Dryad and DryadLINQ

General-purpose Distributed Computing using a High-level Language

Michael Isard
Microsoft Research Silicon Valley

Distributed Data-Parallel Computing

- Workloads beyond standard SQL, HPC
  - Data-mining, graph analysis, …
  - Complex, long-lived application software
- Cloud (shared clusters)
  - Transparent scaling
  - Resource virtualization
- Commodity hardware
  - Fault tolerance with good performance

Talk overview

- Part I
  - High-level language: LINQ
  - Computational model: DAG
  - Execution layer: Dryad+Quincy
- Part II
  - Dryad systems issues
  - Comparison with MapReduce
  - DryadLINQ demo

LINQ

- Microsoft's Language INtegrated Query
- Operators to manipulate datasets in .NET
  - Dataset is a first-class abstraction
  - Select, Join, GroupBy, Aggregate, etc.
  - *Set at a time*, instead of looping over *Object at a time*
- 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
- Extensible
  - Add new operators and implementations

Aggregating partial sums

```
class PartialSum { public int sum; public int count; };

static double MergeSums(PartialSum[] sums)
{
   int totalSum = 0, totalCount = 0;
   int i;
   for (i = 0; i < sums.Length; ++i)
   {
      totalSum += sums[i].sum;
      totalCount += sums[i].count;
   }
   return (double) totalSum / (double) totalCount;
}
```

Aggregating partial sums

```
class PartialSum { public int sum; public int count; };

static double MergeSums(PartialSum[] sums)
{
   return (double) sums.Select(x => x.sum).Sum() /
               (double) sums.Select(x => x.count).Sum();
}
```

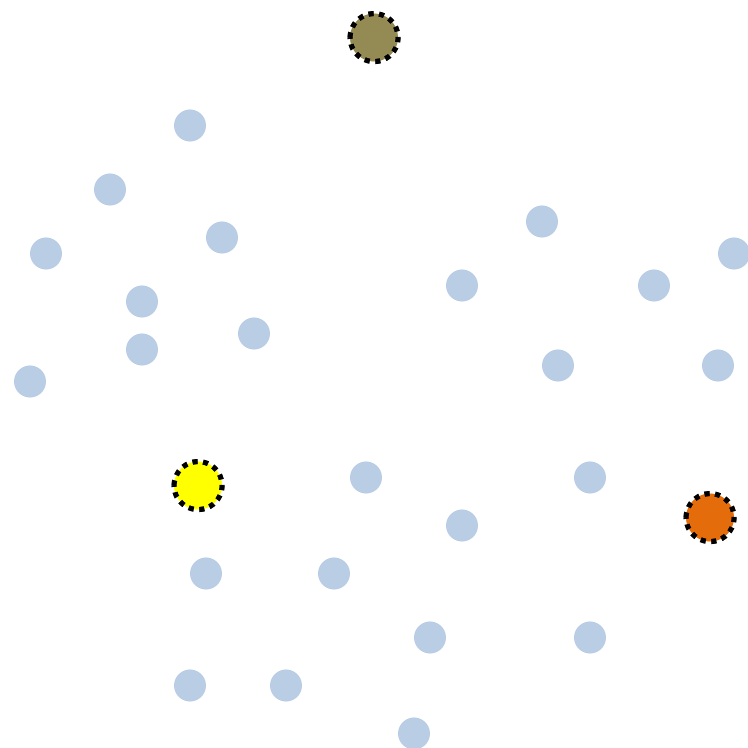
Convenient syntax

```
var words =
    tableOfLines.SelectMany(l => l.Split(' ')).GroupBy(w => w);
```

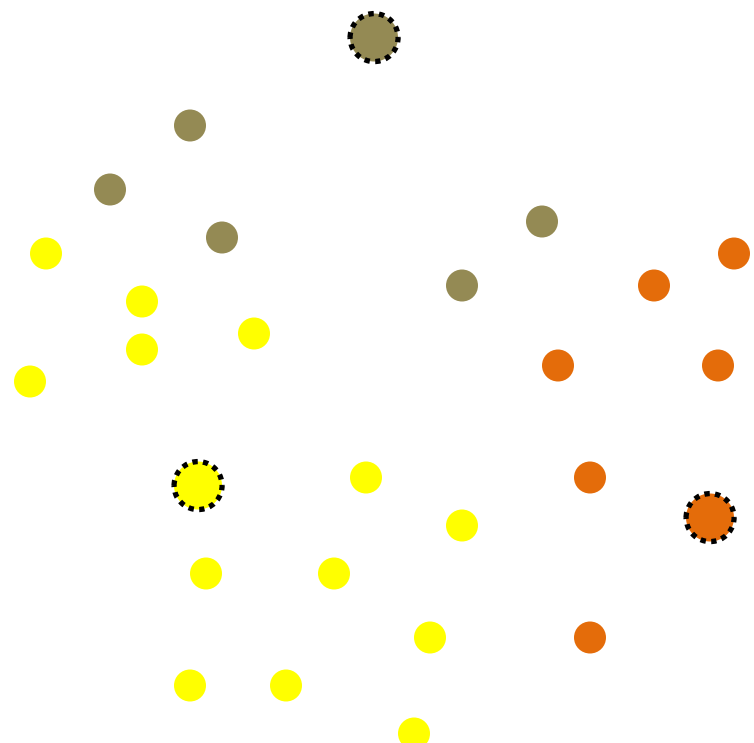
Convenient syntax

```
var words =
    tableOfLines.SelectMany(l => l.Split(' ')).GroupBy(w => w);

IQueryable<IGrouping<string,string>> words =
        tableOfLines.SelectMany(mySplitFunction).GroupBy(myStringIdentity);

IEnumerable<string> mySplitFunction(string line)
{
    return line.Split(' ');
}

string myStringIdentity(string word)
{
    return word;
}
```

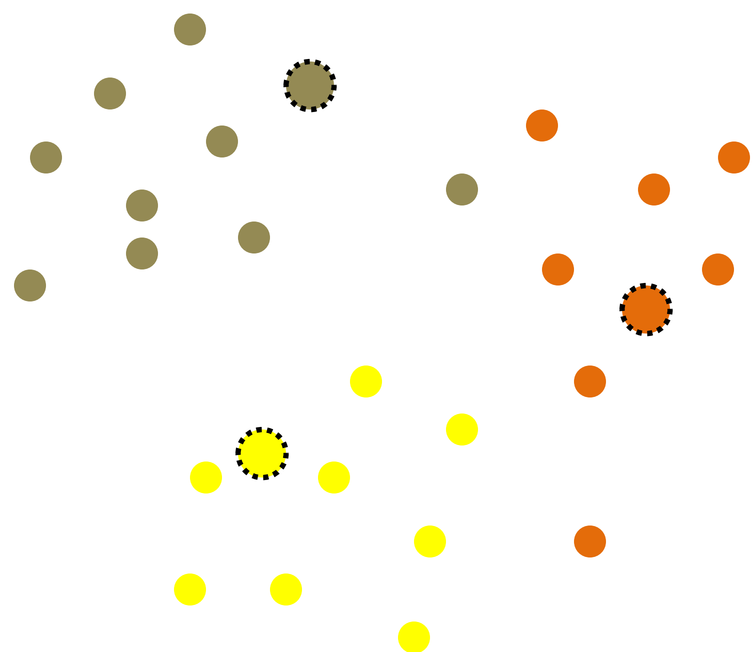
K-means algorithm



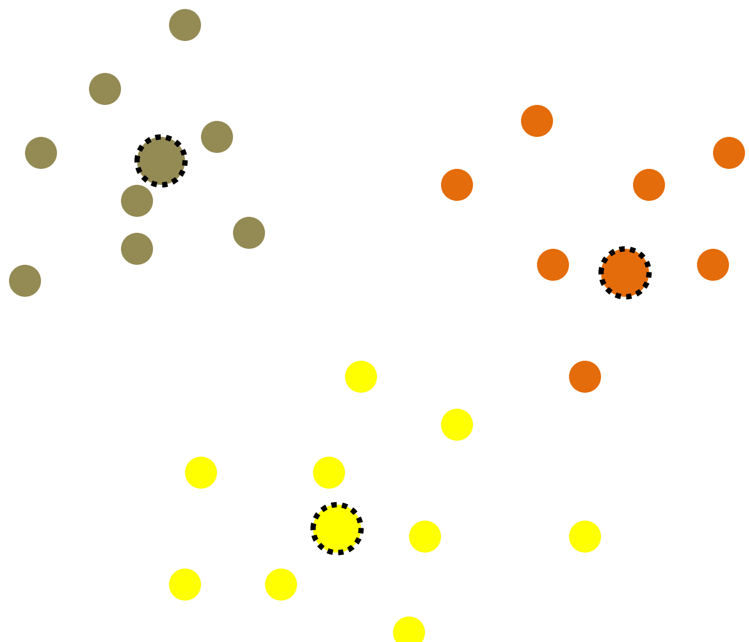
K-means algorithm



K-means algorithm



K-means algorithm



K-means helper functions

```
class Vector { … }

Vector Mean(IEnumerable<Vector> set) {
    Vector sum = set.Aggregate( (x, y) => x + y );
    return sum / set.Count();
}

Vector NearestNeighbor(Vector vect, IEnumerable<Vector> set) {
    return set.Min( e => (e - vect).L2Norm() );
}
```

