DRYAD: DISTRIBUTED DATA-PARALLEL PROGRAMS FROM SEQUENTIAL BUILDING BLOCKS

MICHAEL ISARD, MIHAI BUDIU, YUAN YU, ANDREW BIRRELL, DENNIS FETTERLY
DRYAD GOALS

• Research question: How to make it easier for programmers to express parallel and distributed program?

• General purpose execution environment for distributed, data-parallel applications
  • Focuses on throughput, not latency
  • Assumes secure environment, such as a private data center

• Automatic scheduling, distribution of data and resources, fault tolerance
Dryad is a middleware abstraction that runs programs that are represented as distributed execution graphs.

Dryad receives arbitrary graphs (DAGs) from users/programmers.

Dryad provides mechanisms for allocating resources, scheduling computations, fault-tolerance.
The cluster has a name server (NS) that can be used to enumerate all the available computers. The name server exposes the position of each computer within the network access to the cluster. The job manager contains the state of the computation and how much data has been transferred.

As far as the program in each vertex is concerned, channels can communicate with the remote vertices and monitor a channel. The daemon (D) acts as a proxy to a computer. It maintains the job graph and schedules running vertices (V) is executed on a computer its binary is sent from the job manager to the daemon and subsequently it is executed from (V).

At run time each channel is used to transport a finite sequence of message tuples of base types. The message tuples that are produced by vertices are consumed by compute processes on behalf of the job manager. The first time a vertex of locality is executed, the object's position in the computation graph along with library code to schedule the work can be communicated with the daemon (D) as a proxy. The daemon (D) can communicate with the remote vertices and monitor a channel. The daemon (D) acts as a proxy to a computer.

The job manager (JM) in Figure 1. A Dryad job is coordinated by a process called the "job manager" (denoted JM in the figure) that runs in Figure 1. A Dryad job is coordinated by a process called the "job manager" (denoted JM in the figure) that runs on l.objid = n.neighborObjID

When executed by SQLServer the query uses an index on p.objID = n.objID

There are two tables involved. The first, photoObjAll, includes the object's color, as a magnitude (logarithmic intensity). The query can be expressed in SQL as:

```
SELECT
  p.objID, p.r as r, p.g as g, p.i as i, p.z as z
FROM
  photoObjAll p
JOIN
  neighbors n
ON
  p.objID = n.objID AND n.objID < n.neighborObjID
WHERE
  abs((p.r-p.i)-(l.r-l.i))<0.05
  AND abs((p.u-p.g)-(l.u-l.g))<0.05
```
A DRYAD JOB: DAG

Inputs

Outputs

Processing

Vertices

Channels

Inputs
WHY A DAG?

• Natural “most general” design point, cycles are problematic

• DAG supports full relational algebra
  • Multiple inputs and outputs of different types from the same vertex
  • More general than MR, or defined special cases, no semantics included in the scheduler (just vertices to schedule)
WHY A GENERAL DAG?

• Uniform operations aren’t really uniform
  • e.g., SQL queries after dynamic optimization could look irregular.
WHY A GENERAL DAG?

- Uniform operations aren’t really uniform
  - e.g., SQL queries after dynamic optimization could look irregular.
- Non-trees are common
WHY NO CYCLE?

- No cycles makes scheduling easy:
  - vertex can execute once all its inputs are ready
  - no deadlock
- Fault tolerance is easy
DYNAMIC REFINEMENT OF GRAPH

- Application passes initial graph at start
  - Gets callbacks on interesting events
- Graph can be modified with some restrictions
  - The job scheduler doesn’t itself do these modifications
  - The callbacks go to the application allowing such modifications
  - So this dynamic refinement is really an extension to Dryad
STAGE MANAGER

• Every vertex has a stage manager:
  • manages a bunch of vertices, generally grouped by function
• A place where execution statistics are gathered and callbacks are received
  • request re-executions of failed tasks
• A natural place for building models for tasks executions
CONNECTION MANAGER

- We can overlay a graph on the stage managers
- Any pair of stages can be linked
- Gets callbacks on events in upstream stage:
  - E.g., when vertices finish, new vertices get added
- Most dynamic modifications happen here
VERTEX PROGRAM

Abstract Channels

Vertex Program

Abstract Channels
VERTEX PROGRAM
SOME CASE STUDIES

• SkyServer DB Query
• Query Histogram Computation
SKYSERVER DB 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 and N to find T
    • T = U.color, N.neighborId where U.objId = N.objId
  • Join U and T to find, U.objId
    • U.objId where U.objId = T.neighborId and U.color = T.color
SKYSERVER DB QUERY

- Manually coded the SQL plan in Dryad

Figure 2: The communication graph for an SQL query. Details are in Section 3.1.
Also, each unwieldy to read in parallel from so many channels.

