subsasgn with repeated indices is not well defined.
MATLAB® supports horizontal and vertical concatenation of matrices. The following code, for example, concatenates $A$ and $B$ horizontally, $C$ and $D$ horizontally, and finally concatenates the results of these two operations vertically.

\[
> S = [ A \ B \ C \ D ]
\]

All of these operations are widely used, and users often do not give second thought to the way they use indexing operations. The operations have to accept any sparse matrix, and return a result in the same form with reasonable performance. Communication adds another dimension of complexity in a parallel implementation such as STAR-P. Performance of sparse indexing operations depends upon the underlying data structure, the indexing scheme being used, the non-zero structure of the matrix, and the speed of the communication network.

Our implementation uses `sparse` and `find` as primitives to implement sparse indexing. The idea is actually quite simple. First, find all elements that match the selection criteria on each processor. Depending on the operation being performed, rows and columns may need to be renumbered. Once all processors have picked the non-zero tuples which will contribute to the result, call `sparse` to assemble the matrix.

Such a scheme is elegant because all the complexity of communication is hidden in the call to `sparse`. This simplifies the implementor’s job, who can then focus on simply developing an efficient `sparse` routine.

### 4.5 Sparse matrix transpose

Matrix transpose exchanges the rows and columns of all elements of the matrix. Transpose is an important operation, and has been widely studied in the dense case. In a sparse transpose, apart from communication, the communicated elements have to be re-inserted into the sparse data structure. The MATLAB® syntax for matrix transpose is as follows:

\[
> S = A'
\]

Sparse matrix transpose can be easily implemented using the `sparse` and `find` primitives. First, find all nonzero elements in the sparse matrix with `find`. Then construct the transpose with `sparse`, exchanging the vectors for rows and columns.

\[
[I, J, V] = \text{find} (S);
St = \text{sparse} (J, I, V);
\]

### 4.6 Direct solvers for sparse linear systems

MATLAB® solves the linear system $Ax = b$ with the matrix division operator, $x = A\backslash b$. In sequential MATLAB®, $A\backslash b$ is implemented as a polyalgorithm [15], where every test in the polyalgorithm is cheaper than the next one.

1. If $A$ is not square, solve the least squares problem.
2. Otherwise, if $A$ is triangular, perform a triangular solve.

3. Otherwise, test whether $A$ is a permutation of a triangular matrix (a “morally triangular” matrix), permute it, and solve it if so.

4. Otherwise, if $A$ is Hermitian and has positive real diagonal elements, find a symmetric approximate minimum degree ordering $p$ of $A$, and perform the Cholesky factorization of $A(p, p)$. If successful, finish with two sparse triangular solves.

5. Otherwise, find a column minimum degree order $p$, and perform the LU factorization of $A(:, p)$. Finish with two sparse triangular solves.


Different issues arise in parallel polyalgorithms. For example, morally triangular matrices and symmetric matrices are harder to detect in parallel. In the next section, we present a probabilistic approach to test for matrix symmetry. Design of the best polyalgorithm for “backslash” in parallel is an active research problem. For now, STAR-P offers the user a choice between two existing message-passing parallel sparse solvers: MUMPS [1] and SuperLU Dist [24].

Sparse solvers are extremely complex pieces of software, often taking several years to develop. They use subtle techniques to extract locality and parallelism, and have complex data structures. Most sparse solvers provide an interface only to solve linear systems, $x = A b$. They often do not provide an interface for the user to obtain the factors from the solve, $[L, U] = lu(A)$. MATLAB® uses UMFPACK only when backslash or the four output version of $lu$ is used, $[L, U, P, Q] = lu(A)$. Many MATLAB® codes store the results of LU factorization for later use. Since STAR-P does not yet provide a sparse $lu$ implementation, it may not be able to run certain MATLAB® codes in parallel.

4.7 Iterative solvers for sparse linear systems

Iterative solvers for sparse linear systems include a wide variety of algorithms that use successive approximations at each step. Stationary methods are older, simpler, but usually not very effective. These include methods such as Jacobi, Gauss-Seidel and successive overrelaxation. Nonstationary methods, also known as Krylov subspace methods, are relatively modern, and based on the idea of sequences of orthogonal vectors. Their convergence typically depends upon the condition number of the matrix. Often, a preconditioner is used to transform a given matrix into one with a more favorable spectrum, accelerating convergence.

