Part II: Data Center Software Architecture
Topic 2: Programming Models
Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks

Iyad Kuwatly – Winter 2014
Department of Electrical and Computer Engineering - Wayne State University
Outline

• Credits
• Motivation
• Assumptions
• System Runtime Functionality
• Key Advantages
• Contributions
• The Concept (Dryad Graph)
• Dryad Channel
• Dryad System Organization
• Dynamic Modifications
• Stage Manager
• Sub Graphs
• Data Mining Experiment
Credits

- **Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks**
  Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly

- **Video** of a presentation on Dryad at the Google Campus, given by Michael Isard, Nov 1, 2007.

- **Presentation slides** from a talk on Dryad at University of California at Santa Cruz, by Michael Isard, February 2008.


- All Available at Microsoft Research Page:
Motivation

- To offer flexible general purpose execution environment
- To increase throughput as a design tradeoff at the expense of latency. This system is not designed for real-time applications.
- To make it easier for developers to write efficient parallel and distributed applications making use of the following:
  1. Large scale internet services running on clusters of thousands of general purpose servers
  2. Prediction that future advances in local computing power will come from increasing the number of cores on a chip rather than improving the speed or instruction-level parallelism
Assumptions

1. Resources are in a single administrative domain (no high latency or unreliable communications)
2. Developer explicitly expose the data dependences of a computation using a communication graph
3. Data center is private, not issues with security
System Runtime Functionality

1- Abstract standard concurrency mechanisms
2- Automatically allocate resources
3- Automatically Schedule Tasks
4- Handles components failure
5- Does not try too hard to extract parallelism within the developer provided routine
Key Advantages (Flexibility)

Dryad system allows the developer fine control over the communication graph as well as the subroutines that live at its vertices.

Dryad is notable for allowing graph vertices to use an arbitrary number of inputs and outputs.

Dryad provides good performance for most simple applications.
Contributions

1- General purpose, high performance distributed execution engine
2- Excellent performance all the way from single multi-core system up to clusters of thousands of computers
3- Graph description language and simpler high level abstractions for specific applications
The Concept - Dryad Graph

- The overall structure of a dryad job is determined by its communication flow
- A job is a directed acyclic graph
  - Each vertex is a program
  - Edges represent data channels

- Runtime maps logical computational graph onto physical resources
- Note: There may be more vertices in the graph than execution cores in the cluster
Job = Directed Acyclic Graph

Processing vertices

Channels (file, pipe, shared memory)

Inputs

Outputs
Virtual graph vertices

*Input file expands to set of vertices*
   
   *Each partition is one virtual vertex*

*Output vertices write to individual partitions*

   *Partitions concatenated when outputs completes creating a single file again*
Dryad Channel

Channels produce/consume using heap objects implemented using the following:

1- Shared Memory FIFO (using object pointers less overhead, but end point vertices must be scheduled to run at the same time)

2- TCP Pipes (Serializes data to buffers on TCP Stream, requires no disk access but both vertices must be scheduled at the same time)

3- Temporary files (Serializes data to buffers on a disk, preserved after vertex execution until the job completes)

It is assumed that channels are of finite length and vertices finish.
2-D Piping

- Unix Pipes: 1-D
  grep | sed | sort | awk | perl

- Dryad: 2-D
  grep^{1000} | sed^{500} | sort^{1000} | awk^{500} | perl^{50}
Dryad System Organization

JM: Job Manager  NS: Name Server
V: Running Vertices  D: Daemon as proxy
Dryad Job Manager and Name Server

**Job Manager**
Centralized coordinating process
Runs either with the cluster or from a remote machine
Contains application specific code to construct the job’s communication graph
Library code to schedule the work across the available resources
All data is sent directly between vertices

**Name Server**
Enumerate all available computers
Exposes the position of each computer to account for locality
Dryad Daemon

Running on each computer
Responsible for creating processes on behalf of the job manager
The first time a vertex is executed on a computer, its binary is sent from the job manager to the daemon
Subsequently, it is executed from cache
Can use local disk
Provides information about channels read/write operators to the Job Manager for monitoring purposes
Task Scheduler and File system

**Task Scheduler**
Simple to queue batch jobs

**File System**
Similar to GFS having large files broken to chunks that are replicated and distributed
Also supports NTFS for small clusters
Scheduler Inside Job Manager

- Keeps track of the state and history of each vertex in the graph
- If job manager’s computer fails the job is terminated (central point of failure) although the vertex scheduler could employ checkpointing or replication to avoid this.
- A vertex may be executed multiple times over the length of the job due to failures and more than one job instance of a given vertex may be executing at any given time.
- Each execution of the vertex has a version number and a corresponding “execution record” that contains the state of that execution and the versions of the predecessor vertices from which its inputs are derived.
Scheduler Inside Job Manager

