Parallel Computing with Dryad
Today’s Speakers

- Raman Grover
  UC Irvine
  Advisor: Prof. Michael Carey

- Sanjay Kulhari
  UC Riverside
  Advisor: Prof. Vassilis Tsotras
Acknowledgments

- Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks
- Distributed Data-Parallel Computing Using a High-Level Programming Language
- DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language
- Google Tech Talks
- MSDN Channel 9
Parallel Distributed Computing. . . Why?

- Large-scale Internet Services
  Depend on clusters of hundreds or thousands of general purpose servers.

- Future advances in local computing power:
  Increasing the number of cores on a chip rather than improving the speed or instruction-level parallelism of a single core
Hard Problems

- High-latency
- Unreliable networks
- Control of resources by separate federated or competing entities,
- Issues of identity for authentication and access control.
- The Programming Model
- Reliability, Efficiency and Scalability of the applications
Achieving Scalability

- Systems that automatically discover and exploit parallelism in sequential programs
- Those that require the developer to explicitly expose the data dependencies of a computation.

- Condor
- Shader languages developed for graphic processing units
- Parallel databases
- Google’s MapReduce system
Reasons for Success

- Developer is explicitly forced to consider the data parallelism of the computation

- The developer need have no understanding of standard concurrency mechanisms such as threads and fine-grain concurrency control

- Developers now work at a suitable level of abstraction for writing scalable applications since the resources available at execution time are not generally known at the time the code is written.
Limitations

Not a free lunch!

Restrict an application’s communication flow for different reasons:

- **GPU shader languages**
  - Strongly tied to an efficient hardware implementation

- **Map Reduce**
  - Designed for the widest possible class of developers, aims for simplicity at the expense of generality performance.

- **Parallel databases**
  - Designed for relational algebra manipulations (e.g. SQL) where
Dryad

- Control over the communication graph as well as the subroutines that live at its vertices.
- Specify an arbitrary directed acyclic graph to describe the application’s communication patterns,
- Express the data transport mechanisms (files, TCP pipes, and sharedmemory FIFOs) between the computation vertices.

- MapReduce restricts all computations to take a single input set and generate a single output set.
- SQL and shader languages allow multiple inputs but generate a single output from the user’s perspective, though SQL query plans internally use multiple-output vertices.
- Dryad is notable for allowing graph vertices (and computations in general) to use an arbitrary number of inputs and outputs.
In this talk!

- Dryad: System Overview
- Describing a Dryad Graph
- Communication Channel
- Dryad Job
- Job Execution
- Fault Tolerance
- Runtime Graph Refinement
- Experimental Evaluation
- Building on Dryad
Before we dive into Details...

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

- Dryad: 2-D
  
grep^{1000} | sed^{500} | sort^{1000} | awk^{500} | perl^{50}
Dryad = Execution Layer

Job (Application) ≈ Pipeline

Dryad ≈ Shell

Cluster ≈ Machine
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines

- 2D DAG
- multi-machine
- virtualized
Dryad Job Structure

Input files

Channels

Vertices (processes)

Stage

Output files

grep$^{1000}$ | sed$^{500}$ | sort$^{1000}$ | awk$^{500}$ | perl$^{50}$
Channels

Finite Streams of items

- Distributed filesystem (persistent)
- SMB/NTFS files (temporary)
- TCP pipes (inter-machine)
- Memory FIFOs (intra-machine)
Architecture

Job manager

control plane

cluster

data plane

Files, TCP, FIFO, Network

job schedule
Runtime

- Services
  - Name server
  - Daemon

- Job Manager
  - Centralized coordinating process
  - User application to construct graph
  - Linked with Dryad libraries for scheduling vertices

- Vertex executable
  - Dryad libraries to communicate with JM
  - User application sees channels in/out
  - Arbitrary application code, can use local FS
Job = Directed Acyclic Graph

- Processing vertices
- Channels (file, pipe, shared memory)
- Inputs
- Outputs
Scheduler keeps track of state and history of each vertex in the graph.

