Parallel algorithms for sparse matrix product, indexing, and assignment

Aydın Buluç
Lawrence Berkeley National Laboratory
February 8, 2012

Joint work with John R. Gilbert (UCSB)

With inputs from G. Ballard, J. Demmel, O. Schwartz, E. Solomonik.
Linear-algebraic primitives

Sparse matrix-matrix multiplication (SpGEMM)
\[ C = A \times B \]

Sparse matrix indexing (SpRef)
\[ A = B(I, J) \]

Sparse matrix assignment (SpAsgn)
\[ B(I, J) = A \]

\( A, B \): sparse matrices with arbitrary nonzero distribution
\( I, J \): vectors of indices (arbitrary order and length)
Applications of Sparse GEMM

- **Graph clustering** (*MCL*, peer pressure)
- Subgraph / submatrix indexing
- Shortest path calculations
- **Betweenness centrality**
- Graph contraction
- Cycle detection
- Multigrid interpolation & restriction
- Colored intersection searching
- Applying constraints in finite element computations
- Context-free parsing ...

Loop until convergence

\[
A = A \cdot A; \quad \% \text{expand}
\]
\[
A = A .^2; \quad \% \text{inflate}
\]
\[
sc = 1./\text{sum}(A, 1);
\]
\[
A = A \cdot \text{diag}(sc); \quad \% \text{renormalize}
\]
\[
A(A < 0.0001) = 0; \quad \% \text{prune entries}
\]
Some terminology

\[ \text{nnz: number of nonzeros} \]
\[ \text{flops: number of nonzero arithmetic operations required} \]
\[ \text{nzr: number of rows that are not entirely zero} \]
\[ \text{nzc: number of columns that are not entirely zero} \]
\[ \text{ni: number of indices } i \text{ for which } A(:,i) \neq 0 \text{ and } B(i,:) \neq 0 \]

If nonzeros of A and B are i.i.d with \( d \) nonzeros per row/column, then \( \text{nnz} = dn \) and \( \text{flops} = d^2n \)
1) **outer product:**
   for $k = 1:n$
   
   $C = C + A(:, k) \times B(k, :)$

2) **inner product:**
   for $i = 1:n$
   for $j = 1:n$
   
   $C(i, j) = A(i, :) \times B(:, j)$
3) **Row-by-row formulation:**

for $i = 1:n$

forall $k$ s.t. $A(i,k) \neq 0$

$C(i,:) = C(i,:) + A(i,k) \times B(k,:)$

- **Complexity:** $O(n + \text{nnz} + \text{flops})$
- Due to Gustavson (1978), implemented in Matlab and CSparse.
- Fastest general purpose algorithm in serial.
- “flops-optimal” for $\text{flops} > n, \text{nnz}$
Yuster & Zwick (2005): $\text{nnz}^{0.7} n^{1.2} + n^{2+o(1)}$

1. Perform outer-product of dense rows and columns using fast (Strassen-like) dense rectangular matrix multiplication.
2. Use sparse algorithm for remaining rows and columns.

- “Worst-case optimal”, but worst case only happens for $\text{nnz}(C)=n^2$
- Not “flops optimal”, hence only suitable when output is dense.
- Many applications require a classical (semiring) matrix multiply.
Two versions of Sparse GEMM

1D block-column distribution

\[ C_i = C_i + A B_i \]

Checkerboard (2D block) distribution

\[ C_{ij} += A_{ik} B_{kj} \]
Projected speedup of Sparse 1D & 2D

In practice, 2D algorithms have the potential to scale, but not linearly

\[ E = \frac{W}{p(T_{\text{comp}} + T_{\text{comm}})} = \frac{d^2 n}{\beta d n \sqrt{p} + d^2 n \log\left(\frac{d^2 n}{p}\right) + \alpha p \sqrt{p}} \]
Compressed Sparse Columns (CSC): A Standard Layout

- Stores entries in column-major order
- Dense collection of “sparse columns”
- Uses $O(n + nnz)$ storage.
Submatrix storage in 2D

Submatrices are "hypersparse" (i.e. $nnz \ll n$)