K-means algorithm

```
IEnumerable<Vector> kMeansStep(IEnumerable<Vector> vectors,
                               IEnumerable<Vector> centers) {
    var clusters = vectors.GroupBy(
        vector => NearestNeighbor(vector, centers).VectorId);
    return clusters.Select(cluster => Mean(cluster));
}

IEnumerable<Vector> kMeans(IEnumerable<Vector> vectors,
                           IEnumerable<Vector> centers) {
    for (int i = 0; i < iterations; i++) centers = kMeansStep(vectors, centers);
    return centers;
}
```

Data mining, machine learning, …

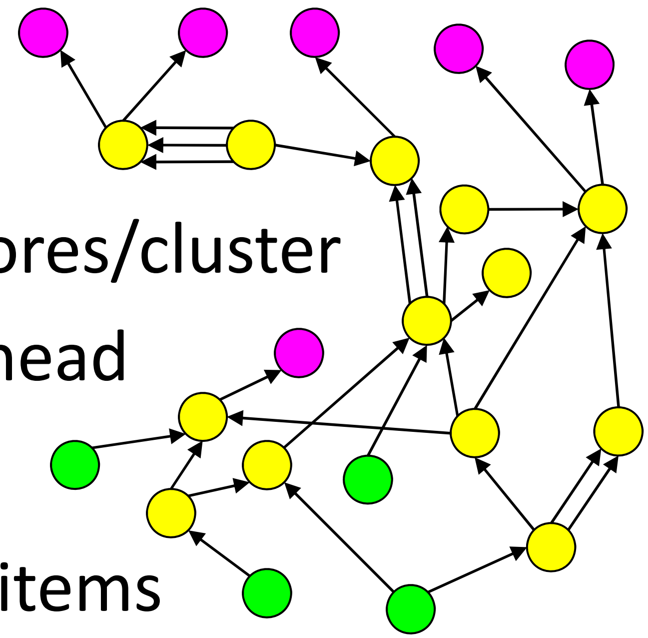
- Decision-tree training
- SVD
- Power iteration (PageRank)
- Image feature extraction/indexing/clustering
- Network trace analysis
- Light-field simulation
- …

Talk overview

- Part I
  - High-level language: LINQ
  - Computational model: DAG
  - Execution layer: Dryad+Quincy
- Part II
  - Dryad systems issues
  - Comparison with MapReduce
  - DryadLINQ demo

Computational model: DAG

- Distributed processing
  - Partition computation across cores/cluster
  - Minimize communication overhead
- Directed-acyclic graph
  - Edge is finite sequence of data items
  - Vertex is computation over input edge sequences



DAG abstraction

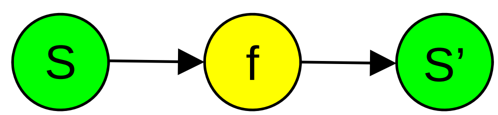
- Explicit dataflow
  - Exposes dependencies within computation
- Absence of cycles
  - Allows re-execution for fault-tolerance
  - Simplifies scheduling: no deadlock
- Cycles can often be replaced by unrolling
  - Unsuitable for fine-grain inner loops
- Very popular
  - Databases, functional languages, …

Map

- Independent transformation of dataset
  – for each x in S, output x' = f(x)
- E.g. simple grep for word w
  – output line x only if x contains w

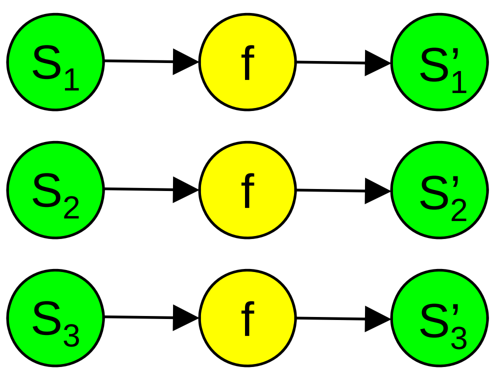
Map

- Independent transformation of dataset
  - for each x in S, output x’ = f(x)
- E.g. simple grep for word w
  - output line x only if x contains w



Map

- Independent transformation of dataset
  - for each x in S, output x' = f(x)
- E.g. simple grep for word w
  - output line x only if x contains w

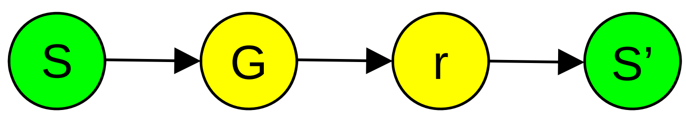


Reduce

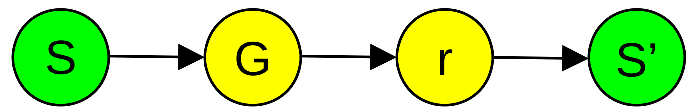
- Grouping plus aggregation
  - 1) Group x in S according to key selector k(x)
  - 2) For each group g, output r(g)
- E.g. simple word count
  - group by k(x) = x
  - for each group g output key (word) and count of g

Reduce

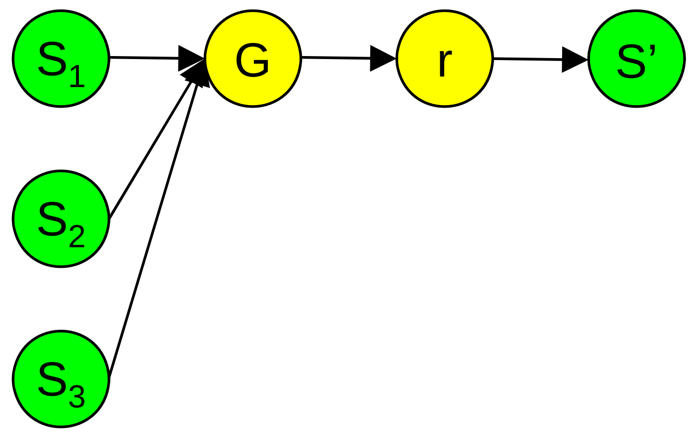
- Grouping plus aggregation
  - 1) Group x in S according to key selector k(x)
  - 2) For each group g, output r(g)
- E.g. simple word count
  - group by k(x) = x
  - for each group g output key (word) and count of g



