MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud

Zhengping Qian[†] Xiuwei Chen[†] Nanxi Kang[♮] Mingcheng Chen[♮] Yuan Yu[‡]
Thomas Moscibroda[†] Zheng Zhang[†]

[†] Microsoft Research Asia, [♮] Shanghai Jiaotong University, [‡] Microsoft Research Silicon Valley

Abstract

The computation core of many data-intensive applications can be best expressed as matrix computations. The MadLINQ project addresses the following two important research problems: the need for a highly scalable, efficient and fault-tolerant matrix computation system that is also easy to program, and the seamless integration of such specialized execution engines in a general purpose data-parallel computing system.

MadLINQ exposes a unified programming model to both matrix algorithm and application developers. Matrix algorithms are expressed as sequential programs operating on tiles (i.e., sub-matrices). For application developers, MadLINQ provides a distributed matrix computation library for .NET languages. Via the LINQ technology, MadLINQ also seamlessly integrates with DryadLINQ, a data-parallel computing system focusing on relational algebra.

The system automatically handles the parallelization and distributed execution of programs on a large cluster. It outperforms current state-of-the-art systems by employing two key techniques, both of which are enabled by the matrix abstraction: exploiting extra parallelism using fine-grained pipelining and efficient on-demand failure recovery using a distributed fault-tolerant execution engine. We describe the design and implementation of MadLINQ and evaluate system performance using several real-world applications.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—*Distributed programming*

General Terms Design, Performance, Reliability

Keywords Matrix Computation, Distributed Systems, Cluster Computing, Pipelining, Fault-tolerance, Dataflow

1. Introduction

Distributed execution engines (MapReduce [19], Hadoop [2], or Dryad [23]) and high-level language support (Pig [30], HIVE [3], and DryadLINQ [36]) have been widely adopted with great success in the development of large-scale, distributed data-intensive applications. Two factors largely account for the success of these systems. First, they provide programmers with easy access to a core subset of relational algebra operators, such as filtering, projection, aggregation, sorting and joins, and allow further extensions via arbitrary user-defined functions. This pragmatic strategy addresses the critical need to deal with the large corpus of Web data. Second, they adopt the direct-acyclic-graph (DAG) execution model, which is both more scalable and failure-resilient compared to alternative parallel-computing paradigms, such as SPMD (Single Process, Multiple Data).

On the other hand, the relational algebra semantics supported by these Web-scale distributed systems is ill-suited to efficiently solve a large class of important problems which require a deeper analysis or manipulation of the data at hand. In such cases, analysis tools involving linear algebra and matrix computations are often called for instead. Machine learning applications, for example, routinely require matrix computations such as multiplication, Cholesky factorization, singular value decomposition (SVD) or LU factorization [21]. The same is true for sophisticated ranking or classification algorithms. And many algorithms commonly used in, say, social web mining or information retrieval on microblogs boil down to traversal-based graph algorithms (e.g., Betweeness Centrality (BC) [20], PageRank, Breadth-First Search (BFS)) that are also essentially sparse matrix computations.

Given the importance of matrix computation, it is therefore logical to design a scalable engine for linear algebra (which could naturally accommodate many important graph algorithms as well). Ideally, such an engine would achieve the following properties: it should allow developers to easily program matrix algorithms and naturally exploit matrix specific optimization, while at the same time maintaining the system scalability and robustness of a DAG execution

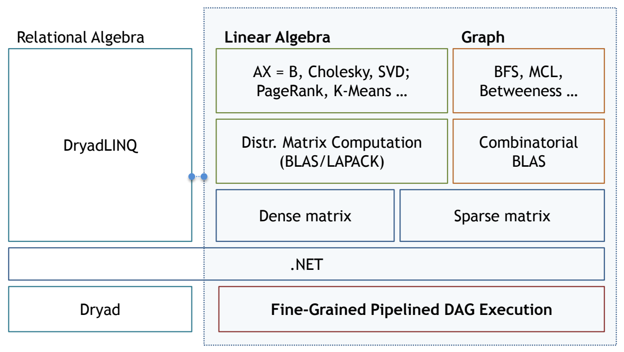


Figure 1. The MadLINQ system stack, and how it interacts with DryadLINQ.

model. Furthermore, it should unify with other, existing engines to deliver a holistic and seamless development experience, ideally in a modern programming language.

Unfortunately, existing solutions fall short in achieving these properties. On the one hand, efficient matrix computation has traditionally been the realm of High-Performance Computing (HPC), with well-known solutions such as ScaLAPACK [17]. These systems require deep understanding of low level primitives such as MPI abstraction to develop new algorithms. The execution model is SPMD, with coarse-grained bulk synchronizations (i.e., barriers). Furthermore, the entire problem must be brought into and efficiently maintained in memory. These constraints severely affect programmability, scalability and robustness and as a result, HPC solutions are rarely considered in Web-scale big data analysis. On the other hand, there have been attempts to implement matrix operations on top of the MapReduce framework (e.g., HAMA [32]). While this removes the constraint of problem size (often referred to as *out-of-core* computing), the MapReduce interface is fundamentally restrictive, making it difficult to program efficient real-world linear algebra algorithms that are structurally far more complex than typical MapReduce jobs. While the execution model is dataflow, MapReduce is implicitly globally synchronized (i.e., no reducers can proceed unless all mappers complete). Further, as we will show later, it fails to take advantage of the well-defined semantics of matrix operations to implement sophisticated optimizations such as pipelining.

These observations motivated us to develop MadLINQ, a system that achieves all the aforementioned desirable properties. Fig. 1 gives the conceptual overview of the MadLINQ system stack. At the bottom is its fine-grained pipelining protocol that exploits the specific structure of matrices, and serves as the fabric to compose the DAG execution on a cluster of machines. Machine-level computation is carried out by industrial-strength multi-core-ready matrix libraries. This execution layer handles both dense and sparse matrix data models, adaptively choosing the appropriate library. Further up the stack, we have developed a large fraction of the traditional linear algebra routines.

In summary, we make the following contributions:

- We introduce a simple programming model for describing matrix computations, based on the familiar tile abstraction in linear algebra. The model supports both dense and sparse matrix data schema, and can easily be used to implement complex matrix algorithms. This flexibility allows us to rapidly develop highly compact programs, covering not only linear algebra routines and graph algorithms, but also new domain-specific algorithms.
- We develop a new *fine-grained pipelining* (FGP) execution model. Unlike existing DAG engines such as Dryad, FGP exchanges data among computing nodes in a pipelined fashion to aggressively overlap computation of depending vertices. As a result, MadLINQ's performance is competitive and often better than mature, highly-customized MPI-based products. For instance, MadLINQ outperforms ScaLAPACK on a 128-node (512-core) cluster, for the standard benchmark of Cholesky by as much as 31.6%.
- We design and implement a lightweight fault-tolerance protocol for FGP, which reduces redundant computation in case of failure to the theoretical minimum. We then show how matrix computation can exploit this to be highly performant and failure resilient. In our test, the system sustains massive failure of machines and arbitrary additions and/or removals of machines, whereas ScaLAPACK cannot withstand any single failure. [1]
- We adopt the language integration approach advocated by LINQ to integrate the domain-specific runtime into a general purpose high-level programming model. This approach allows us to seamlessly combine MadLINQ with other data processing systems such as DryadLINQ, and thus provide the functionality of relational algebra, linear algebra and graph algorithms in one unified platform.

The rest of the paper is organized as follows. Section 2 gives background of matrix algorithms and their building blocks. Section 3 describes MadLINQ's programming model. Section 4 details our design, including DAG generation, execution, and failure handling. We present evaluation results in Section 5 and discuss related work in Section 6. We conclude with a discussion of lessons learned and future work in Section 7.

2. Background & Preliminaries

As mentioned in the introduction, existing large-scale distributed computing engines are typically tailored to solve problems that require relational operators. On the other hand, matrix operations used in many modern data mining or inferencing algorithms are typically hard to capture using such operators.

[1] ScaLAPACK can be made fault-tolerant by invoking global checkpointing only at the cost of a large performance hit; MadLINQ outperforms ScaLAPACK while at the same time being fault-tolerant.

