MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud

By Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, Zheng Zhang

Microsoft Research Asia, Shanghai Jiaotong University, Microsoft Research Silicon Valley

Presenter: Haikal Pribadi (hp356)
MadLINQ

Motivation
Contribution
Evaluation
Future Work
Motivation
Distributed Engines – Good and Bad

Success
- Strong subset of relational operators
  - Filtering, projection, aggregation, sorting and joins
  - Extensions via user-defined functions
- Adopts direct-acyclic-graph (DAG) execution model
  - Scalable and resilient

Problematic
- Deep analysis and manipulation of data
- Requires linear algebra and matrix computation
Distributed Engines - Problem

Linear algebra and matrix computation

- Machine Learning
  - Multiplication, SVD, LU factorization
  - Cholesky factorization
- Ranking or classification algorithm
- Social web-mining or information retrieval
- Hard to capture in relational algebra operators
- Real world matrix and data mining algorithms are extremely hard to implement
High Performance Computing

Solution to matrix computation

However

- Involves low level primitives to develop algorithms
- Single Process Multiple Data (SPMD) execution model
- Problem maintained in memory
- Constrains programmability, scalability and robustness
- Not applicable for web-scale big data analysis
HAMA – Matrix Operation on MapReduce

Removes the constraint of problem size

MapReduce interface is restrictive

- Difficult to program real world linear algebra
- Implicitly synchronized
- Fails to take advantage of semantics of matrix operations
Contribution
Matrix Computation System

Unified programming model
- Matrix development language
- Application development library

Integrate with data-parallel computing system

Maintain scalability and robustness of DAG
- Fine-grained pipelining (FGP)
- Lightweight fault-tolerance protocol
The diagram depicts the relationships between different computational frameworks and mathematical operations. On the left side, Relational Algebra is represented with DryadLINQ and Dryad. The right side is divided into Linear Algebra, Graph, and Combinatorial BLAS sections.

**Linear Algebra** includes operations like $AX = B$, Cholesky, SVD, PageRank, and K-Means.

**Graph** includes operations like BFS, MCL, and Betweenness.

**Combinatorial BLAS** includes operations like Dense matrix and Sparse matrix.

**.NET** acts as an interface layer, connecting the computational frameworks with the mathematical operations.
Programming Model - Matrix

Develop matrix algorithms

Matrix optimizations

Based on tile abstraction

- Square sub-matrices
- Indexed grid of tiles form a matrix
- Matrices expressed naturally
- Structural characteristic of matrices
Matrix multiplication code example:

MadLINQ.For(0, m, 1, i =>
{
    MadLINQ.For(0, p, 1, j =>
    {
        c[i, j] = 0;
        MadLINQ.For(0, n, 1, k =>
        {
            c[i, j] += a[i, k] * b[k, j];
        });
    });
});
Cholesky tile-algorithm implementation

MadLINQ.For(0, n, 1, k =>
{
    L[k, k] = A[k, k].DPOTRF();
    MadLINQ.For(k + 1, n, 1, l =>
        L[l, k] = Tile.DTRSM(L[k, k], A[l, k]));
    MadLINQ.For(k + 1, n, 1, m =>
    {
        A[m, m] = Tile.DSYRK(A[m, k], A[m, m]);
        MadLINQ.For(m + 1, n, 1, l =>
            A[l, m] = Tile.DGEMM(A[l, k], A[m, k], A[l, m]));
    });
});
Programming Model – Application ex.

Collaborative Filtering

- Baseline algorithm with data set from Netflix
- Dataset: matrix $R$ records users' ratings on movies

  - similarity = $R \times R^t$ (sparse matrix)
  - scores = similarity $\times R$ (dense matrix)

Matrix similarity = $R.Multiply(R.Transpose())$;
Matrix scores = similarity$ Multiply(R).Normalize()$;
Programming Model – Application ex.

Markov Clustering

- Adjacency matrix to represent graphs

MadLINQ.For(0, DEPTH, 1, i =>
{
    // Expansion
    G = G.Multiply(G);
    // Inflate: element-wise $x^2$ and row-based normalization
    G = G.EWiseMult(G).Normalize().Prune();
});
Programming Model – Application ex.

Regularized Latent Semantic Index (RLSI)

- web-mining algorithm to derive approximate topic model for Web docs
- Only 10 LoC while SCOPE's adoption of MapReduce takes 1100+ LoC

MadLINQ.For(0, T, 1, i =>
{
    // Update U
    Matrix S = V.Multiply(V.Transpose());
    Matrix R = D.Multiply(V.Transpose());
    // Assume tile size >= K
    MadLINQ.For(0, U.M, 1, m =>
        U[m, 0] = Tile.UpdateU(S[0,0], R[m,0]));
    // Update V
    Matrix Phi = U.Transpose().Multiply(D);
    V = U.Transpose()
        .Multiply(U)
        .Add(TiledMatrix<double>.EYE(U.N, lambda2))
        .CholeskySolve(Phi);
});
Integration with DryadLINQ

