Dryad / DryadLINQ

Slides adapted from those of Yuan Yu and Michael Isard
Dryad

• Similar goals as MapReduce
  – focus on throughput, not latency
  – Automatic management of scheduling, distribution, fault tolerance

• Computations expressed as a graph
  – Vertices are computations
  – Edges are communication channels
  – Each vertex has several input and output edges
WordCount in Dryad
Why using a dataflow graph?

• Many programs can be represented as a distributed dataflow graph
  – The programmer may not have to know this
    • “SQL-like” queries: LINQ

• Dryad will run them for you
**Runtime**

- Vertices (V) run arbitrary app code
- Vertices exchange data through files, TCP pipes etc.
- Vertices communicate with JM to report status

- Job Manager (JM) consults name server (NS) to discover available machines.
- JM maintains job graph and schedules vertices

- Daemon process (D) executes vertices

---

**Job schedule**

**Data plane**

- Files, FIFO, Network

**Control plane**

- JM
- NS
- D
Job = Directed Acyclic Graph

Processing vertices

Channels (file, pipe, shared memory)

Inputs

Outputs
Scheduling at JM

• General scheduling rules:
  – Vertex can run anywhere once all its inputs are ready
    • Prefer executing a vertex near its inputs
  – Fault tolerance
    • If A fails, run it again
    • If A’s inputs are gone, run upstream vertices again (recursively)
    • If A is slow, run another copy elsewhere and use output from whichever finishes first
Advantages of DAG over MapReduce

• Big jobs more efficient with Dryad
  – MapReduce: big job runs >=1 MR stages
    • reducers of each stage write to replicated storage
    • Output of reduce: 2 network copies, 3 disks
  – Dryad: each job is represented with a DAG
    • intermediate vertices write to local file
Advantages of DAG over MapReduce

• Dryad provides explicit join
  – MapReduce: mapper (or reducer) needs to read from shared table(s) as a substitute for join
  – Dryad: explicit join combines inputs of different types
• Dryad “Split” produces outputs of different types
  – Parse a document, output text and references
DAG optimizations: merge tree
DAG optimizations: merge tree
Dryad Optimizations: data-dependent re-partitioning

- Distribute to equal-sized ranges
- Sample to estimate histogram
- Randomly partitioned inputs
Dryad example 1: SkyServer Query

- 3-way join to find gravitational lens effect
- Table U: (objId, color) 11.8GB
- Table N: (objId, neighborId) 41.8GB
- Find neighboring stars with similar colors:
  - Join U+N to find 
    \[ T = N.\text{neighborId} \text{ where } U.\text{objId} = N.\text{objId}, U.\text{color} \]
  - Join U+T to find 
    \[ U.\text{objId} \text{ where } U.\text{objId} = T.\text{neighborId} \]
    \[ \text{and } U.\text{color} \approx T.\text{color} \]
SkyServer query

```sql
select
  u.color, n.neighborobjid
from u join n
where
  u.objid = n.objid

u: objid, color
n: objid, neighborobjid
[partition by objid]
```
[distinct]
[merge outputs]

(u.color, n.neighborobjid)
[re-partition by n.neighborobjid]
[order by n.neighborobjid]

select
  u.objid
from u join <temp>
where
  u.objid = <temp>.neighborobjid
and
  |u.color - <temp>.color| < d
Dryad example 2:
Query histogram computation

- Input: log file (n partitions)
- Extract queries from log partitions
- Re-partition by hash of query (k buckets)
- Compute histogram within each bucket
### Naïve histogram topology