When a job manager fails job is terminated but scheduler can implement checkpointing or replication to avoid this.

Execution record attached with a vertex.

Execution record paired with a available computer, remote daemon is instructed to run the vertex.
Job execution (cont.)

- If an execution of a vertex fails it can start again.

- More than one instance of the vertex may be executing at the same time.

- Each vertex names its output channels uniquely using version number.

Diagram:
- Vertex
  - New execution record created and added to scheduling queue
  - Execution record paired with an available computer
  - Job manager receives periodic status updates from the vertex
  - Input ready
Fault tolerance policy

- All vertex programs are deterministic

- Every terminating execution of the job will give the same results regardless of the failures over the course of execution.

- Job manager will know in any case that something bad happened to a vertex.

- Vertices belong to stages and stage manager can take care of slow or failed vertices of a stage.
Fault tolerance policy (cont.)

- If A fails, run it again
- If A’s inputs are gone, run upstream vertices again.
- If A is slow, run another copy elsewhere and use output from whichever finishes first.
Run-time graph refinement

- To be able to scale to large input sets while conserving scarce network bandwidth.
- For associative and commutative computations aggregation tree can be helpful.
- If internal vertices perform data reduction network traffic between racks will be reduced.
- Keep refining when upstream vertices have completed.
Run-time graph refinement (cont.)

- Partial aggregation operation, to process $k$ sets in parallel.

- Data mining example follows this.

- Dynamic refinement is good because the amount of data to be written is not known in advance and also the required input channels.
Run-time graph refinement (cont.)

B vertex receives 50,000 tuples and Execute DISTINCT on them

B vertex gets 1000 tuples, runs DISTINCT

* vertex gets 30,000 tuples, runs DISTINCT and returns 500 tuples

+ vertex gets 20,000 tuples, runs DISTINCT and returns 500 tuples

Each A vertex sends 10,000 tuples
Run-time graph refinement (cont.)
Experimental evaluation

- **Hardware:**
  - Cluster of 10 computers (Sky server query experiment)
  - Cluster of 1800 computers (Data mining experiment)
  - Each computer had 2 dual core Opteron processors running at 2 GHz. i.e. 4 CPUs total.
  - 8 GB of DRAM
  - 400 GB Western Digital.
  - 1 Gbit/sec Ethernet
Case study I (Sky server 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
    \[ T = (U.color, N.neighborId) \text{ where } U.objId = N.objId \]
  - Join U+T to find
    \[ U.objId \text{ where } U.objId = T.neighborId \]
    and \( U.color \approx T.color \)
SkyServer DB query

- Took SQL plan
- Manually coded in Dryad
- Manually partitioned data

SkyServer DB query

distinct
merge outputs

select
  u.objid
from u join <temp>
where
  u.objid = <temp>.neighborobjid
and
  |u.color - <temp>.color| < d
Optimization
Optimization
The graph shows the speed-up of three different systems: Dryad In-Memory, Dryad Two-pass, and SQLServer 2005, as the number of computers increases. The speed-up is measured on the y-axis and the number of computers on the x-axis. The graph indicates that both Dryad systems scale well with increasing numbers of computers, whereas SQLServer 2005 shows a slower increase in speed-up.
Case study II - 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

P  parse lines
D  hash distribute
S  sort
C  count occurrences
MS merge sort

Each R is:

Each Q is:

Each D is:

Each S is:

Each C is:
Efficient histogram topology

P  parse lines
D  hash distribute
S  sort
C  count occurrences

MS merge sort
M  non-deterministic merge

Each Q’ is:

Each T is:

Each R is:

Each S is:

Each P is:

Each C is:

Each MS is:
Final histogram refinement

1,800 computers
43,171 vertices
11,072 processes
11.5 minutes
Optimizing Dryad applications

- General-purpose refinement rules
- Processes formed from sub graphs
  - Re-arrange computations, change I/O type
- Application code not modified
  - System at liberty to make optimization choices
- High-level front ends hide this from user
All this sounds good! But how do I interact with Dryad?