Iterative methods are not used by default for solving linear systems in STAR-P. This is mainly because efficient methods are not yet available for all classes of problems. Conjugate Gradient (CG) works well for matrices which are symmetric and positive definite (SPD). Methods such as Generalized Minimal Residual (GMRES) or Biconjugate gradient (BiCG,
Matlab® and Star-P spy plots of a web crawl sparse matrix. The Star-P plot also exposes the underlying block row distribution of the sparse matrix. The matrix was constructed by running a breadth-first web crawl from www.mit.edu.

BiCGStab) are used for unsymmetric matrices. However, convergence may be irregular, and it is also possible that the methods may break.

Even when using CG, a preconditioner is required for fast convergence. Preconditioners are often problem specific; their construction often requires knowledge of the problem at hand. An exciting new area of research is combinatorial preconditioners. The graph toolbox in Star-P provides tools for users to build such preconditioners [33].

Although Star-P does not use iterative methods by default, it provides several tools for users to use them when suitable. Preconditioned iterative methods from software such as Aztec [31] and Hypre [14] may also be used in Star-P through the Star-P SDK [23].

4.8 Eigenvalues and singular values

Star-P provides eigensolvers for sparse matrices through PARPACK [26]. PARPACK uses a reverse communication interface, in which it provides the essential routines for the Arnoldi factorization, and requires the user to provide routines for matvec and linear solves. Star-P implementations of matvec and linear solvers are discussed in earlier sections.

Star-P retains the same calling sequence as the Matlab® eigs function. Star-P can also provide singular values in a similar fashion, and retains the same calling sequence for svds.

4.9 Visualization of sparse matrices

We have experimented with a few different methods to visualize sparse matrices in Star-P. A Matlab® spy plot of a sparse matrix shows the positions of all non-zeros. For extremely large sparse matrices, this approach does not work very well since each pixel on the screen represents a fairly large part of a matrix. Figure 7 shows a Matlab® spy plot of a web crawl matrix. It also shows a colored spy plot with a different color for each processor.
Figure 8: A density spy plot. For large matrices, spy may not display the underlying structure. A density plot colors each pixel according the density of the area of the sparse matrix it represents.

The row-wise distribution of the matrix is clearly observed in the colored spy plot. Another approach is to use a 2D histogram or a density spy plot, such as the one in Figure 8, which uses different colors for different non-zero densities. spyy uses sparse matrix multiplication to produce density spy plots. It is similar to the cspy routine in CSparse [11], with the exception that cspy is implemented in C, and cannot be used in STAR-P. spyy operates on large dsparse matrices on the backend, but the resulting images are small, which are easily communicated to the frontend.

5 SSCA #2 graph analysis benchmark

We now describe our implementation of a graph analysis benchmark, which builds upon the sparse matrix infrastructure in STAR-P.

The SSCAs (Scalable Synthetic Compact Applications) are a set of benchmarks designed to complement existing benchmarks such as the HPL [13] and the NAS parallel benchmarks [5]. Specifically, SSCA #2 [3] is a compact application that has multiple kernels accessing a single data structure (a directed multigraph with weighted edges). We describe our implementation of version 1.1 of the benchmark. Version 2.0 [4], which differs significantly from version 1.1, has since been released.

The data generator generates an edge list in random order for a multigraph of sparsely connected cliques as shown in Figure 9. The four kernels are as follows:
Figure 9: The left image shows the conceptual SSCA #2 graph (Kepner). The image on the right is an SSCA #2 graph generated with scale 8 (256 nodes) and plotted with Fiedler co-ordinates.

1. Kernel 1: Create a data structure for further kernels.
2. Kernel 2: Search graph for a maximum weight edge.
4. Kernel 4: Recover the underlying clique structure from the undirected graph.

5.1 Scalable data generator

The data generator is the most complex part of our implementation. It generates edge tuples for subsequent kernels. No graph operations are performed at this stage. The input to the data generator is a \( \text{scale} \) parameter, which indicates the size of the graph being generated. The resulting graph has \( 2^{\text{scale}} \) nodes, with a maximum clique size of \( \lfloor 2^{\text{scale}/3} \rfloor \), a maximum of 3 edges with the same endpoints, and a probability of 0.2 that an edge is uni-directional. Table 5.1 shows statistics for graphs generated with this data generator at different scales.