In fact, systems research work that deals with scalable and fault-tolerant matrix computation has often downplayed the complexity and challenge of this problem. To give just one example, an algorithm such as PageRank, which is often used for evaluating such systems, is already quite hard to implement efficiently on MapReduce. But, real-world matrix algorithms are often even much more complicated in their structure, often containing multiple iterations chained together, each of which possibly being multi-level nested; and engaging multiple operators (e.g., addition, multiplication, factorization, etc.). As a result, while PageRank may be harder to efficiently implement in MapReduce than WordCount or other typical MapReduce jobs, implementing complex data mining algorithms in the MapReduce framework becomes exceedingly hard.

Cholesky Factorization: To illustrate these points, consider a matrix operations such as *Cholesky factorization* [21]. Cholesky factorization is a highly efficient way of solving linear equations, and it is one of the most useful and well-studied linear algebra kernels, with a wide range of important applications, including the solving of partial differential equations, Monte Carlo simulation, non-linear optimization and Kalman filters. As an additional example, the Cholesky factorization is also used in one of the real-world applications that had originally inspired this work, and which we use in the evaluation: the computation of topic models of Web documents (RLSI, see Section 3). Technically, if A is a symmetric and positive definite matrix, then the equation $A \times x = b$ can be solved by first computing the Cholesky factorization $A = L \times L^T$, then solving $L \times y = b$ for y, and finally solving $L^T \times x = y$ for x. As we show in Section 3, the Cholesky factorization kernel highlights some of the key characteristics and typical behaviors of the type of operations used by many experimental matrix algorithms. It has become a de facto benchmark in the HPC community (e.g., [9]), and we will also use it to illustrate the key features of our system design.

Tile Algorithms: As with most matrix algorithms, advanced implementations of Cholesky use the concept of *tiles*. A tile is a square sub-matrix, and the entire matrix is divided into a grid of tiles and indexed appropriately. When extending to cover sparse matrices, tiles of rectangular shape may be used for load-balancing purposes. The concept of tiling (or blocking) has a long history, adopted widely in multi-core-ready math libraries such as Intel's MKL[1]. The idea is to leverage the structured access pattern of matrix computation to maximize cache locality. MadLINQ makes use of this structural characteristic of matrix algorithms and explicitly supports tile algorithms in an efficient way.

Fig. 2 (a), (b), and (c) show the tile algorithms of PageRank (for one iteration), matrix multiplication and Cholesky, respectively. Our aim in showing these algorithms in tile form is to give the reader a feel for the relative structural complexity of the kind of matrix operations commonly found in modern machine learning algorithms. The pseudocode illustrates that a matrix operation like Cholesky has significantly higher structural complexity compared to PageRank. The operations `DPOTRF`, `DTRSM`, `DSYRK` and `DGEMM` used by Cholesky are standard math operations, and are implemented in all of the well-known matrix libraries (e.g., BLAS [25], LAPACK [7]).

Specifically, a run of Cholesky factorization on a matrix divided into a grid of n^2 tiles involves the following steps. In the k-th iteration, the algorithm first performs Cholesky factorization on the k-th diagonal tile (routine `DPOTRF`), and then the $n - k$ column tiles below (`DTRSM`), with a parallelism of $O(1)$ and $O(n)$, respectively (see Fig. 3). This is then followed by the updating of the trailing tiles to the right (`DSYRK` and `DGEMM`), with a parallelism of $O(n^2)$. This discussion highlights another common feature of complex matrix operations: the non-trivial interplay of phases of high parallelism and low parallelism. Thus, in Cholesky, the diagonal and panel tile updates have low parallelism, and the same is true in later iterations. MadLINQ is efficient in utilizing the available opportunities for parallelization by using pipelining.

```
for (int i = 0; i < n; i++)    // R: n-tile long column vector of ranks
  R[i] = 0;                    // P: n x n-tile (adjacent) matrix of page links
  for (int j = 0; j < n; j++)
    R[i] += (P[i, j] * R[j]);
```

(a) PageRank as matrix-vector multiplication

```
for (int i = 0; i < m; i++)
  for (int j = 0; i < k; j++) // A: m x n-tile, B: n x k-tile matrix
    C[i, j] = 0;              // C: m x k-tile result matrix
    for (int l = 0; l < n; l++)
      C[i, j] += (A[i, l] * B[l, j]);
```

(b) Matrix-matrix multiplication

```
for (int k = 0; k < n; k++)
  A[k, k] = DPOTRF(A[k, k]);           // A: n x n-tile symmetric positive
  for (int l = k + 1; l < n; l++)      //    definite (input) matrix
    A[l, k] = DTRSM(A[k, k], A[l, k]);
  for (int m = k+1; m < n; m++)
    A[m, m] = DSYRK(A[m, k], A[m, m]);
    for (int l = m+1; l < n; l++)
      A[l, m] = DGEMM(A[l, k], A[m, k], A[l, m])
```

(c) Cholesky factorization

Figure 2. Tile algorithm for PageRank (one iteration), multiplication and Cholesky factorization. `DPOTRF`, `DTRSM`, `DSYRK` and `DGEMM` are standard tile operators, respectively.

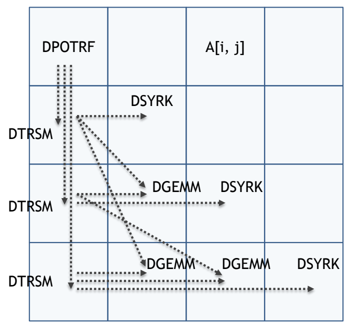


Figure 3. One iteration of tiled Cholesky factorization.

3. Programming Model

MadLINQ embeds a set of domain-specific language constructs into a general-purpose programming language (C#), similar to the approach taken by DryadLINQ and FlumeJava [15] for data-parallel programming. This embedding allows us to expose a unified programming model for developing both matrix algorithms and applications. In particular, it allows us to integrate MadLINQ with DryadLINQ, making it easily accessible to a general purpose data-intensive computing system. In this section, we provide a high-level view of the programming model, using real-world applications as examples to highlight its key aspects. We will evaluate the performance of MadLINQ using these examples in Section 5, and contrast our experience programming using the popular MapReduce paradigm.

3.1 Programming Language

The new domain-specific language constructs are designed to express matrix algorithms efficiently with familiar notations. The key data abstraction is `Matrix`, which, simply defined as a C# class, encapsulates the tile representation of a matrix. Matrix computations are then expressed as a sequence of operations on tiles. For example, the following is the MadLINQ code for the tile-based matrix multiplication algorithm:

```
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]);
  });
});
```

The language allows even complex algorithms to be expressed naturally. For example, the MadLINQ implementation of Cholesky is almost a line-by-line translation of its tile algorithm [9] given in Fig. 3 and discussed in Section 2. Note that `DPOTRF` is a LAPACK's name for Cholesky, and it is applied to the diagonal tile in the beginning of each outer iteration.

```
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]));
  });
});
```

The example shows that programming a tile algorithm in MadLINQ is straightforward, literally a direct translation of the algorithm into a sequential program. That is, the code looks simple even though (as discussed in the previous section) the computational structure of Cholesky is complex. We have also implemented singular value decomposition (SVD), which involves three nested iterations, with the innermost body itself containing two back-to-back loops. It would be challenging to program such algorithms in a MapReduce-style programming environment.

3.2 Programming Applications in MadLINQ

We have developed a library including the core set of linear algebra routines; the library is a set of C# methods and can be easily extended. To check the usefulness of the MadLINQ programming model, we implemented several real-world applications, three of which we will describe in detail here as they form the basis of our evaluation.

Collaborative Filtering (CF): We implement the baseline algorithm of collaborative filtering [11] and evaluate it using the data set from the Netflix challenge [8] in Section 5. In that data set, the matrix R records users' rating on movies, with $R[i, j]$ being user j's rating on movie i. So $R \times R^T$ gives us the similarity between the movies, and multiplying the result with R again yields the predicted ratings of all movies for each user. Matrix R is sparse while matrix $score$ is dense. A final normalization step normalizes the scores.

```
Matrix similarity = R.Multiply(R.Transpose());
Matrix scores = similarity.Multiply(R).Normalize();
```

To illustrate the difference in programming style and effort between implementing CF in MadLINQ and MapReduce, we provide a discussion in Section 3.4.

