DRYAD: DISTRIBUTED DATA-PARALLEL PROGRAMS FROM SEQUENTIAL BUILDING BLOCKS

Authors: Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly

Presenter: Zelin Dai
What is Dryad

- Combines computational “vertices” with communication “channels” to form a dataflow graph.
  - General-purpose
  - High performance
  - Distributed execution engine
Dryad System Organization

- Job Manager (JM)
  - Constructs the job's communication graph
  - Schedules the work across available resources
  - Only responsible for control decisions.
  - Not a bottleneck for data transfer
- **Name Server (NS)**
  - Enumerates all available computers.
  - Exposes the position of each computer within the network topology.
**Dryad System Organization**

- Daemon (D)
  - Running on each computer in the cluster
  - Creates processes in behalf of the job manager.
  - Helps JM monitor the state of computation
A Concrete Example of a Dryad Application

- Database
  - Derived from the Sloan Digital Sky Survey (SDSS).

- Target: find all primary objects that
  - Have neighboring objects within 30 arc seconds.
  - \( \geq 1 \) neighbor has a similar color
A Concrete Example of a Dryad Application

- **SQL Query**

```sql
select distinct p.objID
from photoObjAll p
join neighbors n  --- call this join "X"
on p.objID = n.objID
  and n.objID < n.neighborObjID
  and p.mode = 1
join photoObjAll l  --- call this join "Y"
on l.objid = n.neighborObjID
  and l.mode = 1
  and abs((p.u-p.g)-(l.u-l.g))<0.05
  and abs((p.g-p.r)-(l.g-l.r))<0.05
  and abs((p.r-p.i)-(l.r-l.i))<0.05
  and abs((p.i-p.z)-(l.i-l.z))<0.05
```

- **Table “photoObjAll”**
  - 354,254,163 records, keyed by `objID`.
  - Object’s color in five bands: `u`, `g`, `r`, `i` and `z`.
  - **“mode”** select only “primary” objects.
A Concrete Example of a Dryad Application

- SQL Query

```sql
select distinct p.objID
from photoObjAll p
join neighbors n  — call this join “X”
on p.objID = n.objID
    and n.objID < n.neighborObjID
    and p.mode = 1
join photoObjAll l  — call this join “Y”
on l.objid = n.neighborObjID
    and l.mode = 1
    and abs((p.u-p.g)-(l.u-l.g))<0.05
    and abs((p.g-p.r)-(l.g-l.r))<0.05
    and abs((p.r-p.i)-(l.r-l.i))<0.05
    and abs((p.i-p.z)-(l.i-l.z))<0.05
```

- Table “neighbors”
  - 2,803,165,372 records
  - Eliminates duplication caused by symmetric neighbors relationship.
SQL Query Execution:

- Uses only a few columns
- Uses an index on photoObjAll:
  - Keyed by \textbf{objID}
  - Additional columns for mode, u, g, r, i and z.
- Uses an index on neighbors
  - Keyed by \textbf{objID}
  - An additional neighborObjID column.
Dryad Experimental Environment Setup

- Omitted unused columns
  - Avoid transporting the entire database.
- Extracted indexes into two binary files (U, N)
  - Each sorted same as the indexes.
  - “U”: 36-byte records, 11.8GBytes
  - “N”: 16-byte records, 41.8GBytes
A Concrete Example of a Dryad Application

- **Input Partition:**
  - $n$ equal parts
  - Partitioned by `objID` ranges

- The Dryad communication graph for the SQL query
A Concrete Example of a Dryad Application

- X vertices (Join “X”)
  - Merge partitioned $U_i$ and $N_i$
  - Keyed on objID
  - Filtered by the $<\text{ expression}$
  - $p.mode = 1$
  - Produces records containing:
    - objID and neighborObjID
    - Color columns corresponding to objID.

- The Dryad communication graph for the SQL query
A CONCRETE EXAMPLE OF A DRYAD APPLICATION

- **D vertices**
  - Distribute output records to M
  - Partition by `neighborObjID`

- **M vertices**
  - A non-deterministic merge of inputs.