- Average of $d$ nonzeros per column

\[ nnz' = \frac{d}{\sqrt{p}} \rightarrow 0 \]

Total Storage:
\[ O(n + nnz) \rightarrow O(n\sqrt{p} + nnz) \]

- A data structure or algorithm that depends on matrix dimension $n$ (e.g. CSR or CSC) is asymptotically too wasteful for submatrices

- Use doubly-compressed (DCSC) data structures or compressed sparse blocks (CSB) instead.
Complexity measure trends with increasing $p$ in 2D

Gustavson’s algorithm is $O(nnz + flops + n)$

$n'(\text{dimension}) \approx \frac{n}{\sqrt{p}}$

$nnz'(\text{data size}) \approx \frac{nnz}{p}$

$flops'(\text{work}) \approx \frac{flops}{p\sqrt{p}}$

When multiplying two R-MAT matrices with $nnz/n = 8$ (column/row nonzero counts follow a power law)
Sequential “hypersparse” kernel

Operates on the strictly \(O(\text{nnz})\) DCSC data structure

Sparse outer-product formulation with multi-way merging

Efficient in parallel, i.e. \(T(1) \approx p \ T(p)\)

Time complexity:
\[
O(\text{flops} \cdot \lg ni + \text{nz}c(A) + \text{nz}r(B))
\]
- independent of dimension

Space complexity:
\[
O(\text{nnz}(A) + \text{nnz}(B) + \text{nnz}(C))
\]
- independent of flops
2D algorithm: Sparse SUMMA

Based on dense SUMMA

General implementation that handles rectangular matrices

\[ C_{ij} += \text{HyperSparseGEMM}(A_{\text{recv}}, B_{\text{recv}}) \]
Experimental details

**Platform:** NERSC Franklin, Cray XT4 with quad-core processors.

**Test cases:**
1. Square sparse matrix multiplication
2. Multiplication with the restriction operator
3. Tall skinny right-hand-side matrix

**Main matrix generator:** R-MAT with 8 nonzeros per column

Spy plot of R-MAT matrix (yellows denote nonzeros)

Scale N matrix is $2^N$ by $2^N$

Double precision arithmetic
Square sparse matrix multiplication

Linear scaling until bandwidth costs starts to dominate

Scaling proportional to $\sqrt{p}$ afterwards
Multiplication with the restriction operator

\[
\begin{array}{c}
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{array}
\end{array} \times
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\times
\begin{array}{c}
\begin{array}{ccc}
1 & 1 & 1 \\
\end{array}
\end{array} =
\begin{array}{c}
\begin{array}{c}
\end{array}
\end{array}
\end{array}
\]
Multiplication with the restriction operator

The full restriction operation of order 8 applied to a scale 23 R-MAT

(a) Left to right evaluation: \((SA)S^T\)

(b) Right to left evaluation: \(S(AS^T)\)

Restriction order does NOT affect performance since
- Our algorithms are dimension oblivious
- The expected \(\text{nnz}(C)\) is not affected.
Tall skinny right-hand-side matrix

\begin{align*}
\text{Y (nonzeros/col in fringe)} & \quad \text{X (processors)} \\
& \quad \text{Z (normalized time)}
\end{align*}

\begin{align*}
10^0 & \quad 10^1 & \quad 10^2 & \quad 10^3 & \quad 10^4 & \quad 10^5 \\
64 & \quad 256 & \quad 1024 & \quad 4096
\end{align*}
Comparison of SpGEMM implementations

SpSUMMA = 2-D data layout (Combinatorial BLAS)
EpetraExt = 1-D data layout (Trilinos)

(a) R-MAT × R-MAT product (scale 21).
(b) Multiplication of an R-MAT matrix of scale 23 with the restriction operator of order 8.
Need to reduce communication

- Normalized communication/computation breakdown
- Scale 23 R-MAT times restriction operator of order 4
Remember the 2D algorithm

\[ \text{Bandwidth} = \Theta\left(\frac{dn}{\sqrt{p}}\right) \]
Generalize SUMMA to 2.5D

Maximum replicas: \( c \leq \frac{3\sqrt{p}}{d^{2/3}} \)

**Bandwidth:**

\[ \Theta\left(\frac{d^{4/3} n}{p^{2/3}}\right) \]

- Better scaling with \( p \)
- Worse with \( d \)
Why are \textit{SpRef}/\textit{SpAsgn} important?