Markov Clustering: MadLINQ supports both dense and sparse matrices. Since a graph can be represented as an adjacency matrix, we can naturally implement many graph algorithms, including Breadth-First Search, PageRank and Approximate Betweeness Centrality [10, 20] and Markov Clustering (MCL) [34] which we discuss below.

```
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();
});
```

Unlike clustering such as K-means that requires the number of clusters as a parameter, the clusters are derived from the underlying graph structure. The algorithm operates over a single adjacency matrix A, with non-zeros in the i-th column identifying the set of nodes connected to node i. There are two phases in each iteration. The expansion phase performs an in-place update to A with $A \times A$. The net effect is that the updated $A[i, j]$ has greater value if j can reach i through more paths. This is then followed by an inflation phase, which raises the power of each column, normalizes it, and finally prunes the smaller entries. Therefore, strongly connected nodes gradually cluster together.

Notice that in addition to multiplication, our MCL implementation demonstrates the use of two APIs from the Combinatorial BLAS [12] library, whose basic data structures are sparse matrices: element-wise multiplication to raise the power (`EWiseMult`) and pruning (`Prune`) to cut low-strength edges.

Algorithm 1 Regularized Latent Semantic Indexing

```
Require: $\mathbf{D} \in \mathbb{R}^{M \times N}$
1: $\mathbf{V}^{(0)} \in \mathbb{R}^{K \times N} \leftarrow$ random matrix
2: for $t = 1 : T$ do
3:     $\mathbf{U}^{(t)} \leftarrow \text{Update}\mathbf{U}(\mathbf{D}, \mathbf{V}^{(t-1)})$
4:     $\mathbf{V}^{(t)} \leftarrow \text{Update}\mathbf{V}(\mathbf{D}, \mathbf{U}^{(t)})$
5: end for
6: return $\mathbf{U}^{(T)}, \mathbf{V}^{(T)}$
```

Algorithm 3 Update**V**

```
Require: $\mathbf{D} \in \mathbb{R}^{M \times N}$, $\mathbf{U} \in \mathbb{R}^{M \times K}$
1: $\mathbf{\Sigma} \leftarrow \left(\mathbf{U}^T\mathbf{U} + \lambda_2 \mathbf{I}\right)^{-1}$
2: $\mathbf{\Phi} \leftarrow \mathbf{U}^T\mathbf{D}$
3: for $n = 1 : N$ do
4:     $v_n \leftarrow \mathbf{\Sigma}\boldsymbol{\phi}_n$, where $\boldsymbol{\phi}_n$ is the $n^{th}$ column of $\mathbf{\Phi}$
5: end for
6: return $\mathbf{V}$
```

Figure 4. Kernel routines of Regularized Latent Semantic Index algorithm.

Regularized Latent Semantic Index (RLSI): Our last example is RLSI [35], a new Web-mining algorithm. The goal of the algorithm is to derive an approximate topic model for Web documents. Unlike the more expensive SVD-based topic model and assuming that the topics are sparse, RLSI relies on a series of cheaper operations to factorize the matrix. The algorithm operates over a giant sparse matrix (D) and a (skinnier but also giant) dense matrix (V). D is a doc-to-term matrix, whereas V records the mapping of doc-to-topic. The structure of the algorithm is similar to non-negative matrix factorization (NMF) [26], kernel routines of which are outlined in Fig. 4. Note that the algorithm calls Cholesky factorization as a subroutine when updating V. Interestingly, while the MadLINQ code is about 10 LoC, an implementation in SCOPE [13] requires 1100+ LoC, which is to a large extent due to SCOPE's adoption of MapReduce to describe the algorithm (see Section 3.4). Working together with the original designers of the RLSI algorithm, it took us a single man-day to write the code. As we will report in Section 5, MadLINQ is also significantly faster than the MapReduce-based implementation.

```
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);
});
```

Finally, notice that `Tile.UpdateU` is a user-defined operation on tiles which can be implemented using the extension interface MadLINQ provides.

3.3 Integration with DryadLINQ

We have shown in the previous section how to implement matrix computations in MadLINQ. One strength of our approach is that by embedding the language (and hence the matrix library) in C#, it is natural and easy to inter-operate with other data processing systems such as DryadLINQ. This seamless integration of MadLINQ + DryadLINQ + C# provides a unified and elegant solution to many real-world problems in which certain parts of the computation are naturally handled using relational algebra operators; whereas other parts of the computation require matrix operations. To illustrate these two disparate styles of data analysis/manipulation, consider the following collaborative filtering example. In this example, the input is Netflix data, the output is a recommendation for a movie for each user.

```
// 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();
  .GroupBy(x => x.col)
  .Select(g => g.OrderBy().Take(5));
```

The above code shows the two systems DryadLINQ and MadLINQ inter-operate, each system doing the part of the computation for which it is suited. The initial data $ratings$ is from Netflix and is represented as a text file. There are three steps. In Step 1, DryadLINQ is used to process the input data into the matrix representation accepted by MadLINQ. It boils down to creating matrix R to represent the movie-rating relations. In Step 2, MadLINQ is called to perform the collaborative filtering on matrix R, which is more suitable to do in MadLINQ. In Step 3, after the MadLINQ computation completes, DryadLINQ is used again to create a report that recommends the top 5 movies for each user.

This example highlights MadLINQ's philosophy of preserving a unified programming experience at the surface, while calling into different domain engines to leverage their strengths. Steps 1 and 3 are best handled by relational algebra engines such as DryadLINQ, whereas Step 2 is a linear algebra routine best handled by MadLINQ.

3.4 Alternative using MapReduce

In Section 3.2, we described the programming of matrix algorithms in MadLINQ. It is insightful to compare this to programming the same algorithms in MadReduce. In principle, collaborative filtering (i.e., computing $R \times R^T \times R$ of a matrix R) is sufficiently simple to be expressed in MapReduce. The Apache Mahout [4] project (a comprehensive machine learning package) includes a variant of the algorithm, and also provides an implementation of matrix multiplication. Briefly, a matrix in Mahout is an HDFS file that stores non-zero elements. The file is keyed by the row index, and is a collection of rows. Each row contains a list of tuples, and each tuple is a pair, the column index and the value of the corresponding non-zero entry. In order to apply the MapReduce APIs to perform a matrix multiplication $A \times B$, the mapper takes the i's column of A, multiples it with the i's row of B, and produces a partial matrix. The reducer then reduces on the entry (i, j), aggregates all the partial matrices, and outputs the final matrix. Since the matrix is stored as a set of rows, the first matrix A needs to be transposed first. This is accomplished by another job, whose mapper breaks the list of non-zeros and output tuples that are now keyed by column index, and the reducer aggregates on the column index to produce A^T. Thus, the MapReduce version of the operation $R \times R^T \times R$ used by our CF takes four MapReduce jobs. In our evaluations, we will show that while it performs reasonably well for the multiplication of two sparse matrices (as in $R \times R^T$), MapReduce/Mahout faces severe problems if one of the matrices becomes dense (e.g., the second multiplication). Essentially it requires a different algorithm whose execution plan is difficult to be expressed in MapReduce.

The situation is similar for the RLSI algorithm. Consider $D \times V^T$ (in `UpdateU`, line number 3 of Fig. 4). Both matrices are big, but D is sparse and V is dense. This turns out to be the most time-consuming step of the algorithm. MadLINQ implements this with a single line: `D.Multiply(V.Transpose())`. In an implementation using SCOPE, since V^T is too large to fit into memory, following the same procedure as in [26], the authors structured the computation in two MapReduce phases. The first one generates many sparse matrices by multiplying a column of D with a row of V^T, and the second one aggregates them up, which is essentially the same as in the collaborative filtering application and faces similar problems.

4. System Design and Implementation

We first give an overview of MadLINQ's architecture, and then focus on the key features of our design. In Section 4.2, we describe a fully automatic DAG generation scheme for tiled matrix algorithms that uses in-flight symbolic execution to avoid the problem of DAG-size explosion. Section 4.3 introduces our distributed DAG-based execution engine. The engine is non-blocking, and enables *automatic fine-grained pipelining* (FGP) of the computation, thus leveraging extra parallelism. Section 4.4 describes our novel fault-tolerance protocol, which reduces I/O, and handles failures in a highly-efficient way. Finally, we end the section with noteworthy optimizations.

