MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud

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

Presented by Kenneth Lui
Oct 27th, 2015

MadLINQ Project

- Goals
  - Scalable, efficient and fault-tolerant matrix computation system
  - Seamless integration of the system with a general purpose data-parallel computing system

Gap filled by MadLINQ

- Distributed execution engines (Hadoop, Dryad) and their “high-level language interfaces” (Hive, Pig, DryadLINQ) are subsets of relational algebra
- These system are not native for solving problems involving linear algebra and matrix computation

Programming Model

- Matrix algorithms are expressed as sequential programs operating on **tiles**
- Expose to .NET developer via the **LINQ** technology
  - e.g. (Classes like Matrix, Tile)

Code Sample

```
// The input datasets
var ratings = PartitionedTable.Get(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));
```

System Architecture and Components

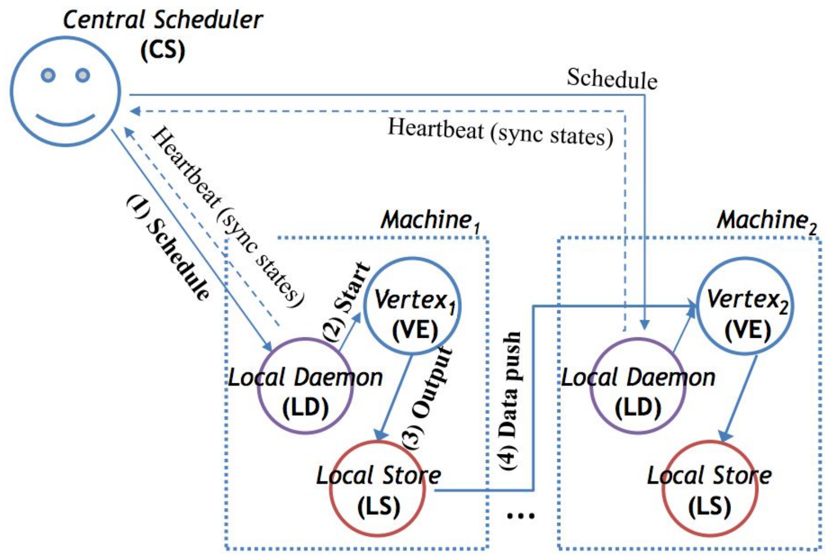


Figure 5. MadLINQ system architecture. The system consists of a Central Scheduler, and a Local Daemon, a Local Store and a Vertex Engine on each compute node.

DAG Generation

- List of running vertices and their children are kept in the memory of scheduler
- Frontier of the execution
- DAG is dynamically expanded through **symbolic execution**
  - Vertices are created based on operations/statements in the program and vertices are connected by data dependencies identified by tiles
  - Removes the need to keep a materialized DAG

Key Contributions

- Extra parallelism using fine-grained pipelining (FGP)
- Efficient on-demand failure recovery

Both enabled by the matrix abstraction

Fine-grained pipelining (FGP)

Fine-grained pipelining (FGP)

- In most DAG, the output of each vertex is “ready” at the same time, i.e. staged. If B depends on A, B waits for A to finish first.
- FGP: exchange data among computing nodes in a pipelined fashion (instead of staged) to aggressively overlap computation of depending vertices (i.e. connected with edges)

Fine-grained pipelining (FGP)

- Parallelism in matrix algorithm fluctuates in different phases/iterations
  - Reduce vertex-level parallelism
  - Cause bursty network utilization
- Introduce Inter-vertex pipelining
  - Vertices consume and produce data in **blocks**, which are essentially smaller tiles
  - Requirement: vertex computation must be expressed as a tile algorithm

Execution Mode

- Staged
  - A vertex is ready when its parents have produced <u>all</u> the data
  - Dryad or MapReduce
- Pipelined
  - A vertex is ready when <u>each input channel</u> has <u>partial</u> results
  - Default for MadLINQ