Subscripting and colon notation:
\begin{itemize}
\item \(\Rightarrow\) Batched and vectorized operations
\item \(\Rightarrow\) High performance and parallelism.
\end{itemize}

\begin{align*}
A &= \text{rmat}(15) \\
A(r,r) &; r \text{ random} \\
A(r,r) &; r = \text{symrcm}(A)
\end{align*}

\begin{itemize}
\item \(\text{Load balance hard}\) \pm \text{Some locality}
\item \(\text{Load balance easy}\)
\item \(\text{No locality}\)
\item \(\text{Good locality}\)
\end{itemize}
More applications of SpRef

Prune isolated vertices; plug-n-play way (Graph 500)

sa = sum(A); // A is symmetric, for undirected graph
nonisov = find(sa>0);
A = A(nonisov, nonisov); // prune isolated vertices
Used for **extracting subgraphs, coarsening grids, relabeling vertices**, etc.

**SpRef**: \( B = A(I, J) \)

**SpAsgn**: \( B(I, J) = A \)

**SpExpAdd**: \( B(I, J) += A \)

**SpRef** using mixed-mode sparse matrix-matrix multiplication (**SpGEMM**). Ex: \( B = A([2,4], [1,2,3]) \)
Sequential \texttt{SpRef} and \texttt{SpAsgn}

\begin{verbatim}
function B = spref(A,I,J)
    R = sparse(1:len(I),I,1,len(I),size(A,1));
    Q = sparse(J,1:len(J),1,size(A,2),len(J));
    B = R*A*Q;
end
\end{verbatim}

\begin{verbatim}
function C = spasgn(A,I,J,B)
    [ma,na] = size(A);
    [mb,nb] = size(B);
    R = sparse(I,1:mb,1,ma,mb);
    Q = sparse(1:nb,J,1,nb,na);
    S = sparse(I,I,1,ma,ma);
    T = sparse(J,J,1,na,na);
    C = A + R*B*Q - S*A*T;
end
\end{verbatim}

$O(\text{nnz}(A))$

$A + \begin{pmatrix} 0 & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 0 \\ 0 & A(I,J) & 0 \\ 0 & 0 & 0 \end{pmatrix}$

$O(\text{nnz}(A) + \text{nnz}(B) + \text{len}(I) + \text{len}(J))$
Parallel algorithm for \textit{SpRef}

Step 1: Form $R$ from $I$ in parallel, on a 3x3 processor grid

Forming $Q$:  
Similar row-wise communication, followed by $Q.\text{Transpose}()$ 

\[
\Theta \left(\alpha \cdot \log(p) + \beta \cdot \frac{\text{len}(I) + \text{len}(J)}{\sqrt{p}}\right)
\]
Parallel algorithm for SpRef

Step 2: \textbf{SpGEMM} using memory-efficient Sparse SUMMA.

Minimize temporaries by:
- Splitting local matrix, and broadcasting multiple times
- Deleting R (and A if in-place) after forming C=R*A

\[
T_{\text{comp}} = O\left(\frac{\text{len}(I) + \text{len}(J) + \text{nnz}(A)}{\sqrt{p}} + \frac{\text{nnz}(A)}{p} \cdot \log\left(\frac{\text{len}(I) + \text{len}(J)}{p} + \sqrt{p}\right)\right)
\]

\[
T_{\text{comm}} = \Theta\left(\alpha \cdot \sqrt{p} + \beta \cdot \frac{\text{nnz}(A)}{\sqrt{p}}\right)
\]

Dominated by \textbf{SpGEMM}

Bottleneck: bandwidth costs

Speedup: \(\Theta\left(\sqrt{p}\right)\)
Matrix/vector distributions, interleaved on each other.