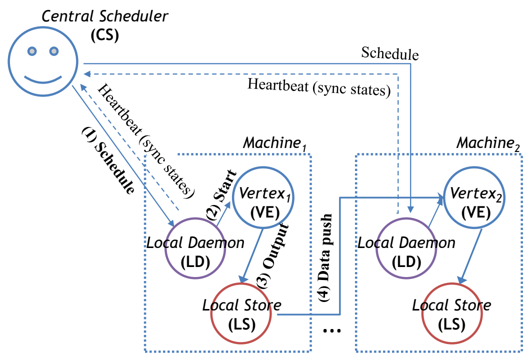


Figure 5. MadLINQ system architecture. The system consists of a Central Scheduler, and a Local Daemon, a Local Store and a Vertex Engine on each compute node.

4.1 System Architecture

As shown in Section 3, MadLINQ programs are written using ordinary C# programs with our embedded language and data types (e.g., `Matrix`, `Tile`). The translation of a program into a DAG is discussed in Section 4.2. At runtime, the execution flows into the MadLINQ system. Fig. 5 shows the flow of execution when a job is entered into the MadLINQ system.

The MadLINQ distributed runtime consists of a central scheduler (CS) running on a server node and three processes, a local daemon (LD), a local store (LS), and a vertex engine (VE), all running on the compute nodes in the cluster. The central scheduler receives a job submitted from a client and schedules it on the compute cluster (Step 1). It also monitors the current utilization levels and availabilities of all the compute nodes. On a compute node, the local daemon is responsible for starting the vertex engine (Step 2). It is also responsible for periodically sending progress reports of the vertex execution to CS.

The vertex engine executes the vertex code corresponding to the program DAG. Each compute node is preloaded with the MadLINQ runtime which includes the full set of math libraries by which the actual computation is carried out. We carefully designed the interface such that we can call any of the state-of-art math libraries (e.g., MKL). This is an important design decision because a high-quality, domain-specific math library is orthogonal to MadLINQ's system architecture, but important for its performance.

Finally, the local store manages the output of the vertex (Step 3), and is responsible for pushing the data to a downstream vertex, possibly in a pipelined fashion (Step 4).

CS assigns a static priority to each vertex based on a critical path analysis of the program. It maintains a priority queue of all ready vertices and schedules the vertices based on their priorities. In systems such as Dryad and MapReduce, a vertex is considered to be ready when its parents have produced all the data. We call such execution *staged*. The default execution in MadLINQ is *pipelined*, a vertex is ready

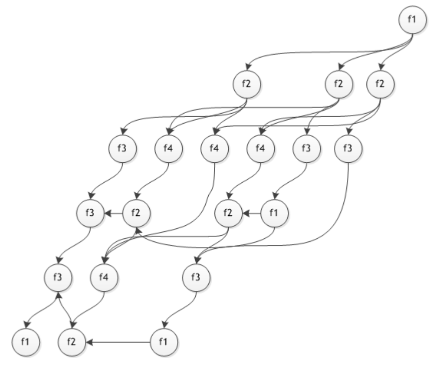


Figure 6. DAG of Cholesky factorization algorithm for a 4×4 tiled matrix (where $f1$ through $f4$ are the tile operators: `DPOTRF`, `DTRSM`, `DSYRK` and `DGEMM`).

when each input channel has partial results, and it can start executing while consuming additional inputs (Section 4.3).

When scheduling a ready vertex, CS takes into account data locality, assigning the vertex to the machine(s) where its inputs are produced when possible. De(re)-scheduling of a vertex may happen when it is done, crashed during execution, or timed out while waiting for input (e.g., a parent crashed) The local daemon reports back to CS periodically the progress of the running vertices so that CS can make informed scheduling decisions. When a vertex fails, CS runs the fault-tolerance protocol (Section 4.4), fixes its state, and schedules it again.

4.2 DAG Generation and Vertex Initialization

The scheduler keeps in memory the list of running vertices and their immediate children, a subset of these children are also kept in the ready queue. Collectively, they comprise the frontier that the execution is exploring. This frontier is dynamic: as a ready vertex becomes running, its children need to be included in the list. Doing so requires consulting the DAG.

Keeping the DAG in memory can be cumbersome and unscalable. This is especially true when the matrix is big and number of iterations (often nested) is high. If the matrix is divided into $n \times n$ tiles, then the DAG size of multiplication and Choleksy is $O(n^3)$, and for Jacobi-based SVD the DAG size is $O(n^4)$. Therefore, in anticipation of large problem size, we need to deal with the issue of *DAG explosion*.

We *dynamically expand* the DAG through symbolic execution. Given the loop boundaries, we can symbolically execute the program. Each statement in the loop body is decomposed into an expression tree, where the inputs are tiles that are labeled according to iteration number. During the exploration, each operator is uniquely identified by the order it is visited and becomes a vertex. Finally, vertices are connected by data dependencies identified by the label of the tiles. Fig. 6 visualizes the DAG of Cholesky for a matrix divided into 4×4 tiles.

This DAG exploration also performs several additional tasks. First, it discovers and assigns vertex priorities according to their positions in the DAG topology. Second, it identifies the type of computation a vertex is to perform, allowing the vertex engine to call appropriate routine (e.g., addition). Finally, it computes the set of blocks (needed for pipelining, explained shortly) symbolically. Those metadata are needed for scheduling and failure handling.

In the implementation, this is done by abstracting the necessary APIs when we need to consult the DAG. For instance, when adding new vertices to the frontier, `GetChild(v)` will return the list of child vertices of a given vertex v, which are then initialized and put into the list. A few other APIs return parent vertices, which are needed in handling failures. These APIs are implemented by executing the program symbolically described above. This approach completely removes the need to keep a materialized DAG, with negligible overhead.

4.3 Fine-Grained Pipelining

In many matrix computations, available parallelism of an algorithm fluctuates. In Cholesky factorization, for example, the bulk of parallelism comes from multiplications in the trailing tiles, which reduces quickly in later iterations (see Section 2). When vertex-level parallelism is low and if there are spare resources, pipelining can effectively explore inter-vertex parallelism. Another benefit is that network utilization becomes less bursty. FGP opportunistically exploits such inter-vertex pipelining to improve performance. [2]

Pipelining requires each vertex to consume and produce data at a finer granularity, which we call a *block*. As a trivial example, suppose we are adding two matrices A and B, each is divided into a 4×4 grid, for a total of 16 tiles. Each tile is recursively divided into 16 blocks, then each of the 16 addition vertices can stream in blocks of its corresponding A and B tile, and similarly output C blocks, all in a pipelined fashion.

In the above example, pipelining means applying the tile algorithm at the block level. More specifically, it requires that 1) the vertex computation be expressed as a tile algorithm and 2) the vertex execution engine can perform the computation incrementally, i.e., computing and outputting partial results. For some matrix algorithms, the vertices are themselves tile algorithms. This is true for simple computation such as multiplication, and for some more sophisticated algorithm such as Cholesky as well. If the vertex is not a tile algorithm, its execution falls back to staged execution, exposing the entire tile only upon termination. Additionally, we provide annotation to allow developers to convert to tile algorithm manually; automatically transforming

[2] Note that another possible technique to address the parallelization problem is to reduce the tile size, as this allows to decrease the amount of time the computation spends in low parallelism phases. However, this approach has its natural limits, as doing so substantially increases the number of tiles and hence the system overhead since the DAG size is O(n^3).

an algorithm into tile algorithm is an active research area in the HPC community.

When an input arrives, the vertex engine checks if any computation can be done, and if so it calls a math library (e.g., Intel MKL) to carry out the execution. We keep the intermediate results as context for reuse and remove this context immediately after all dependent output blocks have been generated. Conceptually, the local vertex execution is dataflow-driven, and follows the the same symbolic execution framework as in the CS (see Section 4.2).

4.4 Handling Failure

Pipelined DAG execution is not a new idea. However, providing fault-tolerance in pipelined DAG is non-trivial, even when each vertex's computation is deterministic (which is the case for MadLINQ). To see why, consider a long chain of vertices. If any one of these vertices fails, its re-execution will re-compute blocks that its immediate downstream vertex has already consumed. This has the cascading effect of triggering unnecessary computation in all the descendants. For this reason, without careful bookkeeping the overhead can be non-trivial and unbounded.

