Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks

Michael Isard
Microsoft Research, Silicon Valley

Mihai Budiu
Microsoft Research, Silicon Valley

Yuan Yu
Microsoft Research, Silicon Valley

Andrew Birrell
Microsoft Research, Silicon Valley

Dennis Fetterly
Microsoft Research, Silicon Valley

ABSTRACT

Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs.

The vertices provided by the application developer are quite simple and are usually written as sequential programs with no thread creation or locking. Concurrency arises from Dryad scheduling vertices to run simultaneously on multiple computers, or on multiple CPU cores within a computer. The application can discover the size and placement of data at run time, and modify the graph as the computation progresses to make efficient use of the available resources.

Dryad is designed to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers. The Dryad execution engine handles all the difficult problems of creating a large distributed, concurrent application: scheduling the use of computers and their CPUs, recovering from communication or computer failures, and transporting data between vertices.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Distributed programming*

General Terms

Performance, Design, Reliability

Keywords

Concurrency, Distributed Programming, Dataflow, Cluster Computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003 ...$5.00.

1. INTRODUCTION

The Dryad project addresses a long-standing problem: how can we make it easier for developers to write efficient parallel and distributed applications? We are motivated both by the emergence of large-scale internet services that depend on clusters of hundreds or thousands of general-purpose servers, and also by the 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 of a single core [3]. Both of these scenarios involve resources that are in a single administrative domain, connected using a known, high-performance communication topology, under centralized management and control. In such cases many of the hard problems that arise in wide-area distributed systems may be sidestepped: these include high-latency and unreliable networks, control of resources by separate federated or competing entities, and issues of identity for authentication and access control. Our primary focus is instead on the simplicity of the programming model and the reliability, efficiency and scalability of the applications.

For many resource-intensive applications, the simplest way to achieve scalable performance is to exploit data parallelism. There has historically been a great deal of work in the parallel computing community both on systems that automatically discover and exploit parallelism in sequential programs, and on those that require the developer to explicitly expose the data dependencies of a computation. There are still limitations to the power of fully-automatic parallelization, and so we build mainly on ideas from the latter research tradition. Condor [37] was an early example of such a system in a distributed setting, and we take more direct inspiration from three other models: shader languages developed for graphic processing units (GPUs) [30, 36], Google's MapReduce system [16], and parallel databases [18]. In all these programming paradigms, the system dictates a communication graph, but makes it simple for the developer to supply subroutines to be executed at specified graph vertices. All three have demonstrated great success, in that large numbers of developers have been able to write concurrent software that is reliably executed in a distributed fashion.

We believe that a major reason for the success of GPU shader languages, MapReduce and parallel databases is that the developer is explicitly forced to consider the data parallelism of the computation. Once an application is cast into this framework, the system is automatically able to provide the necessary scheduling and distribution. The developer

need have no understanding of standard concurrency mechanisms such as threads and fine-grain concurrency control, which are known to be difficult to program correctly. Instead the system runtime abstracts these issues from the developer, and also deals with many of the hardest distributed computing problems, most notably resource allocation, scheduling, and the transient or permanent failure of a subset of components in the system. By fixing the boundary between the communication graph and the subroutines that inhabit its vertices, the model guides the developer towards an appropriate level of granularity. The system need not try too hard to extract parallelism *within* a developer-provided subroutine, while it can exploit the fact that dependencies are all explicitly encoded in the flow graph to efficiently distribute the execution *across* those subroutines. Finally, 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.

The aforementioned systems restrict an application's communication flow for different reasons. GPU shader languages are strongly tied to an efficient underlying hardware implementation that has been tuned to give good performance for common graphics memory-access patterns. MapReduce was designed to be accessible to the widest possible class of developers, and therefore aims for simplicity at the expense of generality and performance. Parallel databases were designed for relational algebra manipulations (e.g. SQL) where the communication graph is implicit.

By contrast, the Dryad system allows the developer fine control over the communication graph as well as the subroutines that live at its vertices. A Dryad application developer can specify an arbitrary directed acyclic graph to describe the application's communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared-memory FIFOs) between the computation vertices. This direct specification of the graph also gives the developer greater flexibility to easily compose basic common operations, leading to a distributed analogue of "piping" together traditional Unix utilities such as `grep`, `sort` and `head`.

Dryad is notable for allowing graph vertices (and computations in general) to use an arbitrary number of inputs and outputs. 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.