// The input datasets
var ratings = PartitionedTable.Get<LineRecord>(NetflixRating);

// Step 1: Process the Netflix dataset in DryadLINQ
Matrix R = ratings.Select(x => CreateEntry(x)).GroupBy(x => x.col)
    .SelectMany((g, i) =>
        g.Select(x => new Entry(x.row, i, x.val)))
    .ToMadLINQ(MovieCnt, UserCnt, tileSize);

// Step 2: Compute the scores of movies for each user
Matrix similarity = R.Multiply(R.Transpose());
Matrix scores = similarity.Multiply(R).Normalize();

// Step 3: Create the result report
var result = scores.ToDryadLinq();
result.GroupBy(x => x.col).Select(g => g.OrderBy().Take(5));
Fine Grained Pipelining (FGP)

A vertex is read when its each input channel has partial results, execute while consuming input

- Data input/output at finer granularity
- Example, adding matrix A and B:
  - Each divided to 4x4 grid = 16 tiles
  - Each tile is divided to 16 blocks
  - Vertices can stream inputs of blocks of A and B
  - Vertices can stream output of C blocks

The inferior mode of execution:

- *Staged execution: a vertex is ready when its parents have produced all data*
Fault Tolerance Protocol for FGP

Long chain of vertices
Re-execution recomputes all descendants
High overhead
Thus: only recompute need blocks
- Recovering vertex query down-stream for needed blocks
- Request specifically needed blocks from upstream
Evaluation
Effects of FGP and Fault Tolerance

CPU utilization on execution of Cholesky, on 96Kx96K dense matrix, 128 cores (16 nodes)
FGP being 15.9% faster
Effects of FGP and Fault Tolerance

Aggregated network traffic volumes
Pipelined behaves more evenly spread
Effects of FGP and Fault Tolerance

Comparison with ScaLAPACK, dense matrix of 128Kx128K
FGP consistently performs better than ScaLAPACK by an average 14.4%
Real World Applications

Regularized Latent Semantic Index (RLSI)

<table>
<thead>
<tr>
<th></th>
<th>16 nodes</th>
<th>32 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE</td>
<td>6000s</td>
<td></td>
</tr>
<tr>
<td>MadLINQ - FGP</td>
<td>1838s</td>
<td>1188s</td>
</tr>
<tr>
<td>MadLINQ - staged</td>
<td>2053</td>
<td>1260</td>
</tr>
</tbody>
</table>
Real World Applications

Collaborative Filtering

Compared against Mahout over Hadoop

<table>
<thead>
<tr>
<th></th>
<th>M = R x Rᵀ (sparse)</th>
<th>M x R (dense)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mahout over Hadoop</td>
<td>630s</td>
<td>780min (after R was broken into 10, otherwise cannot complete)</td>
</tr>
<tr>
<td>MadLINQ</td>
<td>347s</td>
<td>9.5min</td>
</tr>
</tbody>
</table>
## Related Work

<table>
<thead>
<tr>
<th></th>
<th>Programmability</th>
<th>Execution model</th>
<th>Scalability</th>
<th>Failure-handling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ScALAPACK</strong></td>
<td>Grid-based matrix partition; high expressiveness but difficult to program</td>
<td>Bulk Synchronous Parallel (BSP), one process per node, MPI-based communication</td>
<td>Problem size bounded by total memory size; performance bounded by synchronization overhead</td>
<td>Global checkpointing, superstep rollback and recovery, high performance impact</td>
</tr>
<tr>
<td><strong>DAGuE</strong></td>
<td>Tile algorithm; high expressiveness; programmer must annotate data dependencies explicitly</td>
<td>One-level dataflow at tile level</td>
<td>Problem size bounded by total memory size; performance bound by parallelism at tile level</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>HAMA</strong></td>
<td>Tile algorithm; expressiveness constrained by MapReduce abstraction</td>
<td>MapReduce; implicit BSP between map and reduce phases</td>
<td>No constraint on problem size; performance bounded by BSP model</td>
<td>Individual operator rollback at tile granularity</td>
</tr>
<tr>
<td><strong>MadLINQ</strong></td>
<td>Tile algorithm in modern language; high expressiveness for experimental algorithms</td>
<td>Dataflow at tile level, with block-level pipelining across tile execution</td>
<td>No constraint of problem size; performance bounded by tile-level parallelism, improved with block-level pipelining</td>
<td>Precise re-computation at block granularity</td>
</tr>
</tbody>
</table>
Criticism

Prototype Software
Heavy configuration on parameters and settings
Parallelism depends on well tile-algorithms
Not having a solid benchmark
DryadLINQ no longer active
Future Work
Future Work

Auto-tiling
- Vertex is currently pipelineable *iff* it represents a tile algorithm
- Currently done manually

Dynamic re-tiling/blocking
- Matrices may evolve and require different block and tile size

Sparse matrices
- Handling sparse matrix is still difficult
- non-zero distribution causes laud imbalance