For this reason, existing solutions such as TCP streaming in Dryad [23] or streaming support in CIEL [29] choose a different strategy. If any vertex fails, the sub-DAG of which the failed vertex is a root is failed together. This is not an appropriate solution for MadLINQ, because the entire program is pipeline-enabled, and the running vertices may indeed be forming a connected sub-DAG. Thus, adopting these existing strategies would mean that we potentially fail all running vertices. The novel contribution of FGP is that we only do a minimal recomputation, by using lightweight dependency tracking. Furthermore, the protocol can withstand an arbitrary number of fail-stop failures. As we will demonstrate later, such capacity can be used to dynamically size resources, which is an important scenario in the Cloud.

The core idea behind FGP's failure handling is simple. Recall that the input to a vertex is always a set of matrix blocks, from which the vertex computes another set of blocks as its output. Our fault-tolerance mechanism depends on the crucial assumption that for any given set of output blocks S we can automatically derive the set of input blocks that are needed to compute S. Such a backward slicing [31] technique is possible in our system because (rather than using the compiler to determine the data slice), we derive the dependency at the time when we perform symbolic DAG generation. As an example, for a 4×4 matrix multiplication, we derive that $C[0, 0]$ depends on $A[0, 0 : 3]$ (A's first row) and $B[0 : 3, 0]$ (B's first column) at the time when we symbolically execute the multiplication of C.

This dependency calculation enables us to minimize the recovery cost of a failed vertex by re-computing only the needed blocks. At a high level, the recovery procedure is therefore as follows. A recovering vertex queries its downstream vertices for blocks they still need in order for them to

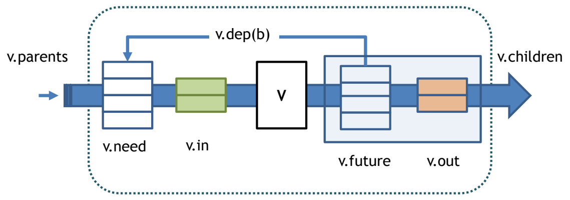


Figure 7. States to describe a vertex for pipelined execution and failure handling.

complete. The union of all blocks needed by its downstream vertices is what the new recovering vertex needs to compute, and from which we apply our dependency analysis to derive the set of input blocks that the recovering vertex needs. The recovering vertex then asks for those input blocks from its upstream vertices. This process can be recursive, in the sense that if the upstream vertex has also crashed or cannot send the missing blocks to this vertex, it too will invoke the same process to request needed blocks from upstream vertices.

We now proceed to give a rigorous description of the FGP protocol. The system-wide invariant is that, as long as 1) there is at least one computing node available and 2) the original input data is available, the execution will terminate correctly.

The states required by the protocol are shown in Fig. 7. $v.in$ specifies the blocks that are available in v's input buffer, $v.need$ specifies the set of blocks that v still needs to complete its computation,$v.future$ identifies the blocks that still need to be computed, and $v.dep()$ is a function that computes the set of input blocks needed for a given set of output blocks. $v.dep()$ is the *inverse* of v's program, and can typically be derived for matrix computation automatically.

Ensuring correct termination can be decomposed into keeping two invariants. The first invariant binds the relationship of buffers inside a vertex. Simply put, what a vertex needs is anything that it needs to produce new blocks, minus what it already has in its input buffer.

(1) $v.need = v.dep(v.future) - v.in$

The second invariant complements the first, and binds the relationship of outwards-facing buffers across dependant vertices. Most importantly, it specifies what $v.future$ really is. In this invariant, $v.out$ is the set of blocks v has computed and made available to the rest of the system. $v.ALL$ is all the blocks that the vertex is to compute in its lifetime (e.g., $C[0 : 3, 0 : 3]$).

(2) $v.future = v.children.need \cap v.ALL - v.out$

The invariant says that what a vertex needs to produce, is the union of everything to satisfy its children intersected with what this vertex is responsible for (as a child vertex may depend on other vertices), but minus what it has already made available to the children.

During the normal execution of a vertex, the system functions as described in Section 4.3. When a vertex v is initialized in CS using symbolic DAG generation, its $v.ALL$ is computed. Next, $v.need$ is set to $v.dep(v.ALL)$, meaning that it needs all the blocks. The system can schedule any vertex that has data to consume (i.e., $v.need \cap v.parents.out \neq \emptyset$). The vertex then arrives at a computing node, along with $v.need$ and $v.ALL$; $v.in$ and $v.out$ are set to $\emptyset$.

When the vertex is instantiated, it will start exchanging $v.need$ to its parent(s). As blocks arrive, $v.in$ is updated and computation starts. $v.out$ is updated when new outputs are produced. Other fields such as $v.need$ and $v.future$ are updated accordingly.

Active vertices also report $v.out$ back to CS periodically. This allows CS to discover newly enabled vertices, and instruct them where to find their inputs when deployed. More importantly, when failure occurs, CS knows precisely the set of blocks that are now lost. The system restores the required invariants by letting the recovering vertex to query its children for their $need$ set, which is sufficient to compute its own $need$ set using the $v.dep()$ function. Note that the $need$ set is used to request data in normal conditions, and all that is special in failure handling is to set the recovering vertex's various fields appropriately. The process is inherently recursive and converging, and can deal with arbitrary number of failures in arbitrary positions of the DAG.

The same principle is upheld to handle even more complicated cases. For instance, retired vertices (i.e., those who have computed all outputs) are said to be *hibernating* at CS, and in that sense they never truly retire. Also, if any of the child vertices of a retired vertex is requesting blocks that are missing from the system due to failure, the vertex is reactivated since their *future* set is no longer empty.

We also developed a formal specification in TLA+ [24] that significantly increased our confidence of its correctness. The specification is about 200 LoC, and verified with a small DAG using TLC [5].

Our design of FGP is quite general and we believe it is applicable in other contexts as well. For FGP to work, two key conditions must be satisfied. First, as described above we assume that it is possible to infer the set of input blocks that a given output block depends on. When this is impossible, the protocol falls back by assuming any output depends on any input, and thus this particular vertex will be executed in the staged model. Second, we assume that vertex computation is deterministic. Supporting non-determinism can be added by recording the random perturbation as part of the input. We have found that some algorithms require this, e.g., picking a random edge in a graph. These conditions are general and not restrictive to matrix computation, as long as each vertex in the dataflow program can satisfy these two assumptions, it is safe to employ FGP.

4.5 Optimizations

Based on our experience of using the system, we added a number of performance optimizations, and some of the more noticeable ones are described below:

- Pre-loading a ready vertex onto an occupied computing node whose current vertex is about to finish. This prefetches data and helps smoothing network traffic.
- Adding order preference (e.g., row-major, column major or any) when requesting input for a vertex. This is because the pipelined execution is sensitive to the arrival order of input data blocks.
- Auto-switching of block representation depending on sparsity. For sparse matrices, we represent blocks using a compressed-column representation. However, during some intermediate computation blocks may become dense. Therefore, when the number of non-zeros inside a block is larger than a threshold, we switch to dense block representation and invoke the dense math library instead.

5. Evaluation

We present detailed performance results for the applications described in Section 3. We have also implemented a number of additional applications including QR factorization, dense SVD, K-means, PageRank and Betweeness Centrality, but due to space constraints, we can only present results from some representative examples. When possible, we include comparisons against competing alternatives.

We performed the experiments on different computer clusters, all running Windows Server 2008 R2. We use MKL 10.3 which is multi-core-ready to carry out basic matrix operations. Our hardware platform and configurations are typical for Cloud-level offerings: two 1.0TB SATA disks, and 16GB memory, interconnected with 1Gbit Ethernet switches. The CPUs differ slightly, a typical one is dual Intel Xeon CPU L5420 at 2.5GHz, with a total of 8 cores.

5.1 Run Configuration

The run configuration is similar to DryadLINQ. As part of job submission, clients include a configuration file that specifies all the necessary parameters of the matrices, including the location of their tiles, tile and block size. As we also handle sparse matrices (for graph algorithms, for example), block format specification is also included. Choosing the right parameters is a tradeoff between multiple factors:

- Smaller tiles allow higher tile-level parallelism, but increase scheduling overhead. There is also a memory constraint, since even though the vertex engine can perform incremental computation at block granularity, the total working set is typically proportional to the tile size.
- The granularity of computation is a block. Multi-core-ready math libraries such as MKL typically yield better performance for bigger blocks. On the other hand, a smaller block size enables better pipelining. We determine the size of blocks with profiling (for dense matrices).
- For sparse matrices, the block size is determined by the number of non-zeros; we tune it such that this total number is the same as in an appropriate dense block.