In this paper, we demonstrate that careful choices in graph construction and refinement can substantially improve application performance, while compromising little on the programmability of the system. Nevertheless, Dryad is certainly a lower-level programming model than SQL or DirectX. In order to get the best performance from a native Dryad application, the developer must understand the structure of the computation and the organization and properties of the system resources. Dryad was however designed to be a suitable infrastructure on which to layer simpler, higher-level programming models. It has already been used, by ourselves and others, as a platform for several domain-specific systems that are briefly sketched in Section 7. These rely on Dryad to manage the complexities of distribution, scheduling, and fault-tolerance, but hide many of the details of the underlying system from the application developer. They use heuristics to automatically select and tune appropriate Dryad features, and thereby get good performance for most simple applications.

We summarize Dryad's contributions as follows:

- We built a general-purpose, high performance distributed execution engine. The Dryad execution engine handles many of the difficult problems of creating a large distributed, concurrent application: scheduling across resources, optimizing the level of concurrency within a computer, recovering from communication or computer failures, and delivering data to where it is needed. Dryad supports multiple different data transport mechanisms between computation vertices and explicit dataflow graph construction and refinement.

- We demonstrated the excellent performance of Dryad from a single multi-core computer up to clusters consisting of thousands of computers on several nontrivial, real examples. We further demonstrated that Dryad's fine control over an application's dataflow graph gives the programmer the necessary tools to optimize trade-offs between parallelism and data distribution overhead. This validated Dryad's design choices.

- We explored the programmability of Dryad on two fronts. First, we have designed a simple graph description language that empowers the developer with explicit graph construction and refinement to fully take advantage of the rich features of the Dryad execution engine. Our user experiences lead us to believe that, while it requires some effort to learn, a programmer can master the APIs required for most of the applications in a couple of weeks. Second, we (and others within Microsoft) have built simpler, higher-level programming abstractions for specific application domains on top of Dryad. This has significantly lowered the barrier to entry and increased the acceptance of Dryad among domain experts who are interested in using Dryad for rapid application prototyping. This further validated Dryad's design choices.

The next three sections describe the abstract form of a Dryad application and outline the steps involved in writing one. The Dryad scheduler is described in Section 5; it handles all of the work of deciding which physical resources to schedule work on, routing data between computations, and automatically reacting to computer and network failures. Section 6 reports on our experimental evaluation of the system, showing its flexibility and scaling characteristics in a small cluster of 10 computers, as well as details of larger-scale experiments performed on clusters with thousands of computers. We conclude in Sections 8 and 9 with a discussion of the related literature and of future research directions.

2. SYSTEM OVERVIEW

The overall structure of a Dryad job is determined by its communication flow. A job is a directed acyclic graph where each vertex is a program and edges represent data channels. It is a logical computation graph that is automatically mapped onto physical resources by the runtime. In particular, there may be many more vertices in the graph than execution cores in the computing cluster.

At run time each channel is used to transport a finite sequence of structured *items*. This channel abstraction has several concrete implementations that use shared memory, TCP pipes, or files temporarily persisted in a file system. As far as the program in each vertex is concerned, channels produce and consume heap objects that inherit from a base type. This means that a vertex program reads and writes its data in the same way regardless of whether a channel serializes its data to buffers on a disk or TCP stream, or passes object pointers directly via shared memory. The Dryad system does not include any native data model for serialization and the concrete type of an item is left entirely up to applications, which can supply their own serialization and deserialization routines. This decision allows us to support applications that operate directly on existing data including exported SQL tables and textual log files. In practice most applications use one of a small set of library item types that we supply such as newline-terminated text strings and tuples of base types.

A schematic of the Dryad system organization is shown in Figure 1. A Dryad job is coordinated by a process called the "job manager" (denoted JM in the figure) that runs either within the cluster or on a user's workstation with network access to the cluster. The job manager contains the application-specific code to construct the job's communication graph along with library code to schedule the work across the available resources. All data is sent directly between vertices and thus the job manager is only responsible for control decisions and is not a bottleneck for any data transfers.