We reduce, so the amount of data which needs to be sorted.

SQLServer result matches our expectations: our special-

...linear in the number of computers used.

...SQL query computation is near-

...hundreds of thousands of partitions, it becomes

...from the parser

...that wrote a total of 118,364,131,628 Bytes. The to-

...most 600 MBytes on the same local switch resulting in 217

...We would like to emphasize several points about the op-

...Details are in Section 6.3. This figure shows the first cut
Optimizations done do not need any code changes, only graph manipulations!
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
HISTOGRAM COMPUTATION: NAÏVE TOPOLOGY

P: Parse lines
D: Hash Distribute
S: Quicksort
C: Count occurrences
MS: Merge Sort

Each Q is:
C
C
Each R is:
S
S

Each

R

MS

Details are in Section 6.3. This figure shows the first cut for the data-mining experiment. It reads query logs, automatically builds a histogram of query frequency, and then distributes it as a stream of key-value pairs. We would like to emphasize several points about the optimization process we used to arrive at the graphs in Figure 10. The non-deterministic merge vertex shown in Figure 7 is used during the optimization process. At no point during the optimization did we have to modify any of the code running inside the vertices. The parser, automated using the refinement in Figure 7 to ensure the communication graph in the first phase now has multiple inputs, grouped subgraphs. Each subgraph, grouped the inputs into at most 1 GBytes at a time, all logically, and builds a histogram of query frequency. The basic computation in this experiment reads query logs gathered by the MSN Search service, extracts the query strings, and builds a histogram of query frequency. The basic computation in this experiment reads query logs gathered by the MSN Search service, extracts the query strings, and builds a histogram of query frequency. The basic computation in this experiment reads query logs gathered by the MSN Search service, extracts the query strings, and builds a histogram of query frequency. The purpose of running this experiment was to verify that Dryad works sufficiently well in these straightforward cases, and that it works at large scales. The data-mining experiment fits the pattern of map then reduce. Each query string, a count, and a hash of the string. Each query string gets to Dryad running on a single computer and times are given in Table 2. The MS: Merge Sort performs an in-memory sort. The output of Comp 2005 SQLServer result matches our expectations: our special-approximative algorithm runs approximately twice as fast as the two-pass variant. The baseline is relative to Dryad running on a single computer and times are given in Table 2. The MS: Merge Sort performs a streaming merge-sort.

1. At no point during the optimization did we have to modify any of the code running inside the vertices: i.e.

2. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

3. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

4. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

5. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

6. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

7. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

8. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

9. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.

10. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e.
HISTOGRAM COMPUTATION: EFFICIENT TOPOLOGY

P: Parse lines
D: Hash Distribute
S: Quicksort
C: Count occurrences
MS: Merge Sort
M: Non-deterministic Merge

**Figure 10:** Rearranging the vertices gives better scaling performance compared with Figure 9.
Figure 6 might be more suitable.

The user supplies graph (a) specifying that different topology might give better performance compared with Figure 9.

Nebula hides most of the details of the Dryad program interface ont o of Dryad. The Nebula layer on top of Dryad, together with some perl scripts, allows a user to specify a computation as a series of stages.

For example, if the reduce phase performed substantial data reduction, then the refine dynamic merge tree as described in Section 5.2 was used. These made it simple to experiment with different optimization schemes that would have been difficult or impossible to implement to experiment with different optimization schemes that would have been di fficult or impossible to implement.

R" by Dynamic refinement, were used. These made it simple to experiment with different optimization schemes that would have been difficult or impossible to implement.

As explained in the introduction, we have targeted Dryad for large-scale data mining, with a low barrier to entry for users. The Nebula system, including subgraph encapsulation and distributed application without compiling any code. The Nebula layer on top of Dryad, together with some perl scripts, allows a user to specify a computation as a series of stages.

Scenarios typically run on thousands of computers and contain per functions, has proved to be very successful for large-scale text processing, with a low barrier to entry for users.
OPTIMIZING DRYAD APPLICATIONS

• Application code is NOT modified
  • Only graph manipulations
  • Users need to provide the vertex programs and the initial graph
  • Then the system optimizes it further, statically and dynamically
SMALL VS LARGE CLUSTERS

- **Small private clusters**
  - Few failures, known resources
  - Can use these sophisticated Dryad features

- **Large public clusters**
  - Unknown resources, failures
  - May not be able to use lot of the graph manipulation features as much
HIGHER-LEVEL PROGRAMMING MODELS

- **SSIS:**
  - SQLServer workflow engine, distributed
- **Simplified SQL:**
  - Perl with a few SQL like operations
- **DryadLINQ**
  - Relational queries integrated in C#
  - a front end for Dryad jobs