The vertex numbers are randomized, and a randomized ordering of the edge tuples is presented to the subsequent kernels. Our implementation of the data generator closely follows the pseudocode published in the spec [3].

5.2 Kernel 1

Kernel 1 creates a read-only data structure that is used by subsequent kernels. We create a sparse matrix corresponding to each layer of the multigraph. The multigraph has three layers, since there is a maximum of three parallel edges between any two nodes in the graph. MATLAB® provides several ways of constructing sparse matrices, \texttt{sparse}, which takes as its input a list of three-tuples: \((i,j,w_{ij})\). Its output is a sparse matrix with a nonzero \(w_{ij}\) in
<table>
<thead>
<tr>
<th>Scale</th>
<th>#Vertices</th>
<th>#Cliques</th>
<th>#Edges (Directed)</th>
<th>#Edges (Undirected)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1024</td>
<td>186</td>
<td>13,212</td>
<td>3,670</td>
</tr>
<tr>
<td>15</td>
<td>32,768</td>
<td>2,020</td>
<td>1,238,815</td>
<td>344,116</td>
</tr>
<tr>
<td>20</td>
<td>1,048,576</td>
<td>20,643</td>
<td>126,188,649</td>
<td>35,052,403</td>
</tr>
<tr>
<td>25</td>
<td>33,554,432</td>
<td>207,082</td>
<td>12,951,350,000</td>
<td>3,597,598,000</td>
</tr>
<tr>
<td>30</td>
<td>1,073,741,824</td>
<td>2,096,264</td>
<td>1,317,613,000,000</td>
<td>366,003,600,000</td>
</tr>
</tbody>
</table>

Table 1: Statistics for the SSCA#2 graph (version 1.1). The directed edges column counts the number of edges in the directed multigraph. The undirected edges column counts the number of edges in the undirected graph used for kernel 4. The statistics are generated by simulating the data generator.

Figure 10: MATLAB® spy plot of the input graph. The input graph is randomized, as evidenced by no observed patterns in the spy plot.

every location \((i,j)\) specified in the input. Figure 10 shows a spy plot of one layer of the input graph.

5.3 Kernel 2

In kernel 2, we search the graph for edges with maximum weight. \texttt{find} is the inverse of \texttt{sparse}. It returns all nonzeros from a sparse matrix as a list of three-tuples. We then use \texttt{max} to find the maximum weight edge.

5.4 Kernel 3

In kernel 3, we perform breadth first searches from a given set of starting points. We use sparse matrix matrix multiplication to perform all breadth first searches simultaneously from the given starting points. Let \(G\) be the adjacency matrix representing the graph and \(S\) be a matrix corresponding to the starting points. \(S\) has one column for each starting point, and one non-zero in each column. Breadth first search is performed by repeatedly multiplying \(G\) by \(S\): \(Y = G \times S\). We perform several breadth first searches simultaneously by using sparse matrix matrix multiplication. STAR-P stores sparse matrices by rows, and parallelism is achieved by each processor computing some rows in the product [30, 32].
Figure 11: The image on the left is a spy plot of the graph, reordered after clustering. The image on the right magnifies a portion around the diagonal. Cliques are revealed as dense blocks on the diagonal.

### 5.5 Kernel 4

Kernel 4 is the most interesting part of the benchmark. It can be considered to be a partitioning problem or a clustering problem. We have several implementations of kernel 4 based on spectral partitioning (figure 9), “seed growing” (figure 11), and “peer pressure” algorithms. The peer pressure and seed growing implementations scale better than the spectral methods, as expected. We now demonstrate how we use the infrastructure described above to implement kernel 4 in a few lines of MATLAB® or Star-P. Figure 11 shows a spy plot of the undirected graph after clustering. The clusters show up as dense blocks along the diagonal.

Our seed growing algorithm (figure 12) starts by picking a small set of seeds (about 2% of the total number of nodes) randomly. The seeds are then grown so that each seed claims all nodes reachable by at least $k$ paths of length 1 or 2, where $k$ is the size of the largest clique. This may cause some ambiguity, since some nodes might be claimed by multiple seeds. We tried picking an independent set of nodes from the graph by performing one round of Luby’s algorithm [25] to keep the number of such ambiguities as low as possible. However, the quality of clustering remains unchanged when we use random sampling. We use a simple approach for disambiguation: the lowest numbered cluster claiming a vertex claims it. We also experimented with attaching singleton nodes to nearby clusters to improve the quality of clustering.