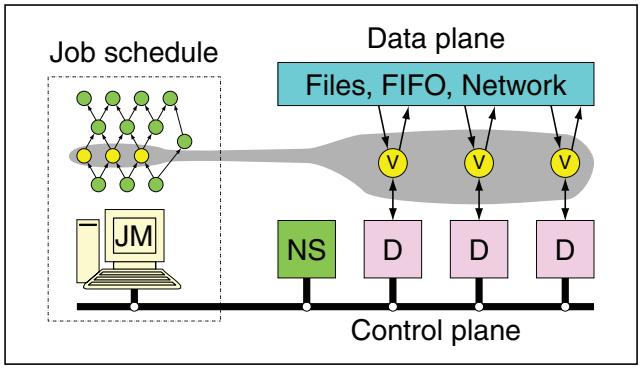

Figure 1: The Dryad system organization. The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

The cluster has a name server (NS) that can be used to enumerate all the available computers. The name server also exposes the position of each computer within the network topology so that scheduling decisions can take account of locality. There is a simple daemon (D) running on each computer in the cluster that is responsible for creating processes on behalf of the job manager. The first time a vertex (V) is executed on a computer its binary is sent from the job manager to the daemon and subsequently it is executed from a cache. The daemon acts as a proxy so that the job manager can communicate with the remote vertices and monitor the state of the computation and how much data has been read and written on its channels. It is straightforward to run a name server and a set of daemons on a user workstation to simulate a cluster and thus run an entire job locally while debugging.

A simple task scheduler is used to queue batch jobs. We use a distributed storage system, not described here, that shares with the Google File System [21] the property that large files can be broken into small pieces that are replicated and distributed across the local disks of the cluster computers. Dryad also supports the use of NTFS for accessing files directly on local computers, which can be convenient for small clusters with low management overhead.

2.1 An example SQL query

In this section, we describe a concrete example of a Dryad application that will be further developed throughout the remainder of the paper. The task we have chosen is representative of a new class of eScience applications, where scientific investigation is performed by processing large amounts of data available in digital form [24]. The database that we use is derived from the Sloan Digital Sky Survey (SDSS), available online at `http://skyserver.sdss.org`.

We chose the most time consuming query (Q18) from a published study based on this database [23]. The task is to identify a "gravitational lens" effect: it finds all the objects in the database that have neighboring objects within 30 arc seconds such that at least one of the neighbors has a color similar to the primary object's color. The query can be expressed in SQL as:

```
select distinct p.objID
from photoObjAll p
join neighbors n    -- call this join "X"
on p.objID = n.objID
  and n.objID < n.neighborObjID
  and p.mode = 1
join photoObjAll l  -- call this join "Y"
on l.objid = n.neighborObjID
  and l.mode = 1
  and abs((p.u-p.g)-(l.u-l.g))<0.05
  and abs((p.g-p.r)-(l.g-l.r))<0.05
  and abs((p.r-p.i)-(l.r-l.i))<0.05
  and abs((p.i-p.z)-(l.i-l.z))<0.05
```

There are two tables involved. The first, **photoObjAll** has 354,254,163 records, one for each identified astronomical object, keyed by a unique identifier **objID**. These records also include the object's color, as a magnitude (logarithmic brightness) in five bands: **u**, **g**, **r**, **i** and **z**. The second table, **neighbors** has 2,803,165,372 records, one for each object located within 30 arc seconds of another object. The **mode** predicates in the query select only "primary" objects. The $<$ predicate eliminates duplication caused by the **neighbors** relationship being symmetric. The output of joins "X" and "Y" are 932,820,679 and 83,798 records respectively, and the final hash emits 83,050 records.

The query uses only a few columns from the tables (the complete **photoObjAll** table contains 2 KBytes per record). When executed by SQLServer the query uses an index on **photoObjAll** keyed by **objID** with additional columns for **mode**, **u**, **g**, **r**, **i** and **z**, and an index on **neighbors** keyed by **objID** with an additional **neighborObjID** column. SQLServer reads just these indexes, leaving the remainder of the tables' data resting quietly on disk. (In our experimental setup we in fact omitted unused columns from the table, to avoid transporting the entire multi-terabyte database across

the country.) For the equivalent Dryad computation we extracted these indexes into two binary files, "**ugriz.bin**" and "**neighbors.bin**," each sorted in the same order as the indexes. The "**ugriz.bin**" file has 36-byte records, totaling 11.8 GBytes; "**neighbors.bin**" has 16-byte records, totaling 41.8 GBytes. The output of join "X" totals 31.3 GBytes, the output of join "Y" is 655 KBytes and the final output is 649 KBytes.