- Each execution names its file-based output channels uniquely using version number to avoid conflicts among versions.
- If the entire job completes successful then each vertex selects a successful execution and renames its output files to their correct final forms.
- When all of a vertex input channels become ready a new execution record is created for the vertex and placed in a scheduling queue.
  - Disk based (when entire file is ready)
  - TCP/shared-memory FIFO (when predecessor vertex has at least one running execution record)
- A vertex can be co-located with a large input file
- If any vertex re-run more than a set number of times then the entire job is failed.
Scheduler Inside Job Manager

- After job is complete:
  - Files representing temporary channels are cleaned
  - Vertices are killed by the daemon
Dryad Graph Characteristics

Directed Acyclic Graph
No cycles
Supports full relational algebra
Multiple inputs coming into the vertex
Multiple outputs coming out of the vertex
Simpler scheduler which deals only with graph
Why a general Directed Acyclic Graph?

“Uniform” stages aren’t really uniform
Why a general Directed Acyclic Graph?

“Uniform” stages aren’t really uniform
Graph complexity composes

- Non-trees common
- E.g. data-dependent re-partitioning for data that we don’t know the distribution of.
  - Combine this with merge trees etc.

Distribute to equal-sized ranges
Sample to estimate histogram
Randomly partitioned inputs
Why no cycles?

- Scheduling is easy
- Vertex can run anywhere once all its input are ready
- Directed-acyclic means there is no deadlock
- Finite-length channels means vertices finish
- When vertex is done it output are marked as ready
Why no cycles?

- Fault tolerance is easy (with deterministic code)
- If A fails, run it again
- If A’s input are gone, run upstream vertices again
- If A is slow, run another copy elsewhere and use output from whichever finishes first
Dynamic modification

• Application passes initial graph at start
  – Gets callbacks on interesting events

• Can modify graph with some restrictions
  • Once a vertex has executed
    • Can no longer be deleted
    • Number and type of channels cannot be changed
  • Could relax some restrictions if necessary
Dynamic Refinement for aggregation

The logical graph on the left connects every input to the single output. The location and sizes of the inputs are not known until run time when it is determined which computer each vertex is scheduled on. At this point the inputs are grouped into subsets that are close in network topology and an internal vertex is inserted for each subset to do a local aggregation, thus saving network bandwidth. The internal vertices are all of the same user-supplied type, in this case shown as “Z”. In the diagram on the right, vertices with the same label ('+' or '*') are executed close to each other in network topology.
A partial aggregation refinement

The successor vertex is replicated k times to process all the sets in parallel.
Stage skeleton Overlay

Stage skeleton overlay

Stage D
Stage C
Stage B
Stage A
Stage Manager

- Every vertex has a stage manager (summary of the overall job)
  - *By convention they group by function*
- Holds a global lock on the job manager data structures
- Used for reporting aggregate statistics
- Gets callbacks on interesting events
  - *Gather execution statistics (how long execution should be?)*
  - *Request duplicate execution*
Stage Manager Graph

- Any pair of stages can be linked
- Get callback on interesting events in upstream stage
  - *This is where most dynamic optimizations are implemented*
Sub graph vertex program
Data Mining Experiment

Experiment Details:
Read query log gathered by MSN search service, extract the query strings and builds a histogram of query frequency.

Steps:
P: Read their part of the log file (query string, count, hash of the string)  
D: Distribute to k outputs based on hash  
S: Performs in memory sort (on hash)  
C: Accumulates total counts for each query  
MS: Performs a streaming merge sort (on hash)

Does not scale well  
Because each partition is 100 MB so after reduction in P the amount of data that needs to be sorted by S is very much small compared to the total RAM on a computer. When R inputs increases so much it becomes unwieldy to read in parallel from so many channels.
Data Mining Experiment

Better Scaling Performance
A: User supplied graph  B: Executed by the system

M: Non-deterministic merge vertex
Take out D distribution vertex to allow another layer of grouping and aggregation
Optimization Process

1. No need to code running inside vertices only graph manipulation
2. Suitable for map reduce where map phase is $P_{vertex}$ and reduce phase is $C_{vertex}$
3. To scale up further we need to add another layer of aggregation between $T$ and $R$
4. Many features of Dryad used:
   1. *Sub graph Encapsulation*
   2. *Dynamic Refinement*
Questions:

1. By using distributed execution engines such as MapReduce and Dryad, what are the issues that are usually hard for programmers to tackle but now become the responsibilities of the these engines’ runtime systems? (the second and third paragraphs in Section 1)

2. In what aspect(s) is Dryad different MapReduce? (Section 1 from the fifth paragraph)