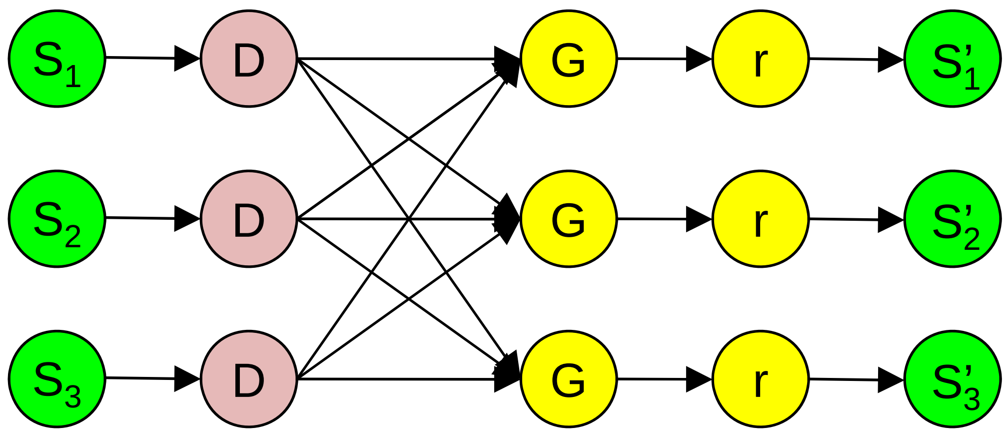
Reduce



Reduce

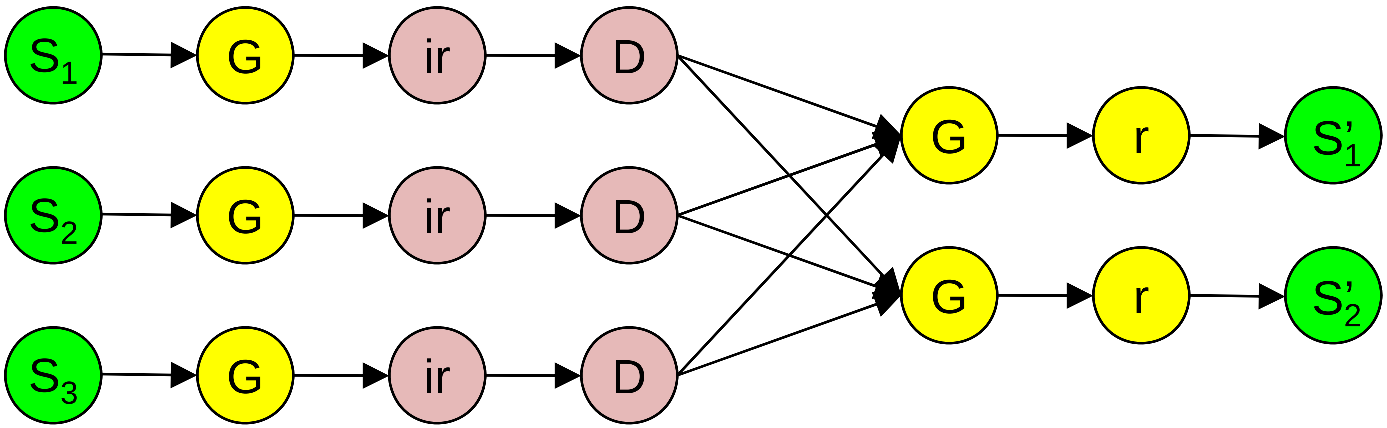


Reduce



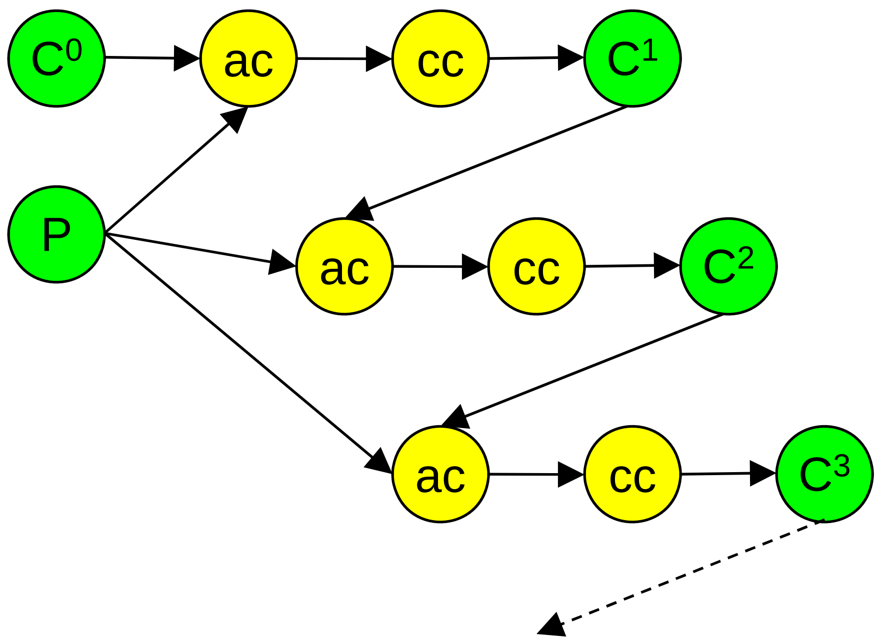
D is *distribute*, e.g. by hash or range