Default distribution in Combinatorial BLAS.

- Performance change is marginal (dominated by \texttt{SpGEMM})
- Scalable with increasing number of processes
- No significant load imbalance
Strong scaling of SpRef

random symmetric permutation ⇔ relabeling graph vertices
- RMAT Scale 22; edge factor=8; a=.6, b=c=d=.4/3
- Franklin/NERSC, each node is a quad-core AMD Budapest
Strong scaling of SpRef

Extracts 10 random (induced) subgraphs, each with $|V|/10$ vert. Higher span $\rightarrow$ Decreased parallelism $\rightarrow$ Lower speedup
Conclusions

• Flexible and scalable **SpGEMM** algorithm: Sparse SUMMA

• Parallel algorithms for **SpRef** and **SpAsgn**

• Systemic algorithm structure imposed by **SpGEMM**

• Complexity analysis made possible for the general case

• Good strong scaling for 1000-way concurrency

• Many applications on sparse matrix and graph world

• Freely available within **Combinatorial BLAS**.

• All presented primitives are ultimately communication-bound
Future Work

- Competitive implementation of the 2.5D algorithm
- Lower bounds for communication
- Direct support for chain products.
- Robust asynchronous implementation.
All primitives incorporated into the Combinatorial BLAS:


**Hypersparsity in 2D decomposition, sequential kernel:** B., Gilbert, On the representation and multiplication of hypersparse matrices, IPDPS’08

**Parallel analysis of Sparse GEMM:** B., Gilbert, Challenges and advances in parallel sparse matrix-matrix multiplication, ICPP’08
**Some Combinatorial BLAS functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Applies to</th>
<th>Parameters</th>
<th>Returns</th>
<th>Matlab Phrasing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SpGEMM</strong></td>
<td>Sparse Matrix</td>
<td>(A, B, ) transpose (A) if true, transpose (B) if true</td>
<td>Sparse Matrix</td>
<td>(C = A \times B)</td>
</tr>
<tr>
<td></td>
<td>(as friend)</td>
<td>sparse matrices</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SpMV</strong></td>
<td>Sparse Matrix</td>
<td>(A, x, ) transpose (A) if true, sparse or dense vector(s)</td>
<td>Sparse or Dense Vector(s)</td>
<td>(y = A \times x)</td>
</tr>
<tr>
<td></td>
<td>(as friend)</td>
<td>sparse matrices</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SpWISEX</strong></td>
<td>Sparse Matrices</td>
<td>(A, B, ) negate (A) if true, negate (B) if true</td>
<td>Sparse Matrix</td>
<td>(C = A \times B)</td>
</tr>
<tr>
<td></td>
<td>(as friend)</td>
<td>sparse matrices</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>REDUCE</strong></td>
<td>Any Matrix</td>
<td>(\text{dim}, ) (\text{binop}) reduction operator</td>
<td>Dense Vector</td>
<td>(\text{sum}(A))</td>
</tr>
<tr>
<td></td>
<td>(as method)</td>
<td>dimension to reduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SPREF</strong></td>
<td>Sparse Matrix</td>
<td>(p, q) row indices vector, column indices vector</td>
<td>Sparse Matrix</td>
<td>(B = A(p, q))</td>
</tr>
<tr>
<td></td>
<td>(as method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SPASGN</strong></td>
<td>Sparse Matrix</td>
<td>(p, q) row indices vector, column indices vector</td>
<td>None</td>
<td>(A(p, q) = B)</td>
</tr>
<tr>
<td></td>
<td>(as method)</td>
<td>matrix to assign</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SCALE</strong></td>
<td>Any Matrix</td>
<td>(\text{rhs}) any object, (except a sparse matrix)</td>
<td>None</td>
<td>Check guiding principles 3 and 4</td>
</tr>
<tr>
<td></td>
<td>(as method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SCALE</strong></td>
<td>Any Vector</td>
<td>(\text{rhs}) any vector</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>(as method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>APPLY</strong></td>
<td>Any Object</td>
<td>(\text{unop}) unary operator, (applied to non-zeros)</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>(as method)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>