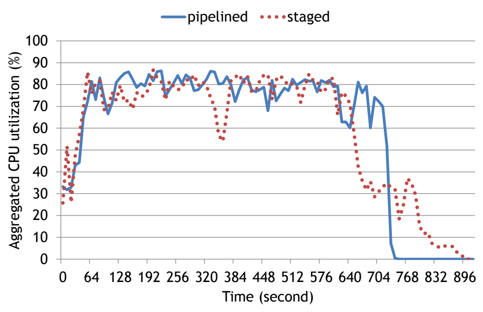


Figure 8. Comparison of aggregated CPU utilization of pipelined and staged executions of Cholesky. 96K × 96K dense matrix, 128 cores (16 nodes).

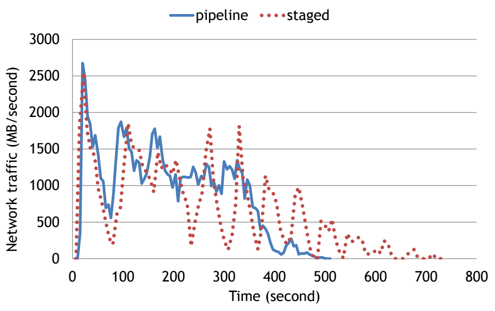


Figure 9. Network traffic comparison for pipelined and staged. Pipelined is more smooth and spread.

5.2 The Effect of Pipelining and Fault Tolerance

We use Cholesky factorization to study the effectiveness of our fine-grained pipelining and fault-tolerance protocol. As described in Section 2, the general pattern is that parallelism fluctuates within an iteration, and the total amount of parallelism decreases with successive outer-loop iterations.

Fig. 8 shows the parallelism curves of two runs, one for the pipelined and another for the staged execution model. The runs are executed on 16 nodes (128 cores) for a problem size of a 96K × 96K dense matrix (roughly 5 billion elements, 36GB), with 8K and 2K as the tile and block size, respectively (364 tile operators, about 167GB intermediate result). The curves show the aggregated CPU utilizations across the entire cluster. Since there is enough parallelism in earlier iterations, both models are equally effective at the beginning, though pipelined is slightly better. As the computation progresses towards later stages, the pipelined model exploits more inter-vertex parallelism and continues to maintain high utilization. In all, the pipelined mode is about 15.9% faster than the staged model.

Pipelining performs better for larger problem sizes and on a larger cluster, as in this case the pipelining can be deeper and there are more spare resources to recruit when vertex-level parallelism is low. For the same problem size on a 256-core cluster, pipelined is 28% faster than staged. One advantage of pipelining is that network traffic is more evenly spread, as blocks start transmitting during the course of a

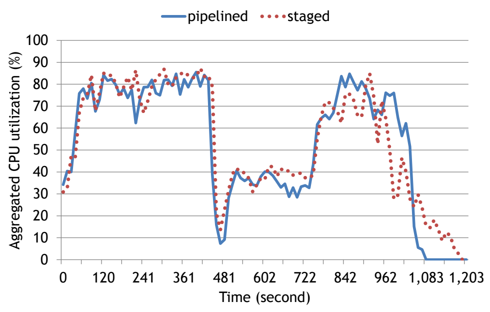


Figure 10. Experiment showing fault-tolerance. We disrupt a run for 5 minutes in the middle of its execution, removing half of the machines and then restoring them. Performance drops, but then recovers accordingly.

vertex's lifetime. Fig. 9 shows the aggregated network traffic volumes of the two runs, it is clear that pipelined behaves in a less bursty pattern.

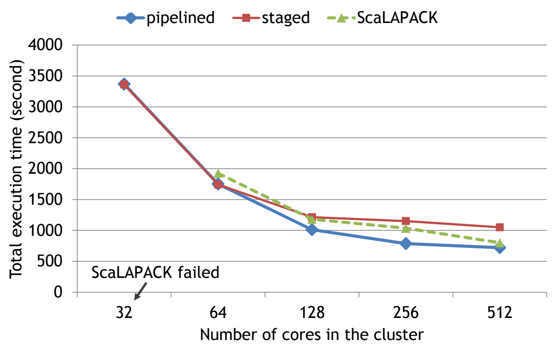
We tested the fault-tolerance mechanism by killing running processes and deleting generated outputs. Fig. 10 shows the extreme case of simultaneously removing half of the machines altogether, and then adding them back after a window of 5 minutes. This emulates the rapid resource fluctuation that may happen in a Cloud environment. As expected, MadLINQ's performance (as indicated by its overall CPU utilization) degrades after resource removal, and then is quickly restored when those resources return.

We performed numerous experiments against ScaLAPACK [17], the best known and widely adopted MPI-based solution. ScaLAPACK is a released product running over Windows HPC Server 2008 R2. Each process owns a partition of the matrix, and communicates with each other using MPI. Barriers are used for global synchronization. It calls exactly the same MKL libraries for local computation within a node, as we do in MadLINQ.

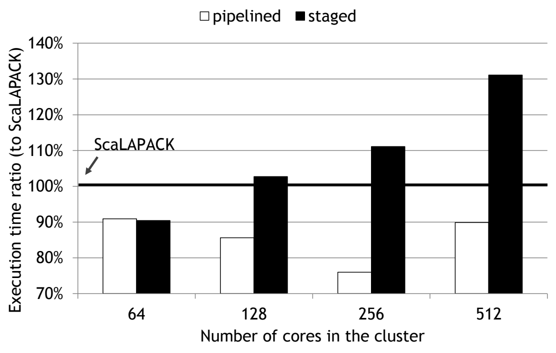
The result is shown in Fig. 11, using a dense matrix of 128K × 128K (64GB, intermediate result approximately 375GB). As expected, the performance difference between pipelined and staged widens as the number of machines increases. With 256-cores, pipelined is 16.6% faster than staged; with 512-cores, the gap widens to 31.6%.

The interesting result is that pipelined consistently outperforms ScaLAPACK by an average of 14.4%. The relative performance of MadLINQ's two models against ScaLAPACK is shown in Fig. 11(b). The gap between pipelined and ScaLAPACK steadily widens as more cores are added, as more resources can be used to fill the valleys. However, with 512-cores, most of the available parallelism is already exploited, and MadLINQ's scheduling overhead somewhat degrades its performance, but still achieves a 10% gain.

Also, despite repeated attempts, ScaLAPACK consistently failed at 32-core because the problem size exceeds the aggregated memory in the cluster. MadLINQ can perform out-of-core computation, and thus removes this constraint and scales easily with different problem sizes; its 32-core



(a) Absolute running time



(b) Relative to ScaLAPACK

Figure 11. Comparison of the time and scalability of pipelined and staged executions with ScaLAPACK (Cholesky of a 128K × 128K dense matrix).

performance is roughly two times slower than that of its 64-core run.

The ScaLAPACK experiment is run without adding any checkpointing facility, as turning it on would significantly hurt its performance, thereby drastically improving MadLINQ's advantage. Consequently, these experiments occasionally failed due to jitters in MPI communication. The fact that MadLINQ, currently still a research prototype, supports fault-tolerance by default and yet competes favorably against an industrial-strength product is very encouraging.

5.3 Real World Applications

RLSI: We compare against a SCOPE-based implementation of RLSI, which consists of 1100 LoC and is run on a production cluster of 16 nodes with a sample of real Web data. The most time and space consuming step is to compute $R = D \times V^T$, where D is a matrix of 7M × 2M with 0.005% sparsity (10.5GB), and V^T is 2M × 500 and dense (7.5GB). This step takes around 6000s.