<table>
<thead>
<tr>
<th>P</th>
<th>parse lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>hash distribute</td>
</tr>
<tr>
<td>S</td>
<td>quicksort</td>
</tr>
<tr>
<td>C</td>
<td>count occurrences</td>
</tr>
<tr>
<td>MS</td>
<td>merge sort</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Each Q is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Each R is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>MS</td>
</tr>
</tbody>
</table>
```
Efficient histogram topology

P  parse lines
D  hash distribute
S  quicksort
C  count occurrences
MS merge sort
M  non-deterministic merge

Each Q’ is:

Each R is:

Each T is:
MS ▶ C

MS ▶ C ▶ D

M ▶ P ▶ S ▶ C

P  parse lines  D  hash distribute
S  quicksort  MS  merge sort
C  count occurrences  M  non-deterministic merge
P  parse lines
S  quicksort
C  count occurrences
D  hash distribute
MS merge sort
M  non-deterministic merge
MS $\Rightarrow$ C

MS $\Rightarrow$ C $\Rightarrow$ D

M $\Rightarrow$ P $\Rightarrow$ S $\Rightarrow$ C

P  parse lines
S  quicksort
C  count occurrences

D  hash distribute
MS  merge sort
M  non-deterministic merge
Parse lines

Quicksort

Count occurrences

Hash distribute

Merge sort

Non-deterministic merge
P  parse lines
S  quicksort
C  count occurrences
D  hash distribute
MS  merge sort
M  non-deterministic merge
Final histogram refinement

1,800 computers
43,171 vertices
11,072 processes
11.5 minutes
DryadLINQ
DryadLINQ

- LINQ: Relational queries integrated in C#
- More general than distributed SQL
  - Inherits flexible C# type system and libraries
  - Data-clustering, EM, inference, …
- Uniform data-parallel programming model
  - From SMP to clusters
LINQ

`Collection<T> collection;`

`bool IsLegal(Key);`

`string Hash(Key);`

`var results = from c in collection
    where IsLegal(c.key)
    select new { Hash(c.key), c.value};`
DryadLINQ = LINQ + Dryad

Vertex code

Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
where IsLegal(c.key)
select new { Hash(c.key), c.value };

Query plan (Dryad job)

Data

collection

results
DryadLINQ System Architecture

Client machine

.NET program
ToTable
foreach

Query Expr
.DotNet Objects

DryadLINQ
Distributed
query plan

Output Table

Cluster

Dryad

Query plan
Vertex code
Input Tables

Dryad Execution

Results
Output Tables
DryadLINQ example: PageRank

- PageRank scores web pages using the hyperlink graph

To compute the pagerank of \((i+1)\)-th iteration:

\[
P_{i+1}(u) = \sum_{v \in \text{In}(u)} \frac{P_i(v)}{|\text{Out}(v)|}
\]

A page \(u\)'s score is contributed by all neighboring pages \(v\) that link to it

The contribution of \(v\) is its pagerank normalized by the number of outgoing links
DryadLINQ example: PageRank

- DryadLINQ express each iteration as a SQL query

1. Join pages with ranks
2. Distribute ranks on outgoing edges
3. GroupBy edge destination
4. Aggregate into ranks
5. Repeat
One PageRank Step in DryadLINQ

// one step of pagerank: dispersing and re-accumulating rank
public static IQueryable<Rank> PRStep(IQueryable<Page> pages,
                                      IQueryable<Rank> ranks)
{
    // join pages with ranks, and disperse updates
    var updates = from page in pages
                  join rank in ranks on page.name equals rank.name
                  select page.Disperse(rank);

    // re-accumulate.
    return from list in updates
             from rank in list
             group rank.rank by rank.name into g
             select new Rank(g.Key, g.Sum());
}
The Complete PageRank Program

```csharp
public static IQueryable<Rank> PRStep(IQueryable<Page> pages,
    IQueryable<Rank> ranks) {
    // join pages with ranks, and disperse updates
    var updates = from page in pages
                   join rank in ranks on page.name equals rank.name
                   select page.Disperse(rank);

    // re-accumulate.
    return from list in updates
            from rank in list
            group rank.rank by rank.name into g
            select new Rank(g.Key, g.Sum());
}

var pages = PartitionedTable.Get<Page>("dfs://pages.txt");
var ranks = pages.Select(page => new Rank(page.name, 1.0));

// repeat the iterative computation several times
for (int iter = 0; iter < n; iter++) {
    ranks = PRStep(pages, ranks);
}

ranks.ToPartitionedTable<Rank>("dfs://outputranks.txt");
```

```csharp
public struct Page {
    public UInt64 name;
    public Int64 degree;
    public UInt64[] links;
    public Page(UInt64 n, Int64 d, UInt64[] l) {
        name = n; degree = d; links = l;
    }
}

public struct Rank {
    public UInt64 name;
    public double rank;
    public Rank(UInt64 n, double r) {
        name = n; rank = r;
    }
}

public static IQueryable<Rank> PRStep(IQueryable<Page> pages,
    IQueryable<Rank> ranks) {
    // join pages with ranks, and disperse updates
    var updates = from page in pages
                   join rank in ranks on page.name equals rank.name
                   select page.Disperse(rank);

    // re-accumulate.
    return from list in updates
            from rank in list
            group rank.rank by rank.name into g
            select new Rank(g.Key, g.Sum());
}

var pages = PartitionedTable.Get<Page>("dfs://pages.txt");
var ranks = pages.Select(page => new Rank(page.name, 1.0));

// repeat the iterative computation several times
for (int iter = 0; iter < n; iter++) {
    ranks = PRStep(pages, ranks);
}

ranks.ToPartitionedTable<Rank>("dfs://outputranks.txt");
```
Multi-Iteration PageRank

Pages

Ranks

Iteration 1

Iteration 2

Iteration 3

Memory FIFO
Lessons of Dryad/DryadLINQ

• Acyclic dataflow graph is a powerful computation model
• Language integration increases programmer productivity
• Decoupling of Dryad and DryadLINQ
  – Dryad: execution engine (given DAG, do scheduling and fault tolerance)
  – DryadLINQ: programming model (given query, generate DAG)