- **S vertices**
  - In-memory Quicksort on `neighborObjID`

- The Dryad communication graph for the SQL query
A Concrete Example of a Dryad Application

Y vertices (Join “Y”)
- Take records from \(S_{4i-3} \ldots S_{4i}\) \((i = 1 \ldots n)\)
- Merge with another read of \(U_i\)
- Keyed on \texttt{objID} (from \(U\)) = \texttt{neighborObjID} (from \(S\))
- Filtered by matching the colors

The Dryad communication graph for the SQL query
A Concrete Example of a Dryad Application

- **H vertices ("distinct")**
  - Merge outputs of Y into a hash table
- A enumeration of hash table delivers the result

- The Dryad communication graph for the SQL query
Describing a Dryad Graph

- A simple language to specify communication idioms.

- The basic object is a graph:

  \[ G = \{ V_G, E_G, I_G, O_G \} \]

  - \( V_G \): a sequence of vertices
  - \( E_G \): a set of directed edges
  - \( I_G \subseteq V_G \) tags input vertices
  - \( O_G \subseteq V_G \) tags output vertices
Creating New Vertices

- A singleton graph:
  \[ G = \langle (v), \emptyset, \{v\}, \{v\} \rangle \]

- A new graph containing k copies
  \[ C = G^k \]
  \[
  C = \left\langle \bigoplus_{i=1}^{k} V_G^i, E_G^i \cup \cdots \cup E_G^k, \bigcup_{i=1}^{k} I_G^i, \bigcup_{i=1}^{k} O_G^i \right\rangle
  \]
  \[ G^n = \langle V_G^n, E_G^n, I_G^n, O_G^n \rangle \]
Creating a New Edge

- A composition operation to two graphs.
- \( C = A \circ B:\)
  \[
  C = \langle V_A \oplus V_B, E_A \cup E_B \cup E_{\text{new}}, I_A, O_B \rangle
  \]
- \( E_{\text{new}}: \) introduced between vertices in \( O_A \) and \( I_B \).
- \( V_A \) and \( V_B \) are enforced to be disjoint
- \( A \) and \( B \) are both acyclic
  \[\Rightarrow C \text{ is also.}\]
**Two Standard Compositions**

- A \(\geq\) B
  - A pointwise composition.

- A \(\gg\) B
  - The complete bipartite graph between \(O_A\) and \(I_B\).
**Merging Two Graphs**

- \( C = A \parallel B \)

\[
C = \langle V_A \oplus *V_B, E_A \cup E_B, I_A \cup *I_B, O_A \cup *O_B \rangle
\]

- \( V_A \oplus *V_B \)
  Concatenation of \( V_A \) and \( V_B \) with duplicates removed
Bypass and Fork/join

- Construct bypass and fork/join using merge.
  - Bypass (g): merge (f) with $AS \geq BS$
  - (h): an asymmetric fork/join
DESCRIBING A DRYAD GRAPH

The graph builder program to construct the query graph:

```plaintext
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^((N*4));
GraphBuilder SSet = moduleS^((N*4));
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;

GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;

GraphBuilder XToY = XSet > DSet >= MSet >= SSet;
for (i = 0; i < N^4; ++i)
{
   XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}

GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;

GraphBuilder final = XInputs || YInputs || XToY || YToH || HOutputs;
```
**CHANNEL TYPES**

- Temporary file (default)
  - Producer writes to disk
  - Consumer reads from that file.
- User specified transport protocols

<table>
<thead>
<tr>
<th>Channel protocol</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>File (the default)</td>
<td>Preserved after vertex execution until the job completes.</td>
</tr>
<tr>
<td>TCP pipe</td>
<td>Requires no disk accesses, but both end-point vertices must be scheduled to run at the same time.</td>
</tr>
<tr>
<td>Shared-memory FIFO</td>
<td>Extremely low communication cost, but end-point vertices must run within the same process.</td>
</tr>
</tbody>
</table>
**INPUT AND OUTPUT**

- Partition a large input file.
- Distribute partitions across the cluster.
- Group a logical input into a graph:
  \[ G = \langle V_p, \emptyset, \emptyset, O_p \rangle \]
  - \( V_p \): "virtual" vertices for input partitions.
- Logically concatenate output partitions
  - On job completion
  - Form a single named distributed file.
**JOB STAGES:**

- Place every vertex in a “stage”
  - Simplify job management.
  - Most stages are connected using “>=”
  - D is connected to M using “>>>”.
WRITE A VERTEX PROGRAM

- Vertex execution
- Efficient pipeline execution
**Vertex Execution**

- A runtime library
  - Setting up
  - Receives:
    - A closure from JM describing the vertex
    - URIs describing the input and output channels
  - Executing vertices
EFFICIENT PIPELINE EXECUTION