- Nebula scripting language
  - Allows users to specify a computation as a series of stages each taking input from one or more previous stages or files system.
- Dryad as generalization of UNIX piping mechanism.
- Writing distributed applications using perl or grep.
- Also a front end that uses perl scripts and sql select, project and join.
Interacting with Dryad (Cont.)

- Integration with SQL Server
  - SQL Server Integration Services (SSIS) supports work-flow based application programming on single instance of SQL server.
  - SSIS input graph generated and tested on a single computer.
  - SSIS graph is run in distributed fashion using dryad.
  - Each Dryad vertex is an instance of SQL server running an SSIS sub graph of the complete Job.
  - Deployed in live production system.
LINQ

- Microsoft’s Language INtegrated Query
  - Available in Visual Studio products
- A set of operators to manipulate datasets in .NET
  - Support traditional relational operators
    - Select, Join, GroupBy, Aggregate, etc.
  - Integrated into .NET programming languages
    - Programs can call operators
    - Operators can invoke arbitrary .NET functions
- Data model
  - Data elements are strongly typed .NET objects
  - Much more expressive than SQL tables
- Highly extensible
  - Add new custom operators
  - Add new execution providers
LINQ System Architecture

Local machine

Execution engines: Scalability

- DryadLINQ
- PLINQ
- LINQ-to-SQL
- LINQ-to-Obj

.NET program (C#, VB, F#, etc)

Query

Objects

Cluster

Multi-core

Single-core
DryadLINQ

- Automatically distribute a LINQ program
- 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
- Few Dryad-specific extensions
  - Same source program runs on single-core through multi-core up to cluster
DryadLINQ System Architecture

Client machine

.NET program
ToTable
foreach
.Net Objects
DryadLINQ
Distributed query plan
Output DryadTable

Data center

Query
Vertex code
Input Tables
JM
Dryad Execution
Output Tables
Results

Invoked
Word Count in DryadLINQ

Count word frequency in a set of documents:

```csharp
var docs = DryadLinq.GetTable<Doc>("file://docs.txt");
var words = docs.SelectMany(doc => doc.words);
var groups = words.GroupBy(word => word);
var counts = groups.Select(g => new WordCount(g.Key, g.Count()));

counts.ToDryadTable("counts.txt");
```
Execution Plan for Word Count

- SelectMany
- sort
- groupby
- count
- distribute
- mergesort
- groupby
- Sum
- pipelined
- pipelined
Execution Plan for Word Count
Automatic query plan generation

Distributed query execution by Dryad

LINQ query

var logentries =
    from line in logs
    where !
    line.StartsWith("#")
    select new LogEntry(line);

Query plan

- select
- where
- logs

Dryad
How does it work?

- Sequential code “operates” on datasets
- But really just builds an expression graph
  - Lazy evaluation
- When a result is retrieved
  - Entire graph is handed to DryadLINQ
  - Optimizer builds efficient DAG
  - Program is executed on cluster
Future Directions

- Goal: Use a cluster as if it is a single computer
  - Dryad/DryadLINQ represent a modest step

- On-going research
  - What can we write with DryadLINQ?
    - Where and how to generalize the programming model?
  - Performance, usability, etc.
    - How to debug/profile/analyze DryadLINQ apps?
  - Job scheduling
    - How to schedule/execute N concurrent jobs?
  - Caching and incremental computation
    - How to reuse previously computed results?
  - Static program checking
    - A very compelling case for program analysis?
    - Better catch bugs statically than fighting them in the cloud?
Conclusions

- **Goal:** Use a compute cluster as if it is a single computer
  - Dryad/DryadLINQ represent a significant step
- Requires close collaborations across many fields of computing, including
  - Distributed systems
  - Distributed and parallel databases
  - Programming language design and analysis