Our peer pressure algorithm (figure 13) starts with a subset of nodes designated as leaders. There has to be at least one leader neighboring every vertex in the graph. This is followed by a round of voting where every vertex in the graph selects a leader, selecting a cluster to join. This does not yet yield good clustering. Each vertex now looks at its neighbors and switches its vote to the most popular leader in its neighborhood. This last step is crucial, and in this case, it recovers more than 95% of the original clique structure of the graph. Figure 14 shows the different stages of the peer pressure algorithm on an example graph.

We experimented with different approaches to select leaders. At first, it seemed that a maximal independent set of nodes from the graph was a natural way to pick leaders. In
function J = seedgrow (seeds)
% Clustering by breadth first searches

% J is a sparse matrix with one seed per column.
J = sparse (seeds, 1:nseeds, 1, n, nseeds);

% Vertices reachable with 1 hop.
J = G * J;
% Vertices reachable with 1 or 2 hops.
J = J + G*J;
% Vertices reachable with at least k paths of 1 or 2 hops.
J = J >= k;

Figure 12: Breadth first parallel clustering by seed growing.

function cluster = peerpressure (G)
% Clustering by peer pressure

% Use maximal independent set routine from GAPDT
IS = mis (G);

% Find all neighbors in the independent set.
neighbors = G * sparse(IS, IS, 1, length(G), length(G));

% Each vertex chooses a random neighbor in the independent set.
R = sprand (neighbors);
[ignore, vote] = max (R, [], 2);

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

Figure 13: Parallel clustering by peer pressure
practice, it turns out that simple heuristics (such as the highest numbered neighbor) gave equally good clustering. We also experimented with more than one round of voting. The marginal improvement in the quality of clustering was not worth the additional computation time.

5.6 Visualization of large graphs

Graphics and visualization are a key part of an interactive system such as MATLAB®. The question of how to effectively visualize large datasets in general, especially large graphs, is still unsolved. We successfully applied methods from numerical computing to come up with meaningful visualizations of the SSCA #2 graph.

One way to compute geometric co-ordinates for the nodes of a connected graph is to use Fiedler co-ordinates [19] for the graph. Figure 9 shows the Fiedler embedding of the SSCA #2 graph. In the 2D case, we use the eigenvectors (Fiedler vectors) corresponding to the first two non-zero eigenvalues of the Laplacian matrix of the graph as co-ordinates for nodes of the graph in a plane.

For 3D visualization of the SSCA #2 graph, we start with 3D Fiedler co-ordinates projected onto the surface of a sphere. We model nodes of the graph as particles on the surface of a sphere. There is a repulsive force between all particles, inversely proportional to the distance between them. Since these particles repel each other on the surface of a sphere, we expect them to spread around and occupy the entire surface of the sphere. Since there are cliques in the original graph, we expect clusters of particles to form on the surface of the sphere. At each timestep, we compute a force vector between all pairs of particles. Each particle is then displaced some distance based on its force vector. All displaced particles are projected back onto the sphere at the end of each timestep.

This algorithm was used to generate figure 15. In this case, we simulated 256 particles
Figure 15: The 3D visualization of the SSCA #2 graph on the left is produced by relaxing the Fiedler co-ordinates projected onto the surface of a sphere. The right figure shows a density spy plot of the SSCA #2 graph.

and the system was evolved for 20 timesteps. It is important to first calculate the Fiedler co-ordinates. Beginning with random co-ordinates results in a meaningless picture. We used PyMOL [12] to render the graph.

5.7 Experimental Results

We ran our implementation of SSCA #2 (ver 1.1, integer only) in STAR-P. The MATLAB® client was run on a generic PC. The STAR-P server was run on an SGI Altix with 128 Itanium II processors with 128G RAM (total, non-uniform memory access). We used a graph generated with scale 21. This graph has 2 million nodes. The multigraph has 321 million directed edges; the undirected graph corresponding to the multigraph has 89 million edges. There are 32 thousand cliques in the graph, the largest having 128 nodes. There are 89 million undirected edges within cliques, and 212 thousand undirected edges between cliques in the input graph for kernel 4. The results are presented in Fig. 16.