- Most vertices contain purely sequential code
- Also supports an event-based programming style
  - Uses shared thread pool (STP)
  - Runtime automatically distinguishes between vertices
  - Encapsulated large graphs are executed on STP
Job Execution

- JM has a scheduler
- Monitors state and history of each vertex.
  - Every vertex eventually completes
    => Job Success!
  - Any vertex is re-run > t times
    => Entire job failure!
Job Execution

- A vertex may be executed multiple times

- Each execution of a vertex has:
  - A version number
  - An execution record
    - The state of that execution
    - The versions of the predecessor vertices
Fault Tolerance Policy

- Assumes all vertex programs are deterministic:

  - Acyclic communication graph
  - Terminating execution of a job
  - Immutable inputs

  Same result
Fault Tolerance Policy

- JM is informed for any vertex failure

Diagram:
- Vertex reports error
- Process crashes
- Daemon fails
- Heartbeat Timeout
- Daemon
- Job Manager
**Fault Tolerance Policy**

- **Read Error Handle**
  - Read error on an input channel
    - Job Manager
      - Mark channel generator $V_c$ as failed
  - Recreate the offending channel
    - Re-execute $V_c$
      - Terminate its process if it is running
Stage manager callback mechanism

- Stage Manager
  - Receives a **callback** on:
    - Every state transition of a vertex execution
    - A regular timer interrupt
  - Holds a **global lock** on JM data structures
    - Implement sophisticated behaviors

Peer 1: Peers
Peer 2: Slow vertex
Peer 3: Peers

Heuristics

Stage Manager

Duplicate executions
Run-time graph refinement

- An aggregation tree
  - Insert a new layer of internal vertices
  - Each internal vertex reads data from closer inputs.
Partial Aggregation

- Group the inputs into $k$ sets
- Replicate the downstream vertex $k$ times
  - Process all sets in parallel
EXPERIMENTAL EVALUATION

Two variants of the Dryad graph:
  - In-memory
  - Two-pass

M_i -> S_i -> Y is by a shared-memory FIFO.
  - Pulls four sorters into the same process
  - Execute in parallel on the four CPUs.
EXPERIMENTAL EVALUATION

- “In-memory” variant:
  - $D_i \rightarrow four \ M_j$ is also by a shared-memory FIFO
  - Rest of the $D_i \rightarrow M_k$ edges use TCP pipes
- All other communication in both variants:
  - NTFS temporary files
## Experimental Evaluation

- **Two-pass:**
  - $n = 40$
  - # of computers from 1 to 9
- **In-memory:**
  - $n = 6-9$
  - Each time on $n$ computers
- **SQL Server:**
  - One computer

<table>
<thead>
<tr>
<th>Computers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server</td>
<td>3780</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two-pass</td>
<td>2370</td>
<td>1260</td>
<td>836</td>
<td>662</td>
<td>523</td>
<td>463</td>
<td>423</td>
<td>346</td>
<td>321</td>
</tr>
<tr>
<td>In-memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>217</td>
<td>203</td>
<td>183</td>
<td>168</td>
</tr>
</tbody>
</table>
Experimental Evaluation

- “Two-pass”:
  - Works on all cluster sizes
  - Close to linear speed-up

- “In-memory”:
  - Works for n = 6 and up
  - Close to linear speed-up
  - Twice as fast as the “two-pass”
Experimental Evaluation -- Data Mining

- Read query logs gathered by the MSN Search service
- Extract the query strings
- Build a histogram of query frequency
- Basic communication graph:
  - Does not scale well for very large datasets
EXPERIMENTAL EVALUATION -- DATA MINING

- Optimal communication graphs
Experimental Evaluation -- Data Mining

- 10,160,519,065,748 Bytes of input data
- A cluster of around 1800 computers
- The input was divided into 99,713 partitions
- The application is specified to use 450 R subgraphs.
- 33,375,616,713 Bytes output from the R subgraphs

- Time: 11 minutes and 30 seconds.
BUILDING ON DRYAD

- The “Nebula” scripting language
- Integration with SSIS
- Distributed SQL queries
Conclusions

- A general-purpose data-parallel execution engine.
- Automatically exploit network locality.
- Excellent scaling behavior on small clusters.
- Successfully execute jobs containing hundreds of thousands of vertices.
- Process many terabytes of input data in minutes.
- Easily create large-scale distributed applications.
Thank you!

Questions?