Reduce

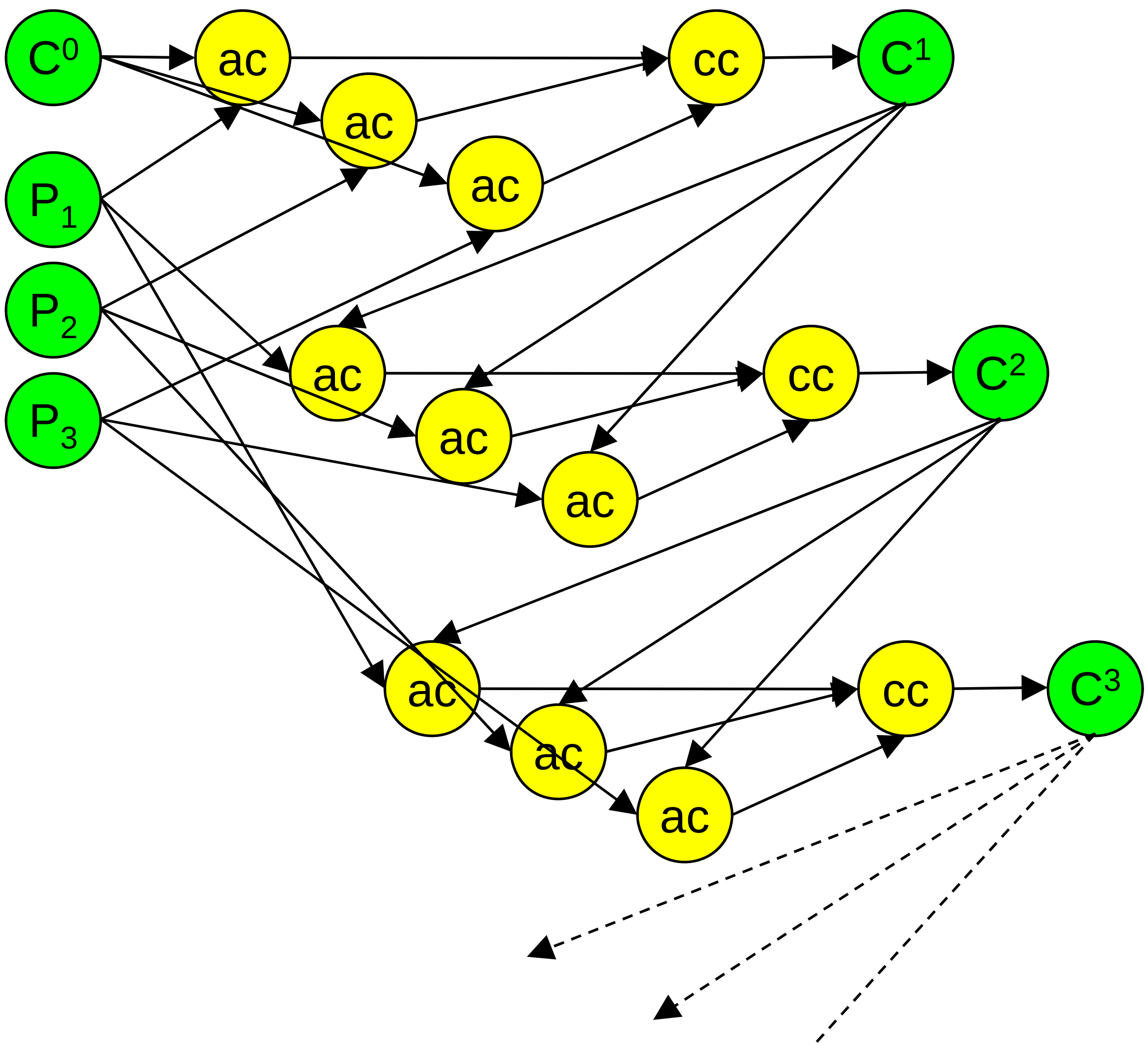


ir is *initial reduce*, e.g. compute a partial sum

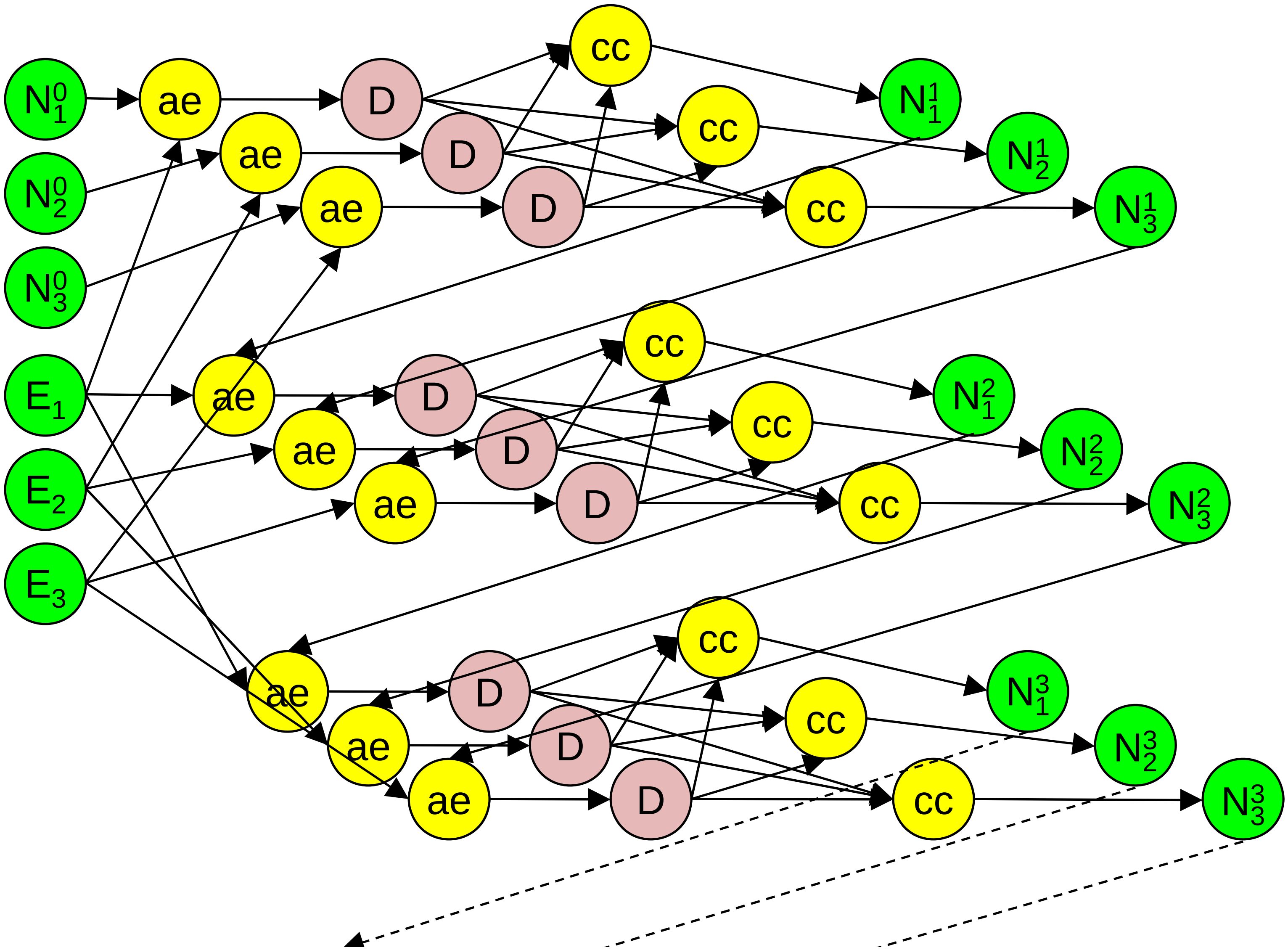
K-means



K-means



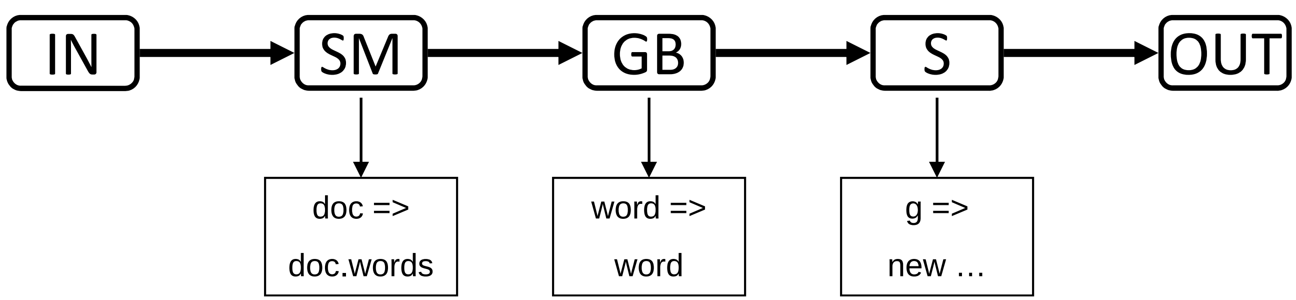
PageRank



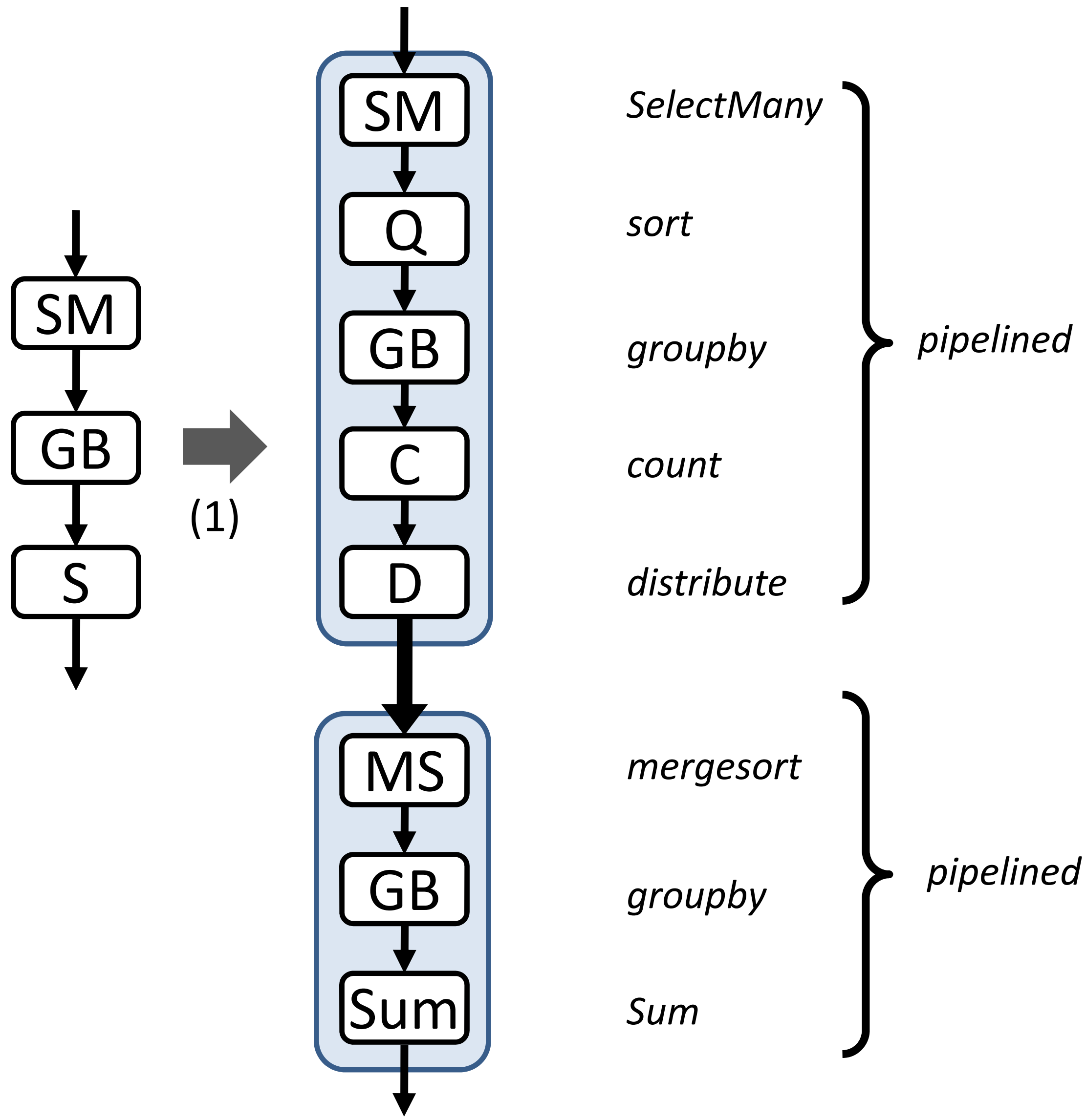
Distributed Word Count

Count word frequency in a set of documents:

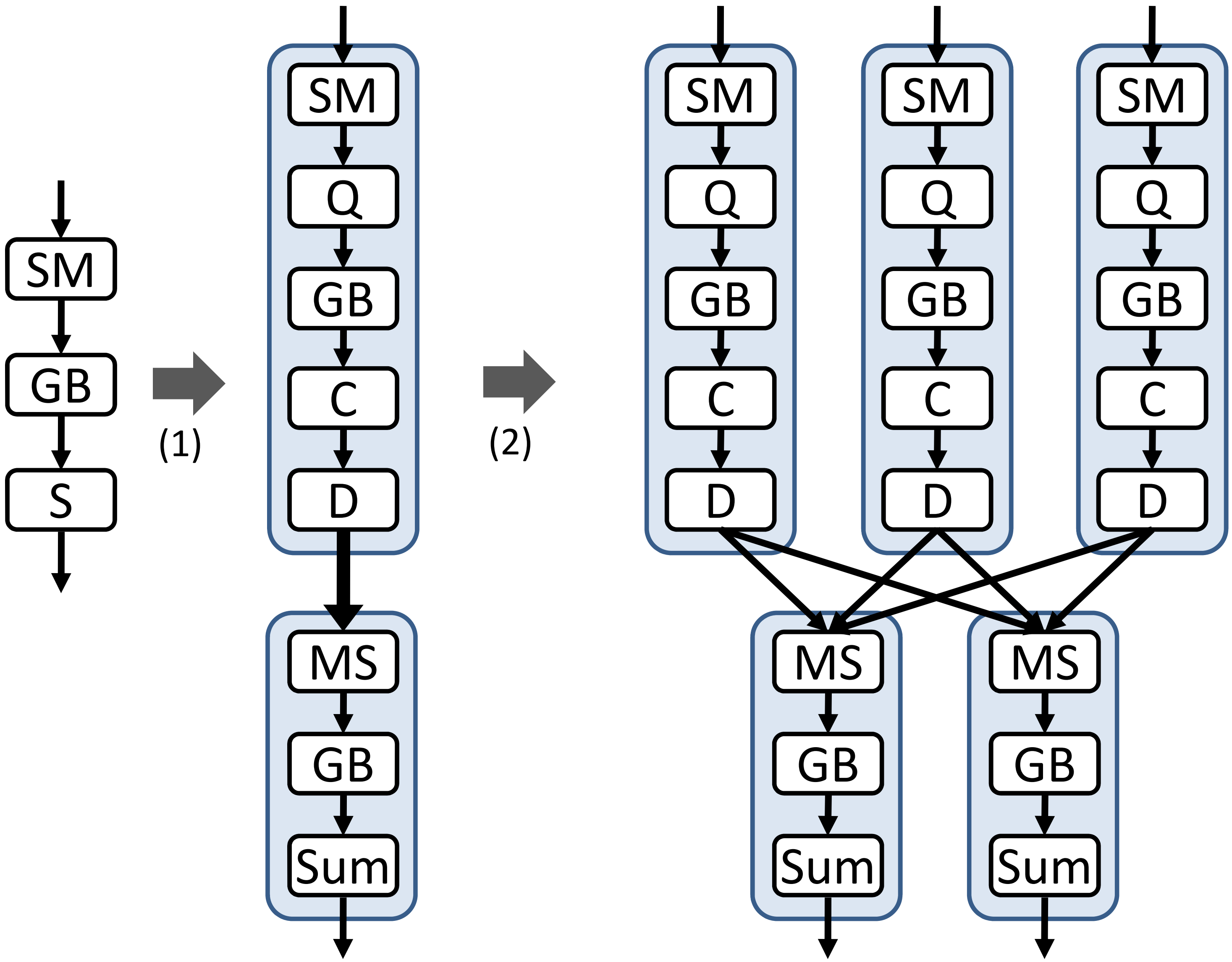
```
var words = docs.SelectMany(doc => doc.words);
var groups = words.GroupBy(word => word);
var counts = groups.Select(g => new WordCount(g.Key, g.Count()));
```



Execution Plan for Word Count



Execution Plan for Word Count



Talk overview

- Part I
  - High-level language: LINQ
  - Computational model: DAG
  - Execution layer: Dryad+Quincy
- Part II
  - Dryad systems issues
  - Comparison with MapReduce
  - DryadLINQ demo

Dryad

- General-purpose execution engine
  - Batch processing on immutable datasets
  - Well-tested on large clusters
- Automatically handles
  - Fault tolerance
  - Distribution of code and intermediate data
  - Scheduling of work to resources

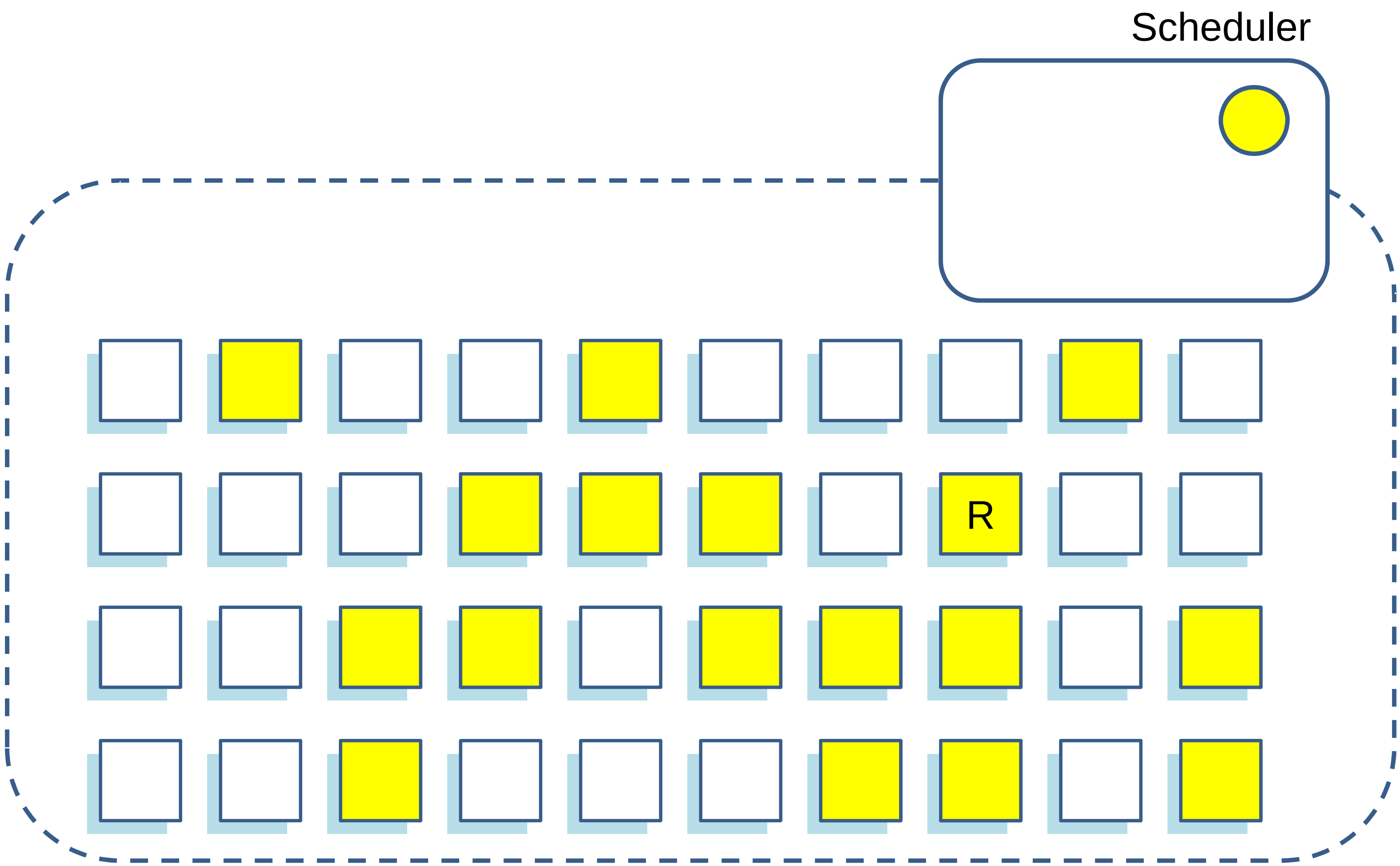
Fault tolerance

- Buffer data in (some) edges
- Re-execute on failure using buffered data
- Speculatively re-execute for stragglers