Our data generator scales well; the benchmark specification does not require the data generator to be timed. A lot of time is spent in kernel 1, where data structures for the subsequent kernels are created. The majority of this time is spent in searching the input triples for duplicates, since the input graph is a multigraph. Kernel 1 creates several sparse matrices using sparse, each corresponding to a layer in the multigraph. Time spent in kernel 1 also scales very well with the number of processors. Time spent in Kernel 2 also scales as expected.

Kernel 3 does not show speedups at all. Although all the breadth first searches are performed in parallel, the process of subgraph extraction for each starting point creates a lot of traffic between the STAR-P client and the STAR-P server, which are physically in different states. This client server communication time ends up dominating over the computation time. This overhead can be minimized by vectorizing kernel 3 more aggressively.

Kernel 4, the non-trivial part of the benchmark, actually scales very well. We show results
Figure 16: SSCA #2 version 1.1 execution times (Star-P, Scale=21)

for our best performing implementation of kernel 4, which uses the seed growing algorithm.

The evaluation criteria for the SSCAs also include software engineering metrics such as code size, readability, maintainability, etc. Our implementation is extremely concise. We show the source lines of code (SLOC) for our implementation in Table 2. We also show absolute line counts, which include blank lines and comments, as we believe these to be crucial for code readability and maintainability. Our implementation runs without modification in sequential MATLAB®, making it easy to develop and debug on the desktop before deploying on a parallel platform.

We have run the full SSCA #2 benchmark (version 0.9, integer only) on graphs with $2^{27} = 134$ million nodes on the SGI Altix. Scaling results for the full benchmark (version 1.1, integer only) are presented in figure 16. We have also generated and manipulated extremely large graphs (1 billion nodes and 8 billion edges) on an SGI Altix with 256 processors using Star-P.

This demonstrates that the sparse matrix representation is a scalable and efficient way to manipulate large graphs. Not that the codes in figure 12 and figure 13 are not pseudocodes, but actual code excerpts from our implementation. Although the code fragments look very simple and structured, the computation manipulates sparse matrices, resulting in highly irregular communication patterns on irregular data structures.
Table 2: Line counts for STAR-P implementation of SSCA#2. The “Source LOC” column counts only executable lines of code, while the “Total line counts” column counts the total number of lines including comments and whitespace.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source LOC</th>
<th>Total line counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data generator</td>
<td>176</td>
<td>348</td>
</tr>
<tr>
<td>Kernel 1</td>
<td>25</td>
<td>63</td>
</tr>
<tr>
<td>Kernel 2</td>
<td>11</td>
<td>34</td>
</tr>
<tr>
<td>Kernel 3</td>
<td>23</td>
<td>48</td>
</tr>
<tr>
<td>Kernel 4 (spectral)</td>
<td>22</td>
<td>70</td>
</tr>
<tr>
<td>Kernel 4 (seed growing)</td>
<td>55</td>
<td>108</td>
</tr>
<tr>
<td>Kernel 4 (peer pressure)</td>
<td>6</td>
<td>29</td>
</tr>
</tbody>
</table>

6 Looking forward: A next generation parallel sparse library

We now discuss the design goals of a next generation parallel sparse library, based on our current experience.

Our initial goal was to develop a parallel sparse library for STAR-P similar to the one in MATLAB®. We wanted it to be robust, scalable, efficient and simple. Hence, all the design decisions we made always favored robustness and simplicity. For instance, we decided early on to support only the CSR data structure to store sparse matrices, and use a 1D block layout by rows. A 2D layout may be more efficient, and it would definitely be nice to support other data structures for storage. However, these would complicate the implementation to a point where it may not be possible to implement all the operations we currently support for all combinations of data structures and layouts.

That said, there are some crucial differences between MATLAB® and STAR-P’s sparse libraries. First, MATLAB®’s focus is solely on numerical computing. However, sparse matrices are increasingly lending themselves to more than just numerical computing. Second, parallel computing is still not as easy as sequential computing. Parallel sparse matrices provide an elegant way for a user to represent large sparse datasets as matrices without using complex data structures to store and query them. Operations on such large datasets often require irregular communication and irregular data access. Sparse matrix computations allow this to be done in a concise and systematic way, without worrying about low level details. The use of distributed sparse matrices to represent graphs for combinatorial computing is one such example [33].