We mapped the query to the Dryad computation shown in Figure 2. Both data files are partitioned into n approximately equal parts (that we call U_1 through U_n and N_1 through N_n) by **objID** ranges, and we use custom C++ item objects for each data record in the graph. The vertices X_i (for $1 \le i \le n$) implement join "X" by taking their partitioned U_i and N_i inputs and merging them (keyed on **objID** and filtered by the $<$ expression and **p.mode=1**) to produce records containing **objID**, **neighborObjID**, and the color columns corresponding to **objID**. The D vertices distribute their output records to the M vertices, partitioning by **neighborObjID** using a range partitioning function four times finer than that used for the input files. The number four was chosen so that four pipelines will execute in parallel on each computer, because our computers have four processors each. The M vertices perform a non-deterministic merge of their inputs and the S vertices sort on **neighborObjID** using an in-memory Quicksort. The output records from $S_{4i-3} \ldots S_{4i}$ (for $i = 1$ through n) are fed into Y_i where they are merged with another read of U_i to implement join "Y". This join is keyed on **objID** (from U) = **neighborObjID** (from S), and is filtered by the remainder of the predicate, thus matching the colors. The outputs of the Y vertices are merged into a hash table at the H vertex to implement the **distinct** keyword in the query. Finally, an enumeration of this hash table delivers the result. Later in the paper we include more details about the implementation of this Dryad program.

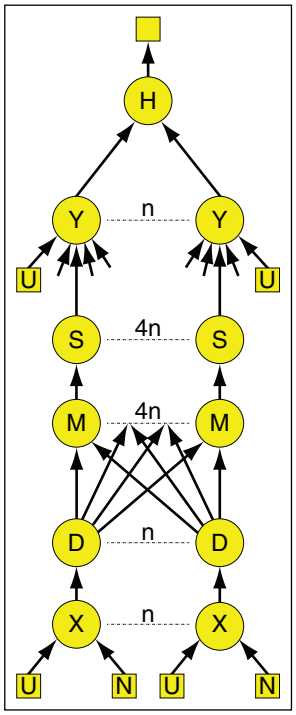

Figure 2: The communication graph for an SQL query. Details are in Section 2.1.

3. DESCRIBING A DRYAD GRAPH

We have designed a simple language that makes it easy to specify commonly-occurring communication idioms. It is currently "embedded" for convenience in C++ as a library using a mixture of method calls and operator overloading.

Graphs are constructed by combining simpler subgraphs using a small set of operations shown in Figure 3. All of the operations preserve the property that the resulting graph is acyclic. The basic object in the language is a graph:

$$G = \langle V_G, E_G, I_G, O_G \rangle.$$

G contains a sequence of vertices V_G, a set of directed edges E_G, and two sets $I_G \subseteq V_G$ and $O_G \subseteq V_G$ that "tag" some of the vertices as being *inputs* and *outputs* respectively. No graph can contain a directed edge entering an input vertex in I_G, nor one leaving an output vertex in O_G, and these tags are used below in composition operations. The input and output edges of a vertex are ordered so an edge connects specific "ports" on a pair of vertices, and a given pair of vertices may be connected by multiple edges.

3.1 Creating new vertices

The Dryad libraries define a C++ base class from which all vertex programs inherit. Each such program has a textual name (which is unique within an application) and a static "factory" that knows how to construct it. A graph vertex is created by calling the appropriate static program factory. Any required vertex-specific parameters can be set at this point by calling methods on the program object. These parameters are then marshaled along with the unique vertex name to form a simple closure that can be sent to a remote process for execution.

A singleton graph is generated from a vertex v as $G = \langle (v), \emptyset, \{v\}, \{v\} \rangle$. A graph can be cloned into a new graph containing k copies of its structure using the `^` operator where `C = G^k` is defined as:

$$C = \langle V_G^1 \oplus \cdots \oplus V_G^k, E_G^1 \cup \cdots \cup E_G^k, \\ I_G^1 \cup \cdots \cup I_G^k, O_G^1 \cup \cdots \cup O_G^k \rangle.$$

Here $G^n = \langle V_G^n, E_G^n, I_G^n, O_G^n \rangle$ is a "clone" of G containing copies of all of G's vertices and edges, $\oplus$ denotes sequence concatenation, and each cloned vertex inherits the type and parameters of its corresponding vertex in G.