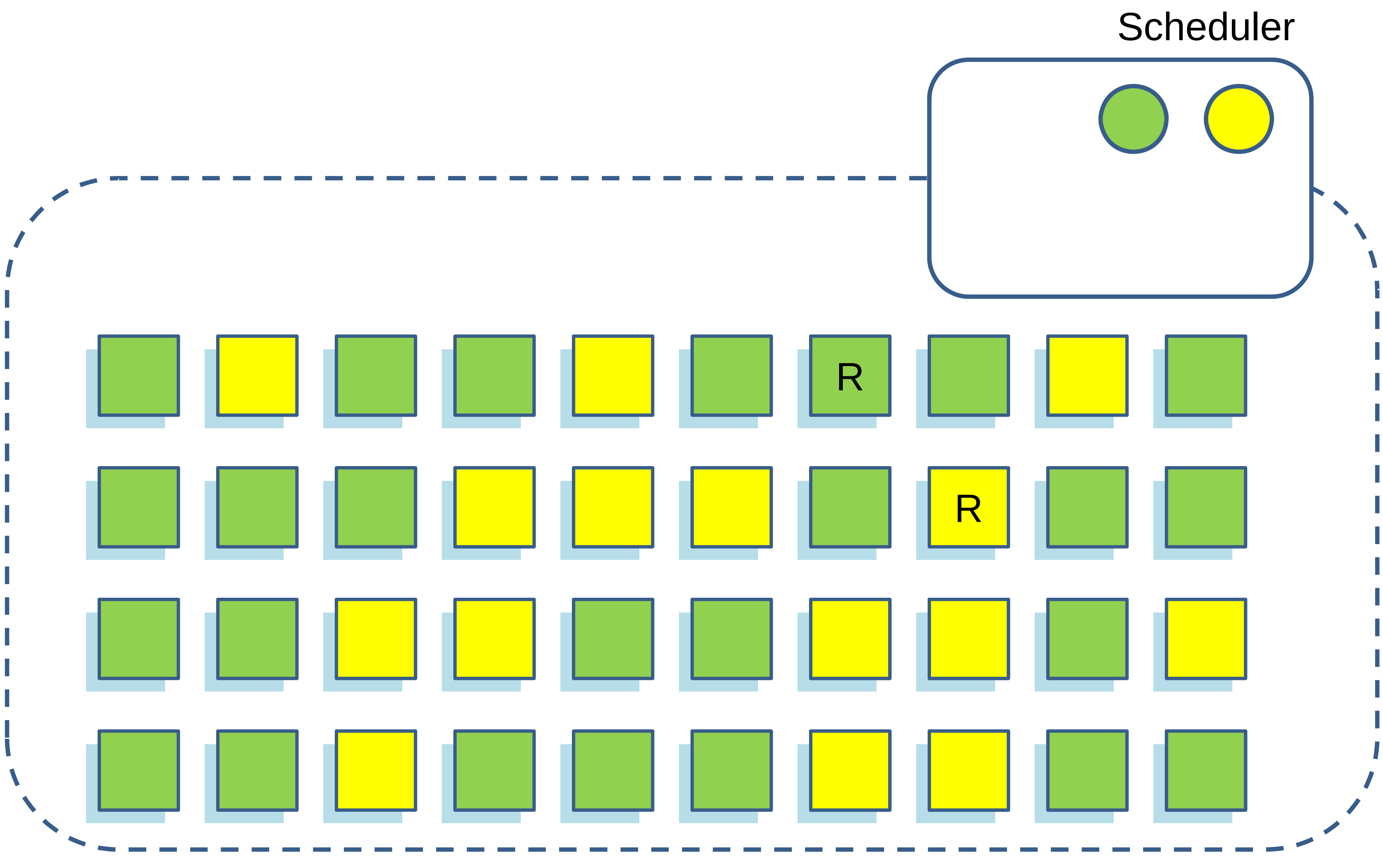
Rewrite graph at runtime

- Loop unrolling with convergence tests
- Adapt partitioning scheme at run time
  - Choose #partitions based on runtime data volume
  - Broadcast Join vs. Hash Join, etc.
- Adaptive aggregation and distribution trees
  - Based on data skew and network topology
- Load balancing
  - Data/processing skew (cf work-stealing)

Dryad System Architecture



Dryad System Architecture



Quincy DAG Scheduler

- Data locality and fairness (SLAs)
- SOSP 2009

Production system

- Dryad well-tested, scalable
  - Daily use supporting Bing for over 3 years
  - Clusters with >10k computers
- Applicable to large number of computations
  - 250 computer cluster at MSR SVC, Mar->Nov 09
    - 15k jobs (tens of millions of processes executed)
    - Hundreds of distinct programs
      - Network trace analysis, privacy-preserving inference, light-transport simulation, decision-tree training, deep belief network training, image feature extraction, …

Conclusion

- DryadLINQ supports many computations
  - Easy to use, flexible
- DAG-structured jobs scale to large clusters
  - Transient failures common, disk failures daily
- Publically available for download

  http://connect.microsoft.com/Dryad

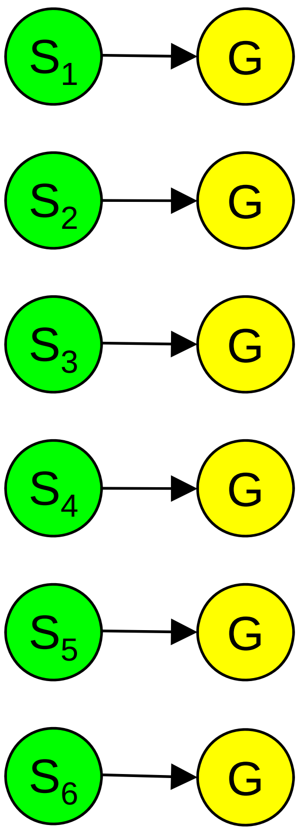
Talk overview

- Part I
  - High-level language: LINQ
  - Computational model: DAG
  - Execution layer: Dryad+Quincy
- Part II
  - Dryad systems issues
  - Comparison with MapReduce
  - DryadLINQ demo