We have learnt some lessons from our experience with the STAR-P sparse matrix library, which might be helpful for the development of a next generation parallel sparse library.

Although the CSR data structure has served us well, it does cause some problems. MATLAB® codes written to specifically take advantage of the column storage of sparse matrices must be rewritten to use row storage. Although it is not yet clear how much the performance difference may be for a real life application, it is inconvenient for a user writing
highly tuned codes using sparse matrices. Such a code will also have different performance characteristics in \textsc{Matlab}® and \textsc{Star-P}.

We propose adding a third design principle to the two stated in section 4. The difference in performance between accessing a sparse matrix by rows or columns must be minimal. Users writing sparse matrix codes should not have to worry about organizing their sparse matrix accesses by rows or columns, just as they do not worry about how dense matrices are stored.

For an example of the third principle, consider the operation below. It is much simpler to extract the submatrix specified by \( p \) from a matrix stored in the CSR format than from a matrix stored in the CSC format. In the latter case, a binary search is required for every column, making the operation slower.

\[
\begin{align*}
\text{>> } A &= S(p, :) \quad \% \text{ } p \text{ is a vector}
\end{align*}
\]

A parallel sparse library that needs to scale to hundreds or thousands of processors will not work well with a one-dimensional block layout. A two-dimensional block layout is essential for scalability of several sparse matrix operations. The benefits of 2D layouts are well known at this point, and we will not reiterate them. It is important to note that compressed row/column data structures are not efficient for storing sparse matrices in a 2D layout.

Another point of departure is a different primitive for indexing operations. Currently, we use the “sparse-find” method to perform all the indexing operations such as submatrix indexing, assignment, concatenation and transpose. We used the concept of the \texttt{sparse} function as a primitive using which we built the rest of the operations. We propose that a next generation sparse matrix library should use sparse matrix multiplication as the basic primitive in the library.

We illustrate the case of submatrix indexing using matrix multiplication. Suppose \( A \) is a matrix, and we wish to extract the submatrix \( B = A(I, J) \). Multiplying from the left picks out the rows, while multiplying from the right picks out the columns, as shown in code fragment 17.

We believe that it will be possible to implement sparse matrix multiplication more efficiently with a 2D block distribution than a 1D block distribution for large numbers of processors. Indexing operations may then be implemented using sparse matrix multiplication as a primitive. An efficient sparse matrix multiplication implementation might actually use a polyaalgorithm to simplify the implementation for special cases when more information is available about the structure of matrices being multiplied, as is the case for indexing operations.

The choice of a suitable data structure to store sparse matrices in a 2D block layout and allow efficient matrix multiplication still remains an open question for future research. We believe that once the right data structure is selected, there will not be a large difference in performance when multiplying from the left or the right. This directly translates into symmetric performance for all indexing operations when accessing a matrix either by rows or columns. We believe this will lead to higher programmer productivity, freeing users from
function B = index_by_mult (A, I, J)
% Index a matrix with matrix multiplication

[nrows, nccols] = size(A);
nI = length(I);
NJ = length(J);

% Multiply on the left to pick the required rows
row_select = sparse(1:nI, I, 1, nI, nr);

% Multiply on the right to pick the required columns
col_select = sparse(J, 1:nJ, 1, nc, nJ);

% Compute B with sparse matrix multiplication
B = row_select * A * col_select;

Figure 17: Matrix indexing and concatenation can be implemented using sparse matrix-
matrix multiplication as a primitive. Multiplication from the left picks the necessary rows,
while multiplication from the right picks the necessary columns.

7 Conclusion

We described the design and implementation of a distributed sparse matrix infrastructure in
STAR-P. This infrastructure was used to build the “Graph Algorithms and Pattern Discovery
Toolbox (GAPDT)”. We demonstrated the effectiveness of our tools by implementing a graph
analysis benchmark in STAR-P, which scales to large problem sizes on large processor counts.

We conclude that distributed sparse matrices provide a powerful set of primitives for
numerical and combinatorial computing. We hope that the our experiences will shape the
design of future parallel sparse matrix infrastructures in other languages.
References