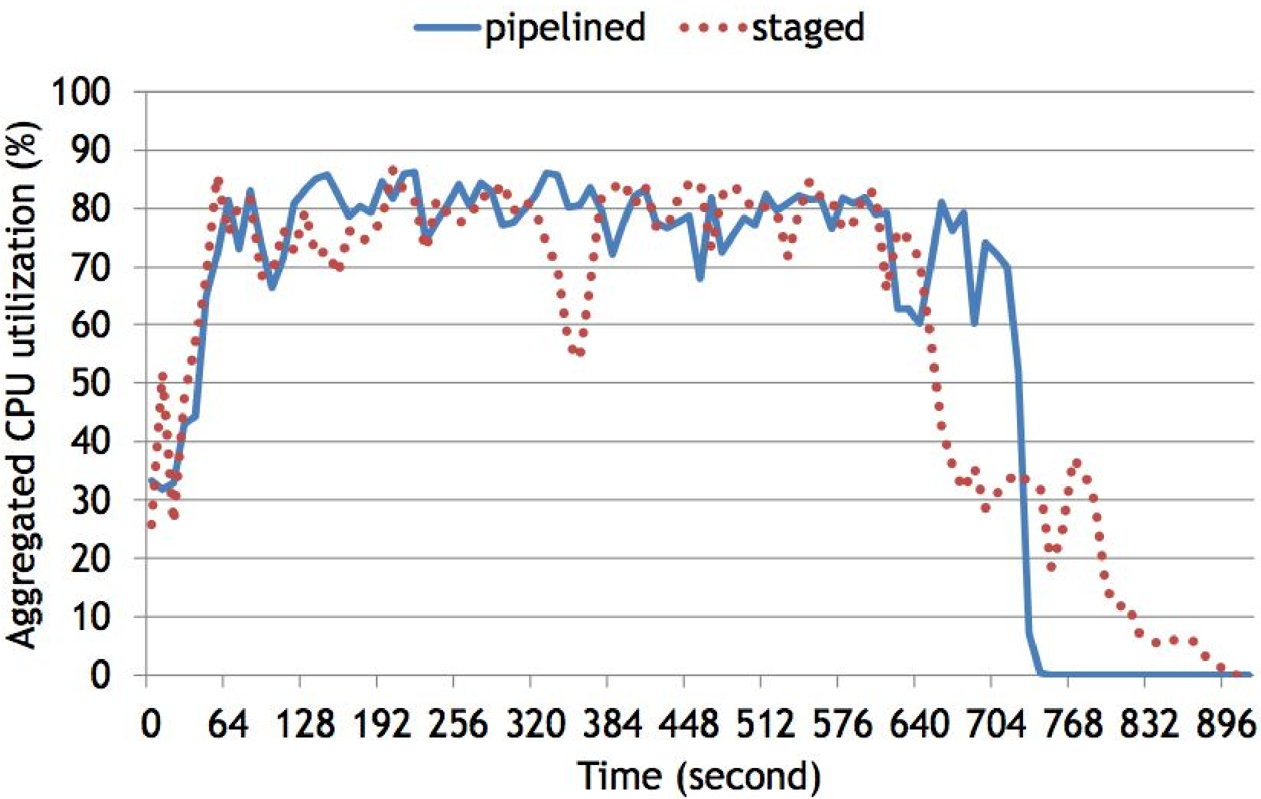
Fault-tolerant protocol

- Using lightweight dependency tracking, FGP allows for minimal recomputation upon failure
- For any given set of output blocks S, we can automatically derive the set of input blocks that are needed to compute S (backward slicing)
- Support arbitrary additions and/or removals of machines (dynamic capacity change)