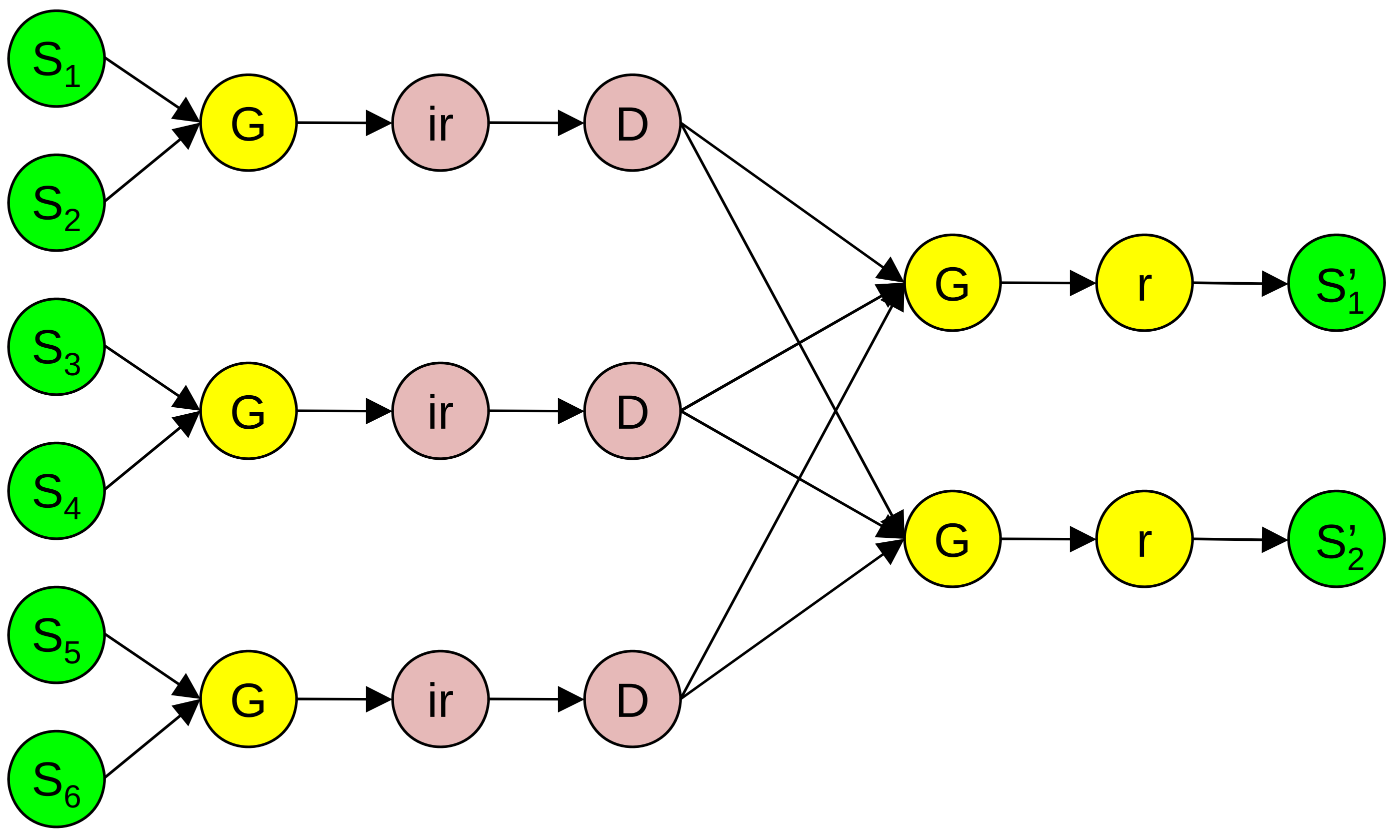
Dryad Inputs and Outputs

- Partitioned data set
  - Records do not cross partition boundaries
  - Data on compute machines: NTFS, SQLServer, …
- Optional semantics
  - Hash-partition, range-partition, sorted, etc.
- Loading external data
  - Partitioning "automatic"
  - File system chooses sensible partition sizes
  - Or known partitioning from user

Partitioning driven by data

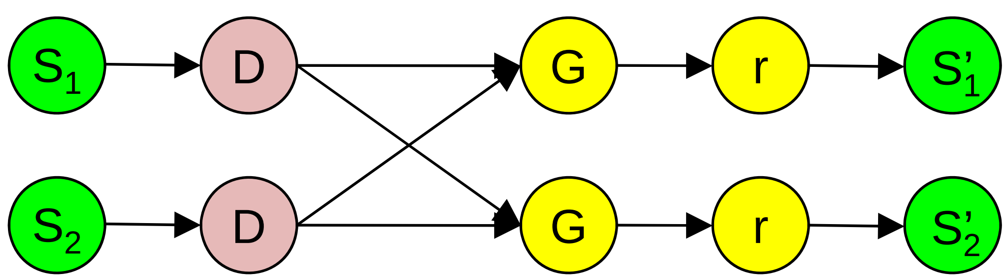


Partitioning driven by data

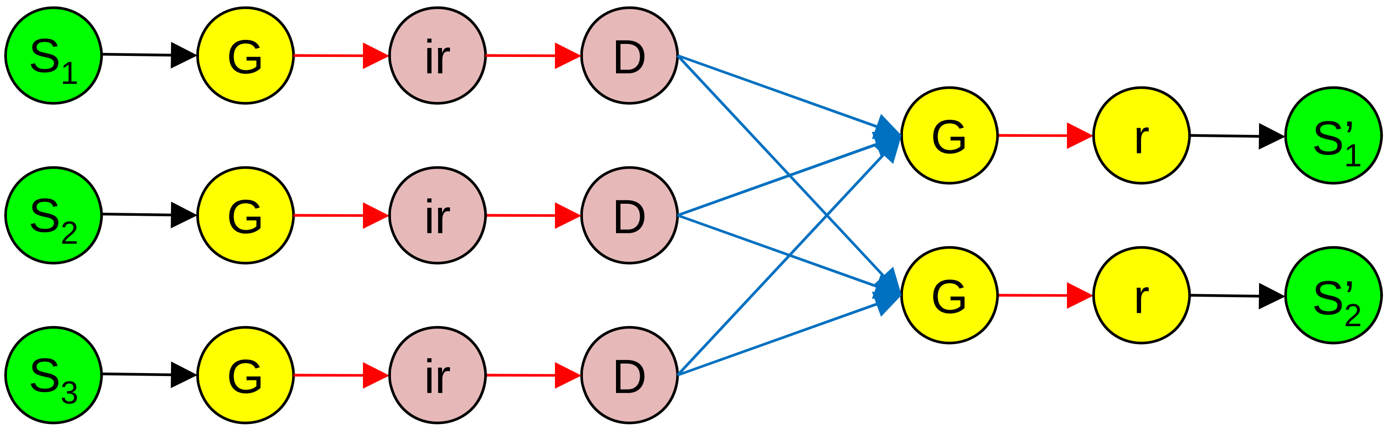


Push vs Pull

- Databases typically ‘pull’ using iterator model
  - Avoids buffering
  - Can prevent unnecessary computation
- But DAG must be fully materialized
  - Complicates rewriting
  - Prevents resource virtualization in shared cluster



Channel abstraction



Push vs Pull

- Channel types define connected component
  - Shared-memory or TCP must be gang-scheduled
- Pull within gang, push between gangs

Fault tolerance

- Buffer data in (some) edges
- Re-execute on failure using buffered data
- Speculatively re-execute for stragglers
- 'Push' model makes this very simple

DryadLINQ Internals

- Distributed execution plan
  - Static optimizations: pipelining, eager aggregation, etc.
  - Dynamic optimizations: data-dependent partitioning, dynamic aggregation, etc.
- Automatic code generation
  - Vertex code that runs on vertices
  - Channel serialization code
  - Callback code for runtime optimizations
  - Automatically distributed to cluster machines
- Separate LINQ query from its local context
  - Distribute referenced objects to cluster machines
  - Distribute application DLLs to cluster machines

Decomposable Functions

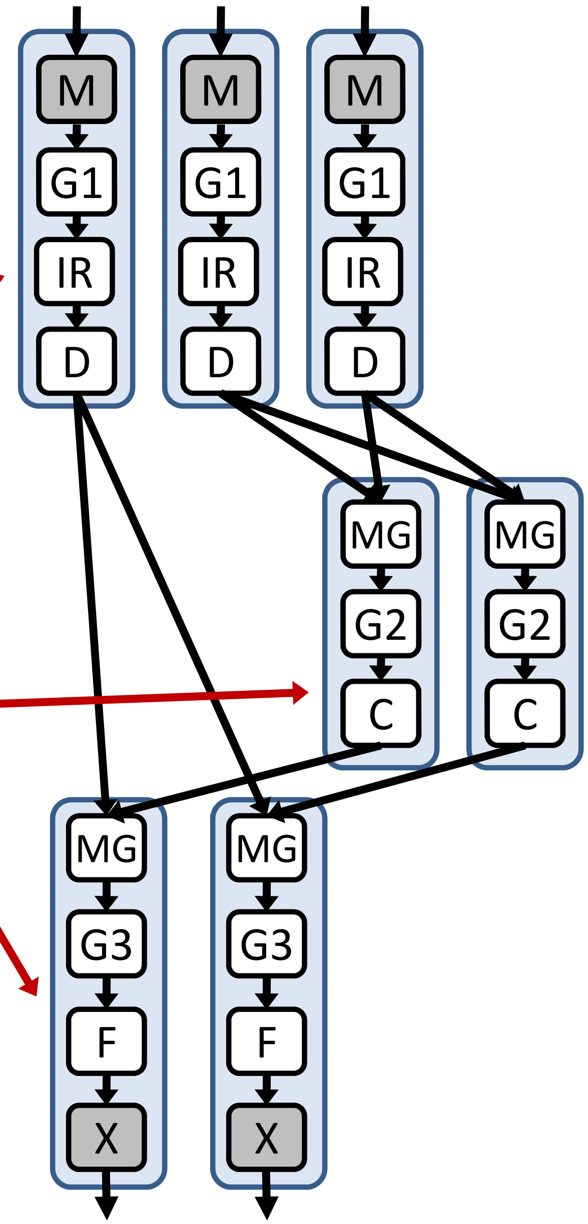
- Roughly, a function H is decomposable if it can be expressed as composition of two functions IR and C such that
  - IR is commutative
  - C is commutative and associative
- Some decomposable functions
  - Sum: IR = Sum, C = Sum
  - Count: IR = Count, C = Sum
  - OrderBy.Take: IR = OrderBy.Take,
    C = SelectMany.OrderBy.Take

