Distributed Machine Learning and Graph Processing with Sparse Matrices

Shivaram Venkataraman*, Erik Bodzsar#
Indrajit Roy+, Alvin AuYoung+, Rob Schreiber+

*UC Berkeley, #U Chicago, +HP Labs

Big Data, Complex Algorithms

PageRank
(Dominant eigenvector)

Machine learning + Graph algorithms

Anomaly detection
(Top-K eigenvalues)

User Importance
(Vertex Centrality)
PageRank

Web Graph

Adjacency Matrix

Page Rank

0 0 1 1
0 0 0 0
1 0 0 0
1 0 0 0

0.03
5
0.00
6
0.00
8 ...
0.03
2

PageRank Using Matrices

M = web graph matrix
p = PageRank vector

Power Method
Dominant eigenvector

Iterate
Array-oriented programming environment

Millions of users, thousands of free packages

Popular among statisticians, bioinformatics communities

PageRank Using Matrices

Simplified algorithm: \( \text{repeat } \{ p = M \times p \} \)

Power Method
Dominant eigenvector

\( M = \) web graph matrix
\( p = \) PageRank vector
PageRank Using Matrices

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
\mathbf{p}_1 \\
\mathbf{p}_2 \\
\vdots \\
\mathbf{p}_n
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\mathbf{p}_1 \\
\mathbf{p}_2 \\
\vdots \\
\mathbf{p}_n
\end{array}
\end{array}
= \\
\begin{array}{c}
\begin{array}{c}
\mathbf{p}_1 \\
\mathbf{p}_2 \\
\vdots \\
\mathbf{p}_n
\end{array}
\end{array}
\end{align*}
\]

Web graph matrix \times \text{Pagerank vector} = \text{Pagerank vector}

Power Method
Dominant eigenvector

\[\mathbf{M} = \text{web graph matrix} \quad \mathbf{p} = \text{PageRank vector}\]

Large-Scale Processing Frameworks

- Process each \textit{record} in parallel
- Use case: Computing sufficient statistics

Graph-centric frameworks – Pregel/GraphLab (2010)
- Process each \textit{vertex} in parallel
- Use case: Graphical models

Array-based frameworks – MadLINQ (2012)
- Process \textit{blocks} of array in parallel
- Challenges with sparse matrices
Challenge 1 – Communication

Sparse matrices →
Communication overhead

- R - single-threaded
- Share data through pipes/network
- Time-inefficient (sending copies)
- Space-inefficient (extra copies)

Server 1

Server 2

Challenge 2 – Sparse Matrices

10
Challenge 2 – Sparse Matrices

Presto

Framework for large-scale iterative linear algebra

Extend R for scalability
Outline

- Motivation
- Programming model
- Design
- Applications and Results
for each $h$

$$f(x)$$

PageRank Using Presto

$$M \leftarrow \text{darray}(\text{dim}=c(N,N), \text{blocks}=(s,N))$$

$$P \leftarrow \text{darray}(\text{dim}=c(N,1), \text{blocks}=(s,1))$$

while(...){
  foreach(i,1:len,
    calculate(p=splits(P,i),m=splits(M,i),
    x=splits(P_old),z=splits(Z,i)) {
      p $\leftarrow (m \times x) + z$
    }
  )
  P_old $\leftarrow P$
}
PageRank Using Presto

\[ M \leftarrow \text{darray}(\text{dim}=\text{c}(N,N), \text{blocks}=\text{c}(s,N)) \]
\[ P \leftarrow \text{darray}(\text{dim}=\text{c}(N,1), \text{blocks}=\text{c}(s,1)) \]

while(..){
  foreach(i,1:len,
    calculate(p=splits(P,i),
      m=splits(M,i),
      x=splits(P_old,i),
      z=splits(Z,i))
    {p \leftarrow (m\times x)+z}
  }
}

\[ P_{\text{old}} \leftarrow P \]

Execute function in a cluster
Pass array partitions

Breadth-first Search Using Matrices

\[ G = \text{adjacency matrix} \]
\[ X = \text{BFS vector} \]

Simplified algorithm:
repeat \{ X = G\times X \}
Outline

- Motivation
- Programming model
- Design
- Applications and Results

Presto Architecture
Dynamic Partitioning of Matrices

Profile execution

Partition

Size Invariants

\textit{invariant}(\text{Mat,vec})
Outline

• Motivation
• Programming model
• Design
• Applications and Results

demo
lj_matrix ← darray(dim=c(n,n),blocks=c(n,n))

in_vector ← darray(dim=c(n,1), blocks=(s,1),
                   data=1/n)

out_vector ← darray(dim=c(n,1), blocks=(s,1))

foreach(i, 1:length(splits(lj_matrix)),
       function(g = splits(lj_matrix, i),
               i = splits(in_vector),
               o = splits(out_vector, i)) {
      n ← g %*% o
      update(n)
   })
Examples

dotprod <- function(a,b) {
  tmp <- darray(d=dim(a)/a@blocks,c(1,1))
  foreach(i, 1:length(splits(a)),
    mult <- function(tmp = splits(tmp,i),
      a = splits(a,i),
      b = splits(b,i)) {
      tmp <- sum(a * b)
      update(tmp)
    }, progress=FALSE)
  return(sum(getpartition(dotprod.tmp)))
}

