Distributed Data-Parallel Computing Using a High-Level Programming Language

Slides adapted from those of Michal Isard and Yuan Yu

Distributed Data-Parallel Computing
• Research problem: How to write distributed data-parallel programs for a compute cluster?
• DryadLINQ programming model
  – Sequential, single machine programming abstraction
  – Same program runs on single-core, multi-core, or cluster
  – Familiar programming languages
  – Familiar development environment
• Dryad
  – Provides a general, flexible execution layer
  – Automatic handling
  • Scheduling
  • Distribution
  • Fault-tolerance

Dryad and DryadLINQ
DryadLINQ provides automatic query plan generation
Dryad provides automatic distributed execution

Dryad
• Similar goals as MapReduce
  – focus on throughput, not latency
  • Why is throughput important and 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
- Maintains job graph and schedules vertices
- Daemon process (D) executes vertices

**Job = Directed Acyclic Graph**

- Outputs
- Processing vertices
- Channels (file, pipe, shared memory)
- Inputs

**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.neighborId where U.objId = N.objId, U.color
  - Join U+T to find U.objId where U.objId = T.neighborId and U.color ≈ T.color

SkyServer query
- select u.color, n.neighborobjid from u join n where u.objid = n.objid

Immutable Input
- Assumes that inputs are immutable
  - High performance: Scales out to shared-nothing clusters made up of thousands of machines
  - Significantly simplifies the design/implementation
    - Simple fault-tolerant story
    - No need to handle complex transaction and synchronization
  - Good for processing largely static datasets
  - Not suitable for fine-grain, frequent updates
DryadLINQ

**LINQ**

- Microsoft’s Language INtegrated Query
  - Available in .NET3.5 and Visual Studio 2008
- 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 invoke operators
    - Operators can invoke arbitrary .NET functions
- Data model
  - Data elements are strongly typed .NET objects
  - Much more expressive than relational tables
    - For example, nested data structures

DryadLINQ

- More general than distributed SQL
  - Inherits flexible C# type system and libraries
  - Data-clustering, inference, ...
- Uniform data-parallel programming model
  - From SMP to clusters

DryadLINQ = LINQ + Dryad

**DryadLINQ System Architecture**

---

```csharp
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 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. Group by edge destination
4. Aggregate into ranks
5. Repeat

One PageRank Step in DryadLINQ

```csharp
// 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
// 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());
}
```

Multi-Iteration PageRank

Lessons of Dryad/DryadLINQ

- Acyclic dataflow graph is a powerful computation model
  - Trees
  - SQL
  - Cyclic graphs
  - ML
- 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)
### Questions on Dryad/DryadLINQ

- **What do you think about DryadLINQ?**
  - Sweet spot between imperative and declarative?
  - Supports complex objects
    - What about access methods?
- **Hard problems**
  - Loops
    - No data dependent termination
  - Query optimization
    - Not dynamic

---

### Questions on Dryad/DryadLINQ

- **Comparison**
  - Two language approach
    - PigLatin
    - SCOPE
  - Map Reduce Online (Eurosys’10)
- **Centralized scheduler**
  - Quincy (SOSP’09)