3.2 Adding graph edges

New edges are created by applying a composition operation to two existing graphs. There is a family of compositions all sharing the same basic structure: `C = A ∘ B` creates a new graph:

$$C = \langle V_A \oplus V_B, E_A \cup E_B \cup E_{\text{new}}, I_A, O_B \rangle$$

where C contains the union of all the vertices and edges in A and B, with A's inputs and B's outputs. In addition, directed edges E_{new} are introduced between vertices in O_A and I_B. V_A and V_B are enforced to be disjoint at run time, and since A and B are both acyclic, C is also.

Compositions differ in the set of edges E_{new} that they add into the graph. We define two standard compositions:

- `A >= B` forms a pointwise composition as shown in Figure 3(c). If $|O_A| \ge |I_B|$ then a single outgoing edge is created from each of A's outputs. The edges are assigned in round-robin to B's inputs. Some of the vertices in I_B may end up with more than one incoming edge. If $|I_B| > |O_A|$, a single incoming edge is created to each of B's inputs, assigned in round-robin from A's outputs.

- `A >> B` forms the complete bipartite graph between O_A and I_B and is shown in Figure 3(d).

We allow the user to extend the language by implementing new composition operations.

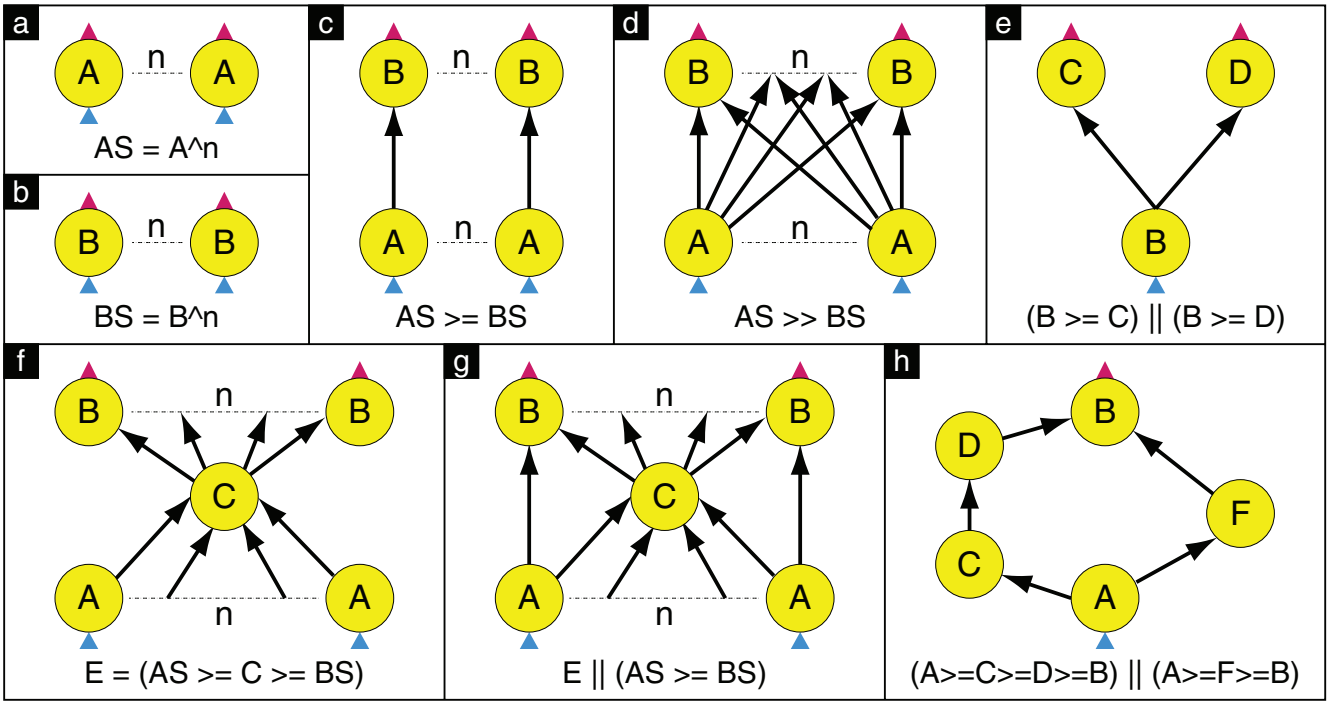

Figure 3: The operators of the graph description language. Circles are vertices and arrows are graph edges. A triangle at the bottom of a vertex indicates an *input* and one at the top indicates an *output*. Boxes (a) and (b) demonstrate cloning individual vertices using the ^ operator. The two standard connection operations are pointwise composition using >= shown in (c) and complete bipartite composition using >> shown in (d). (e) illustrates a merge using `||`. The second line of the figure shows more complex patterns. The merge in (g) makes use of a "subroutine" from (f) and demonstrates a bypass operation. For example, each A vertex might output a summary of its input to C which aggregates them and forwards the global statistics to every B. Together the B vertices can then distribute the original dataset (received from A) into balanced partitions. An asymmetric fork/join is shown in (h).