Examples

reduce <- function(f,d) {
  i <- 1
  n <- length(splits(d))
  repeat {
    step <- 2*i
    reducers <- floor((n-i-1)/step)+1
    foreach(j, 1:reducers,
      reduce.pair <- function(s1=splits(d, (j-1)*step+1),
        s2 = splits(d,(j-1)*step+1+i),
        fun = f) {
        s1 <- fun(s1,s2)
        update(s1)
      }, progress=FALSE)
    i <- i*2
    if (1+i > n) {
      break
    }
  }
}
## Applications Implemented in Presto

<table>
<thead>
<tr>
<th>Application</th>
<th>Algorithm</th>
<th>Presto LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>Eigenvector calculation</td>
<td>41</td>
</tr>
<tr>
<td>Triangle counting</td>
<td>Top-K eigenvalues</td>
<td>121</td>
</tr>
</tbody>
</table>

### Fewer than 140 lines of code

<table>
<thead>
<tr>
<th>Application</th>
<th>Algorithm</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centrality measure</td>
<td>Graph algorithm</td>
<td>132</td>
</tr>
<tr>
<td>k-path connectivity</td>
<td>Graph algorithm</td>
<td>30</td>
</tr>
<tr>
<td>k-means</td>
<td>Clustering</td>
<td>71</td>
</tr>
<tr>
<td>Sequence alignment</td>
<td>Smith-Waterman</td>
<td>64</td>
</tr>
</tbody>
</table>

## Repartitioning Progress

![Repartitioning Progress Chart]

- **Split size (GB)**: 0, 10, 20, 30
- **Iteration count**: 1 to 15

![Repartitioning Progress Chart]
Repartitioning benefits

No Repartition

Repartition

Versioning Distributed Arrays

Presto

Co-partitioning matrices

Caching partitions

Locality-based scheduling
Conclusion

Linear Algebra is a powerful abstraction
Easily express machine learning, graph algorithms

Challenges: Sparse matrices, Data sharing
Presto – prototype extends R

Blockus

• Expressive distributed computing systems are in-memory
• Being in-memory is problematic for (very) big data
  – Expensive
  – Fault tolerance problems

• Scale Presto vertically
• Eliminate memory limitation
Vertical scaling

• Use SSDs
  – Low latency
  – Fast small I/O
  – Parallel I/O

• SSDs still significantly slower than memory
• Need to do better than OS swap!

Opportunities from Vertical Scaling

• Enable big data analytics on small systems
  – Laptop!
  – Small cluster

• Energy savings for extreme scale systems

• Reduced cost, increased fault tolerance
**Related work: OS paging/buffer cache**

- General purpose
- (almost) no application knowledge
- LRU caching, (conservative) read-ahead
- Reactive (do I/O on pagefault)

**Related work: SSDAlloc**

- C library, replace malloc with malloc_object
- Objects are stored on SSD
- Memory is used as a cache

- Advantage over OS paging: object-level caching
- Useful for web servers, etc.
Related work: GraphChi

- Vertex-level programming
- Iterative, update vertex neighborhoods in each iteration, for each vertex
- GraphChi: I/O optimized execution engine
- Make sure I/O is sequential (essential for HDDs, works for SSDs too)

Blockus Idea

- Use some form of application knowledge to optimize I/O
- Know future computation → prefetching
  - From programmer hints, static analysis, history, etc.
- Know about parallelism → reorder computation to decrease I/O
- Block usage history: better caching (e.g. always keep popular parts of a graph in memory)
- Deeper application knowledge → reorganize data, change computation etc.
Blockus architecture

- Worker I/O engine: executes all I/O operations
- Scheduler: performs I/O and task scheduling

Scheduler challenges

- Presto scheduler
  - Assumes everything fits in DRAM
  - Schedules each task on worker which has most bytes of its input arrays
  - Transfers non-local input data greedily (no network scheduling)

- Blockus: better scheduling policies
  - Load balancing (in memory)
  - Intelligent Prefetching
  - Intelligent computation prioritization
  - Adaptive Caching
Blockus task scheduling policies

- Reorder computation based on memory contents to minimize I/O
- Different reordering policies

![Graph showing running time per iteration](image)

Project ideas

- Fault tolerance by keeping track of lineage
- Smart communication (e.g. pipelines, broadcasting)
- Static analysis of programs to better understand dependencies
  - Better garbage collection
  - Better asynchronicity/reordering
- Recursive parallelism
- Matrix reordering
  - To improve caching for out-of-core
  - Load balancing for distributed system
- Distributed out-of-core computation
- Heterogeneous storage (different SSDs, HDDs, etc.)