Two Key Questions

- How do we decompose a function?
  - Two interfaces: iterator and accumulator
  - Choice of interfaces can have significant impact on performance
- How do we deal with user-defined functions?
  - Try to infer automatically
  - Provide a good annotation mechanism

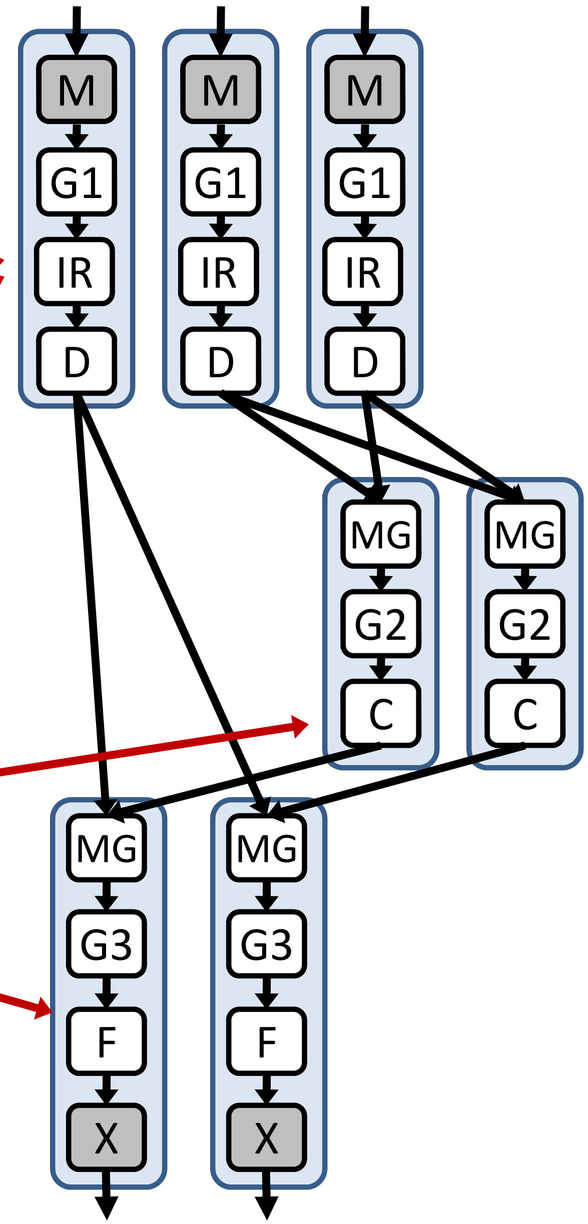
Iterator Interface in DryadLINQ

```
[Decomposable("InitialReduce", "Combine")]
public static IntPair SumAndCount(IEnumerable<int>
g) {
    return new IntPair(g.Sum(), g.Count());
}

public static IntPair InitialReduce(IEnumerable<int> g)
{
    return new IntPair(g.Sum(), g.Count());
}

public static IntPair Combine(IEnumerable<IntPair> g)
{
    return new IntPair(g.Select(x => x.first).Sum(),
                                     g.Select(x =>
x.second).Sum());
}
```



Accumulator Interface in DryadLINQ

```
[Decomposable("Initialize", "Iterate", "Merge")]
public static IntPair SumAndCount(IEnumerable<int>
g) {
    return new IntPair(g.Sum(), g.Count());
}

public static IntPair Initialize() {
    return new IntPair(0, 0);
}

public static IntPair Iterate(IntPair x, int r) {
    x.first += r;
    x.second += 1;
    return x;
}

public static IntPair Merge(IntPair x, IntPair o) {
    x.first += o.first;
    x.second += o.second;
    return x;
}
```



Iterator PartialSort

- G1+IR and G2+C
  - Keep only a fixed number of chunks in memory
  - Chunks are processed in parallel: sorted, grouped, reduced by IR or C, and emitted
- G3+F
  - Read the entire input into memory, perform a parallel sort, and apply F to each group
- Observations
  - G1+IR can always be pipelined with upstream
  - G3+F can often be pipelined with downstream
  - G1+IR may have poor data reduction
  - PartialSort is the closest to MapReduce

Accumulator FullHash

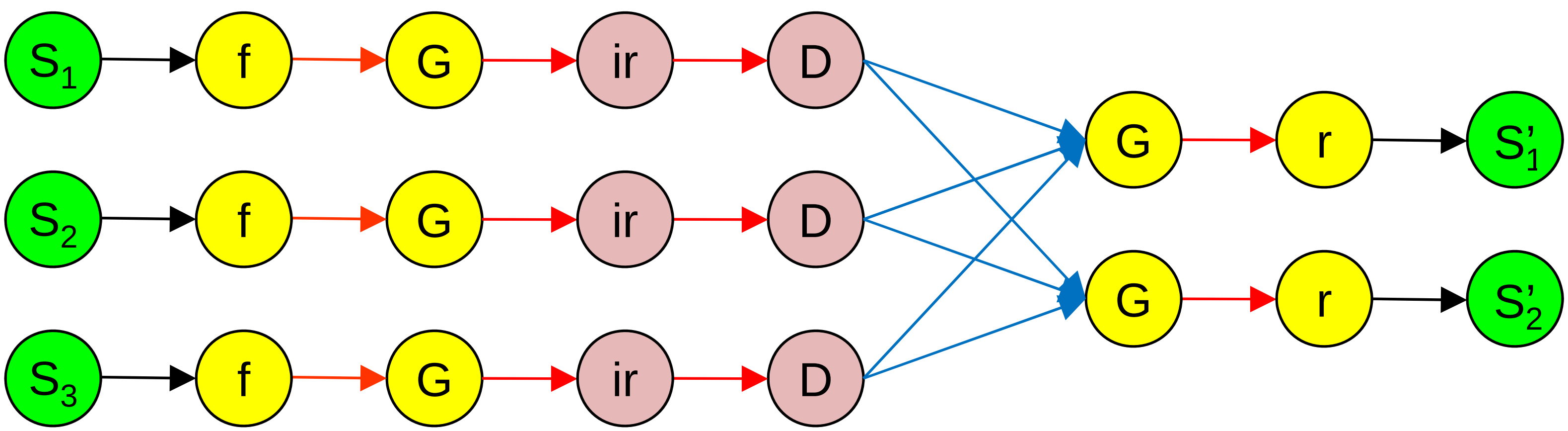
- G1+IR, G2+C, and G3+F
  - Build an in-memory parallel hash table: one accumulator object/key
  - Each input record is “accumulated” into its accumulator object, and then discarded
  - Output the hash table when all records are processed
- Observations
  - Optimal data reduction for G1+IR
  - Memory usage proportional to the number of unique keys, not records
    - So, we by default enable upstream and downstream pipelining
  - Used by DB2 and Oracle

Talk overview

- Part I
  - High-level language: LINQ
  - Computational model: DAG
  - Execution layer: Dryad+Quincy
- Part II
  - Dryad systems issues
  - Comparison with MapReduce
  - DryadLINQ demo

MapReduce (Hadoop)

- MapReduce restricts
  - Topology of DAG
  - Semantics of function in compute vertex
- Sequence of instances for non-trivial tasks



MapReduce language complexity

- Simple to describe MapReduce model
- Can be hard to map algorithm to framework
  - cf k-means: combine C+P, broadcast C, iterate, …
  - HIVE, PigLatin etc. mitigate programming issues

MapReduce system complexity

- Simple to describe MapReduce system
- Implementation not uniform
  - Different fault-tolerance for mappers, reducers
  - Add more special cases for performance
    - Hadoop introducing TCP channels, pipelines, …
  - Dryad has same state machine everywhere

DryadLINQ demo

Conclusions

- High-level language is good
  - For ease of use, maintainability, expressiveness
- Computational abstraction is important
  - Suitable target for compiler, not developer
    - Common patterns should be efficient
    - Optimization should be easy
- LINQ is a pretty good language abstraction
- DAG is a very good computational model