SCOPE requires two MapReduce jobs to compute the step $D \times V^T$ (see Section 3). In comparison, MadLINQ implements this with a single line of code as follows: `D.Multiply(V.Transpose())`. Good performance requires tuning of parameters, and we give some details here. Simple partitioning yields satisfactory results, but the best

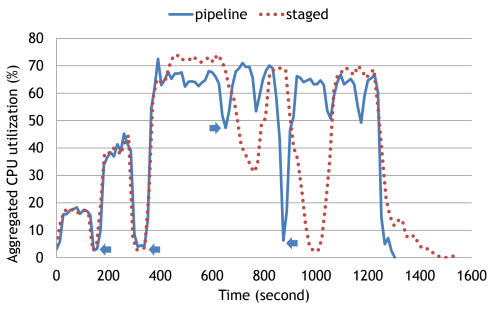


Figure 12. Aggregated CPU utilizations for MCL runs on 64 machines. The valleys correspond to the reduction phase (marked with arrows).

performance is achieved when we partition D into 20 vertical tiles, and V^T into 4 vertical partitions, each of which is then horizontally partitioned into 20 stripes. Such decomposition steps are standard when partitioning for large matrix multiplications. We measured the performance on a 16-node cluster, the result was 1838s, a speedup of more than a factor of 3. Running on a 32-node cluster reduces the time to 1188s. The corresponding numbers for staged is 2053s and 1260s, for 16-node and 32-node runs respectively. This real-world example demonstrates MadLINQ's superior programmability and performance against a MapReduce alternative.

MCL: We use R-MAT [14] to generate a synthetic graph, which models the behavior of several real-world graphs such as the Web graph, small world graphs, and citation graphs. We use the default parameter to generate a graph with half a billion nodes. The node degree ranges between 5 and 50. Thus, after the expansion phase, we choose a pruning threshold such that the average node degree is 50 (about 376GB). The tile size is 32K and the block size is 8K.

With 96 machines, pipelined and staged performs four iterations of MCL for about 1000s and 1172s, respectively. With 64 machines, the number becomes 1300s and 1560s, respectively. The less than perfect scaling is due to the fact that, in each iteration, there is a reduction stage to normalize each column before pruning happens. Fig. 12 shows the two runs' aggregated CPU curves in the case of 64 machines.

We note a few patterns in our results. First, computation progresses in phases. The valleys are the reduction stages, where the degree of parallelism is the lowest. This creates stragglers and resource vacancy, which the pipelined execution effectively exploits. As a result, the valleys of the pipelined run are "thinner". Second, overall utilization rises over time, because the matrix becomes denser over time. This is typical for many iterative graph algorithms, illustrating that a good matrix computation platform must handle both sparse and dense matrices well.

Collaborative Filtering: We evaluate this application with the data set from the Netflix challenge [8]. A 20K × 500K matrix R records user's rating on movies, with $R[i, j]$ being user j's rating on movie i. This matrix has a sparsity of 1.19% (about 2GB). Recall that this is an algorithm where

MadLINQ and DryadLINQ integrate to come up with the final top-5 recommendations to each user (Section 3.3).

The MadLINQ portion first produces a dense 20K $\times$ 20K movie *similarity matrix* $R \times R^T$. The resulting matrix multiplied again with R yields the predicted rating of a user over all movies. The step takes 840s on 48 machines. After that, DryadLINQ takes 334s to rank and produce the recommendations using the same cluster.

We compared the performance of $R \times R^T \times R$ using the multiplication provided by Mahout over Hadoop (version 0.020.203.0), running over Windows. The first multiplication ($R \times R^T$) takes approximately 630s, almost twice as much as MadLINQ (347s). The resulting matrix, call it M, is dense. The second multiplication ($M \times R$) therefore has problems to even complete on Hadoop, despite multiple attempts using different configurations. The reason is that the partial matrices (Sec. 3) are dense, and the total working set is so large ($\sim$ 20TB) that I/O thrashing occurs.

To cope with this problem, we divide R into 10 equally-sized matrices of dimension $20K \times 50K$, and multiply M with each of these sub-matrices. Each of these multiplications takes approximately 78 minutes. Thus, $M \times R$ takes a total of 780 minutes. In contrast, MadLINQ's second multiplication is only 9.5 minutes. In performing this step, MadLINQ's optimization of switching between dense and sparse representation transparently is beneficial. Without this optimization, the running time becomes 16 minutes. These numbers are using staged execution.

To make the CF algorithm practical, the implementation in Mahout over Hadoop performs aggressive pruning of M so as to re-sparsify it. This may have an impact on the end result. Whether such an efficiency-vs-accuracy tradeoff is reasonable is not the focus of our study, but it allows the algorithm to terminate in Mahout. We find that matrix multiplication in Mahout is reasonable for sparse matrices, however, as soon as dense matrices are involved, the algorithm needs to be changed altogether. The MapReduce APIs has made it difficult to construct a more efficient execution plan, which would be necessary for this algorithm to be implemented efficiently. In contrast, MadLINQ is both flexible in forming the plan and robust with regard to the density of the matrix.

6. Related Work

Existing domain-specific engines for handling big data have a focus on relational algebra (e.g., MapReduce [19], Hive [3], DryadLINQ [36], Pig Latin [30]), with more recent developments in graph analysis (e.g., Pregel [27], GraphLab [28]). MadLINQ complements these efforts with its focus on linear algebra. Since matrices naturally represent graphs, MadLINQ can also tackle a substantial subset of graph algorithms. Inspired by works such as Combinatorial BLAS [12], we have successfully ported graph algorithms such as BFS, MCL, Betweeness Centrality [12], PageRank, random bipartite graph matching and semi-clustering to MadLINQ. Furthermore, recognizing that a holistic system must incorporate multiple domain-specific engines, we demonstrated how to seamlessly integrate MadLINQ with DryadLINQ under the uniform language framework of LINQ.

MadLINQ provides a set of domain-specific constructs and translates a tiled matrix algorithm into the DAG automatically, using in-flight symbolic execution that avoids the DAG explosion problem. Other DAG generation processes like CIEL [29] could in theory be leveraged by MadLINQ, but they are not particularly tailored for the tile algorithms that we focus on.

In terms of system design, our core contribution is the fine-grained pipelined DAG execution engine. Unlike other DAG engines such as Dryad and CIEL, FGP enables fault-tolerant streaming by default. The performance gain of streaming is application dependent, but it is one optimization that simultaneously exploits more parallelism, reduces burstiness of network traffic, and removes the dependency of expensive disk I/O, provided it deals with the complexity of handling failure across vertices. In the pipelined execution, when a failure occurs, all downstream vertices are affected. A conservative approach, adopted for example in Dryad [23] TCP streaming, MapReduce Online [18] and CIEL, is to restart all those nodes and/or throw away useful work. Yet, simply restarting the failed vertices will trigger redundant recomputation in the downstream nodes instead. Alternatively, one can adopt the classical Chandy-Lamport protocol [16] to handle failures in the pipeline. The protocol is general, where the communication can have arbitrary pattern (instead of a DAG) and computation can be non-deterministic (instead of deterministic). However, checkpointing may impose significant runtime overhead for our scenario.

To the best of our knowledge, FGP is the first design that is capable of minimizing recomputation for failure recovery, and is robust against arbitrary failure patterns. Critically, it only requires that data dependencies can be computed or tracked at runtime. In the context of large-scale matrix computation, we have demonstrated this through formal protocol development. Our experiments show that the system can withstand massive resource fluctuation, an important scenario in Cloud computing.

MadLINQ advances the state of the art as a matrix computation platform. Table 1 summarizes existing approaches with regard to the key properties: programmability, execution model, scalability, and failure-handling. We believe that these are important attributes in today's context, where new and experimental algorithms are being developed continuously, and the system needs to deal with large volume of data while relying on unreliable Cloud-level hardware.

HPC Solutions: Matrix computation has been a focus area in the HPC community for many years. LAPACK [7] was developed nearly 20 years ago, and its algorithms are highly tuned for shared-memory and multi-core architectures. ScaLAPACK [17] is its distributed variant and uses an SPMD model, which is problematic in terms of scalabil-

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

Table 1. Comparison with alternative approaches and systems.

ity and fault-tolerance: the problem size is constrained by aggregate memory size, and implicit global barrier is executed at each matrix operation. As such, today's HPC solutions often require high-end network support, and are unsuitable for today's Cloud-level hardware. MadLINQ leverages the well-tuned LAPACK for local computation, but replaces its distributed framework with a pipelined DAG execution model. Even at the stage of a research prototype, MadLINQ has demonstrated that it is competitive with ScaLAPACK in terms of performance, while providing a level of fault-resilience that ScaLAPACK is unable to offer.