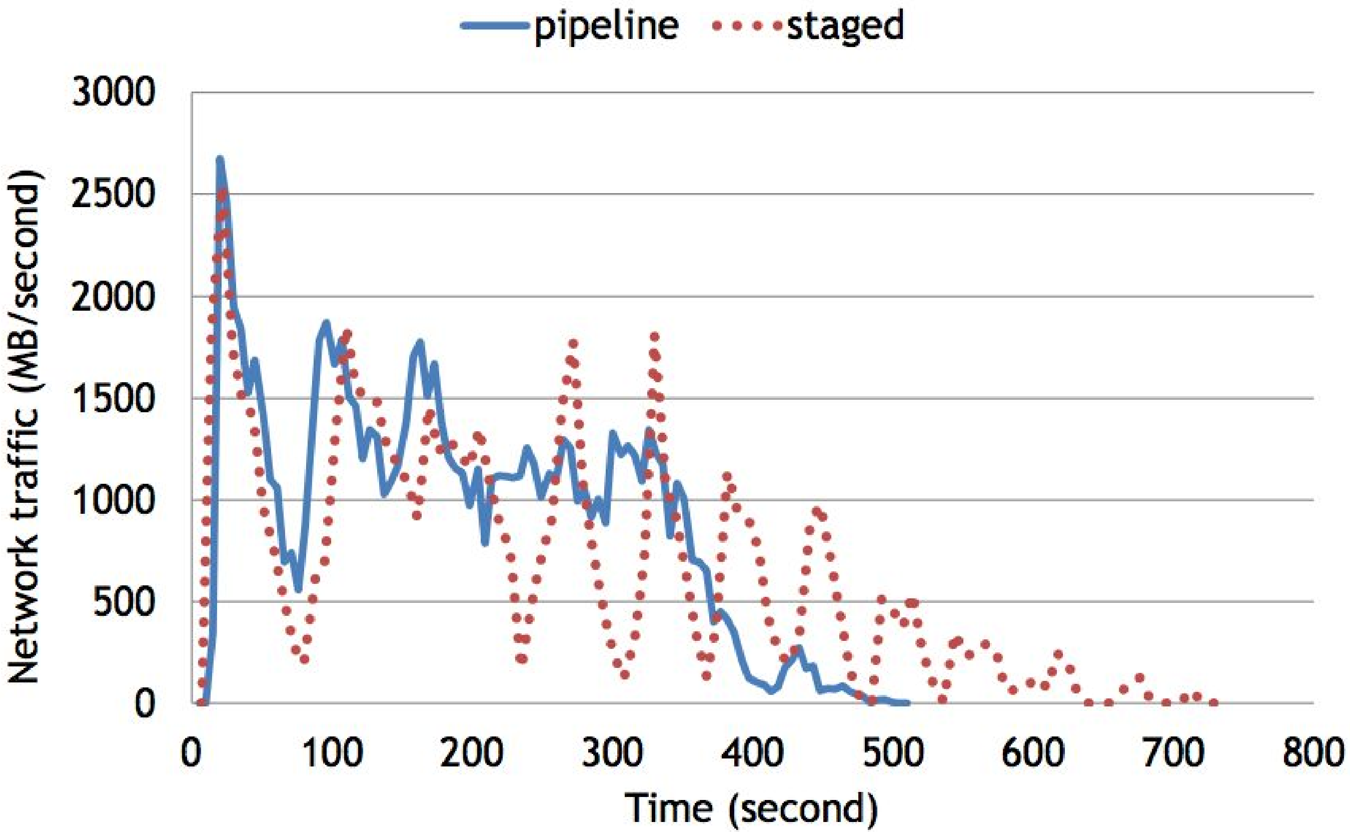
Fault-tolerant protocol - Assumptions

1. Can infer the set of input blocks that a given output block depends on
   a. If not, the protocol falls back to staged model
2. Vertex computation is deterministic

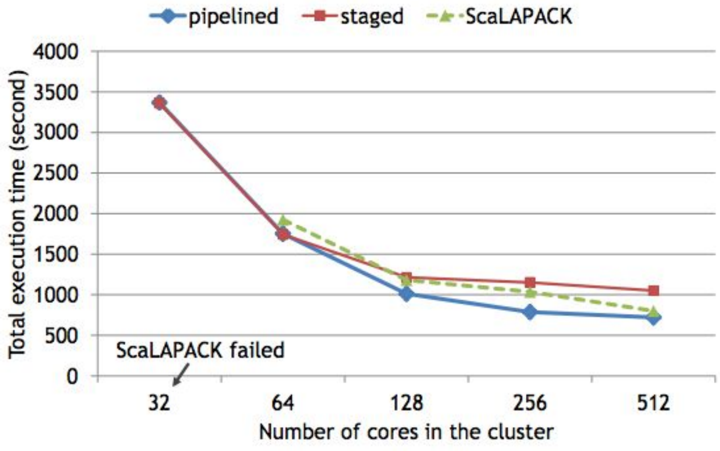
Experiment Result (Cholesky Factorization)



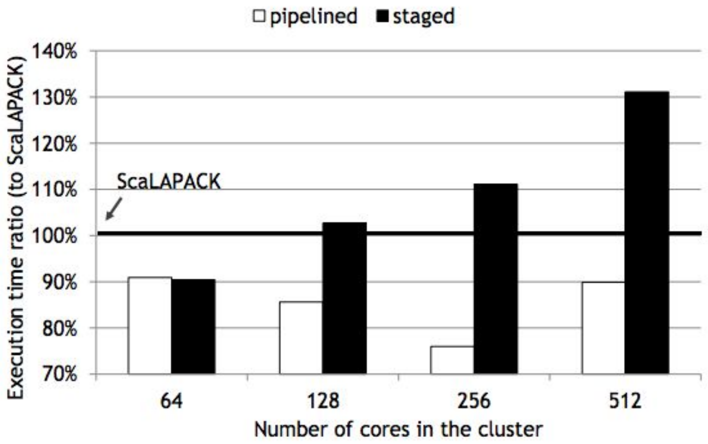
Experiment Result (Cholesky Factorization)



Experiment Result (Comparison to ScaLAPACK)



(a) Absolute running time



(b) Relative to ScaLAPACK

Optimization

- Pre-loading a ready vertex onto a computing node which will finish its current vertex soon
- Adding order-preference (e.g. row-major, column-major) when requesting input for a vertex
- Auto-switching of block representation depending on matrix sparsity
    - and invoke different math library

Configurable parameters

- Tile size
  - smaller tiles = more tile-level parallelism, but increases scheduling/memory overhead
- Block size
  - Underlying math libraries (e.g. Intel MKL) typically yield better performance for bigger blocks
  - But smaller block size => better pipelining

Related Works

	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.

What the paper didn't explain much

- Where are the intermediate data stored?
- Does it assume full-use of the computing cluster (like Dryad)?
- CPU-bound v.s. IO-bound problems?
- How does it compare to DAGuE and HAMA?

Comments

- Seem to make use of property of matrix operation very well in DAG
- Doesn’t seem to bring new “system” invention
- Converting an algorithm into tile algorithm is the key to “gain” from this framework, but this is not easy and remains an active research area in HPC field