Tile Algorithms & DAG Execution: Using tile algorithms to derive DAG style execution for matrix computation is not new. For a more complete treatment, we refer readers to recent work such as FLAME [22], PLASMA [6] and DAGuE [9]. DAGuE adds annotations to a tiled matrix algorithm so that the DAG is explicitly embedded along the edges of data inputs and outputs. Vertices are then statically mapped out to a cluster of machines, and the computation is entirely decentralized. As such, this architecture can exploit maximum parallelism by using much smaller tiles. MadLINQ employs a central scheduler so as to deal with failures and resource dynamics, which are not addressed in these systems. The consequence is that tile size cannot be too small, otherwise scheduling overhead can be significant. This design choice, however, leads to reduced vertex level parallelism. We mitigate this by using pipelining at block granularity to exploit inter-vertex parallelism. Finally, MadLINQ removes the need of user annotation altogether by deriving the DAG automatically.

Matrix Algorithms in MapReduce: The wide-spread adoption of MapReduce in dealing with big data presents a recent paradigm shift. Open source projects such as HAMA [32] and PEGASUS [33] have used MapReduce (or Hadoop) to express matrix algorithms. However, expressing and composing matrix algorithms using the narrow MapReduce APIs is tedious and improvisational. The otherwise straightforward flow of the algorithm needs to be broken down into various map and reduce phases and as a consequence, programming even slightly complex matrix algorithms such as Cholesky requires serious effort (see NMF [26]). The root of the problem is that MapReduce is a subset of relational algebra, and is fundamentally ill-suited to express linear algebra algorithms directly and naturally. Furthermore, solutions such as HAMA are based on the MapReduce abstraction and the execution is bulk synchronous since no reducer can proceed until all mappers complete. In contrast, MadLINQ's engine is fully dataflow-driven (like Dryad), but with the additional ability to perform fault-tolerant pipelining.

7. Discussion and Conclusion

Our original impetus for designing MadLINQ was a demand from our peer researchers in data mining who were unable to find an easy-to-program and scalable matrix computation platform. The system is now actively being used by these and other researchers to develop algorithms such as RLSI. During the course of developing and using the system, we have learned many lessons and identified what we believe to be key avenues for future research:

- **Auto-Tiling**. Currently, a vertex is pipelineable if and only if it represents a tile algorithm. In general, this is not true. For example, our current implementation of SVD is a tile algorithm, but its DAG includes vertices whose operation is not, and thus has to be run in staged mode. Auto-tiling (or blocking) is an important research field in the HPC community, and we will adopt appropriate techniques when available.

- **Dynamic Re-Tiling/Blocking**. As the MCL results indicate, especially for graph algorithms, the nature of the matrices may evolve and require different block and tile size.
- **Sparse Matrices**. Handling sparse matrices well is more difficult than dense matrices, because non-zero distribution can create severe load imbalance.

The current emphasis by the system community on scalable engines such as MapReduce, DryadLINQ and Hive is not accidental. These systems represent and scale-out a subset of the most useful relational algebra APIs. Deeper analysis using linear algebra and graph algorithms, often experimental in nature and operating on large-scale data sets, also need a system that is similarly easy to program, scalable, fault-tolerant and inter-operable. We believe MadLINQ contributes much to fill this vacuum.

Acknowledgements

We thank Cheng Xie, Guowei Liu and Yong He for their help when incubating this project, Jun Xu and Hang Li for the porting of RLSI, Steve Reinhardt for the introduction to Combinatorial BLAS, Jeff Baxter and Zhaoguo Wang on ScaLAPACK and Mahout experiments. We are indebted to our reviewers for their insightful comments on the paper, and particularly to our shepherd Terence Kelly for his feedback and help in writing the final version of this paper.

References

[1] Intel Math Kernel Library. `http://software.intel.com/en-us/articles/intel-mkl/`

[2] Hadoop project.`http://hadoop.apache.org/`

[3] HIVE project. `http://hadoop.apache.org/hive/`

[4] Mahout project. `http://mahout.apache.org/`

[5] TLC—The TLA+ Model Checker. `http://research.microsoft.com/en-us/um/people/lamport/tla/tlc.html`

[6] AGULLO, E., DEMMEL, J., DONGARRA, J., ET AL. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series 180*, 2009.

[7] ANDERSON, E., BAI, Z., DONGARRA, J., ET AL. LAPACK: A portable linear algebra library for high-performance computers. In *Supercomputing*, 2002.

[8] BENNETT, J., LANNING, S. The Netflix Prize. In *KDD*, 2007.

[9] BOSILCA, G., BOUTEILLER, A., DANALIS, A., ET AL. Dague: A generic distributed dag engine for high performance computing. Tech. Rep. ICL-UT-10-01, EInnovative Computing Laboratory, University of Tennessee, 2010.

[10] BRANDES., U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25*, 2001.

[11] BREESE, J.S., HECKERMAN, D., KADIE, C., ET AL. Empirical analysis of predictive algorithms for collaborative filtering. In *Conf. on Uncertainty in Artificial Intelligence*, 1998.

[12] BULUÇ, A. Linear algebraic primitives for parallel computing on large graphs. PhD thesis, University of California at Santa Barbara, 2010.

[13] CHAIKEN, R., JENKINS, B., LARSON, P.Å., ET AL. Scope: easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.

[14] CHAKRABARTI, D., ZHAN, Y., FALOUTSOS, C. R-MAT: A recursive model for graph mining. *SIAM Data Mining 6*, 2004.

[15] CHAMBERS, C., RANIWALA, A., PERRY, F., ET AL. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[16] CHANDY, K., LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3*, 1985.

[17] CHOI, J., DONGARRA, J., POZO, R., WALKER, D. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[18] CONDIE, T., CONWAY, N., ALVARO, P., ET AL. MapReduce online. In *NSDI*, 2010.

[19] DEAN, J., GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[20] FREEMAN, L. A set of measures of centrality based on betweenness. *Sociometry 40*, 1977.

[21] GOLUB, G.H., VAN LOAN, C.F. Matrix computations. Johns Hopkins Univ Pr., 1996.

[22] GUNNELS, J., GUSTAVSON, F., HENRY, G., VAN DE GEIJN, R. Flame: Formal linear algebra methods environment. *ACM Trans. on Math. Software (TOMS) 27*, 2001.

[23] ISARD, M., BUDIU, M., YU, Y., ET AL. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[24] LAMPORT, L. Specifying systems: The TLA+ language and tools for hardware and software engineers, 2002.

[25] LAWSON, C., HANSON, R., KINCAID, D., KROGH, F. Basic linear algebra subprograms for Fortran usage. *ACM Trans. on Math. Software (TOMS) 5*, 1979.

[26] LIU, C., YANG, H., FAN, J., ET AL. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, 2010.

[27] MALEWICZ, G., AUSTERN, M.H., BIK, A.J.C., ET AL. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[28] LOW, Y., GONZALEZ, J., KYROLA, A., ET AL. Graphlab: A new framework for parallel machine learning. In *Conf. on Uncertainty in Artificial Intelligence*, 2010.

[29] MURRAY, D.G., SCHWARZKOPF, M., SMOWTON, C., ET AL. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI*, 2011.

[30] OLSTON, C., REED, B., SRIVASTAVA, U., ET AL. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[31] SCHOENIG, S., DUCASSÉ, M. A backward slicing algorithm for prolog. *Static Analysis*, 1996.

[32] SEO, S., YOON, E., KIM, J., ET AL. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *CloudCom*, 2010.

[33] KANG, U., TSOURAKAKIS, C.E., FALOUTSOS, C. PEGASUS: mining peta-scale graphs. *Knowledge and Information Systems 27*, 2011.

[34] VAN DONGEN, S. Graph clustering via a discrete uncoupling process. *SIAM J. Matrix Anal. Appl 30*, 2008.

[35] WANG, Q., XU, J., LI, H., CRASWELL, N. Regularized latent semantic indexing. In *SIGIR*, 2011.

[36] YU, Y., ISARD, M., FETTERLY, D., ET AL. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.