3.3 Merging two graphs

The final operation in the language is `||`, which merges two graphs. `C = A || B` creates a new graph:

$$C = \langle V_A \oplus^* V_B, E_A \cup E_B, I_A \cup^* I_B, O_A \cup^* O_B \rangle$$

where, in contrast to the composition operations, it is not required that A and B be disjoint. $V_A \oplus^* V_B$ is the concatenation of V_A and V_B with duplicates removed from the second sequence. $I_A \cup^* I_B$ means the union of A and B's inputs, minus any vertex that has an incoming edge following the merge (and similarly for the output case). If a vertex is contained in $V_A \cap V_B$ its input and output edges are concatenated so that the edges in E_A occur first (with lower port numbers). This simplification forbids certain graphs with "crossover" edges, however we have not found this restriction to be a problem in practice. The invariant that the merged graph be acyclic is enforced by a run-time check.

The merge operation is extremely powerful and makes it easy to construct typical patterns of communication such as fork/join and bypass as shown in Figures 3(f)–(h). It also provides the mechanism for assembling a graph "by hand" from a collection of vertices and edges. So for example, a tree with four vertices a, b, c, and d might be constructed as `G = (a>=b) || (b>=c) || (b>=d)`.

The graph builder program to construct the query graph in Figure 2 is shown in Figure 4.

3.4 Channel types

By default each channel is implemented using a temporary file: the producer writes to disk (typically on its local computer) and the consumer reads from that file.

In many cases multiple vertices will fit within the resources of a single computer so it makes sense to execute them all within the same process. The graph language has an "encapsulation" command that takes a graph G and returns a new vertex v_G. When v_G is run as a vertex program, the job manager passes it a serialization of G as an invocation parameter, and it runs all the vertices of G simultaneously within the same process, connected by edges implemented using shared-memory FIFOs. While it would always be possible to write a custom vertex program with the same semantics as G, allowing encapsulation makes it efficient to combine simple library vertices at the graph layer rather than re-implementing their functionality as a new vertex program.

Sometimes it is desirable to place two vertices in the same process even though they cannot be collapsed into a single graph vertex from the perspective of the scheduler. For example, in Figure 2 the performance can be improved by placing the first D vertex in the same process as the first four M and S vertices and thus avoiding some disk I/O, however the S vertices cannot be started until all of the D vertices complete.

When creating a set of graph edges, the user can optionally specify the transport protocol to be used. The available protocols are listed in Table 1. Vertices that are connected using shared-memory channels are executed within a single process, though they are individually started as their inputs become available and individually report completion.

Because the dataflow graph is acyclic, scheduling deadlock is impossible when all channels are either written to temporary files or use shared-memory FIFOs hidden within

```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;

GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;

GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i)
{
    XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}

GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;

GraphBuilder final = XInputs || YInputs || XToY || YToH || HOutputs;
```

Figure 4: An example graph builder program. The communication graph generated by this program is shown in Figure 2.

Channel protocol	Discussion
File (the default)	Preserved after vertex execution until the job completes.
TCP pipe	Requires no disk accesses, but both end-point vertices must be scheduled to run at the same time.
Shared-memory FIFO	Extremely low communication cost, but end-point vertices must run within the same process.

Table 1: Channel types.

encapsulated acyclic subgraphs. However, allowing the developer to use pipes and "visible" FIFOs can cause deadlocks. Any connected component of vertices communicating using pipes or FIFOs must all be scheduled in processes that are concurrently executing, but this becomes impossible if the system runs out of available computers in the cluster.

This breaks the abstraction that the user need not know the physical resources of the system when writing the application. We believe that it is a worthwhile trade-off, since, as reported in our experiments in Section 6, the resulting performance gains can be substantial. Note also that the system could always avoid deadlock by "downgrading" a pipe channel to a temporary file, at the expense of introducing an unexpected performance cliff.

3.5 Job inputs and outputs

Large input files are typically partitioned and distributed across the computers of the cluster. It is therefore natural to group a logical input into a graph $G = \langle V_P, \emptyset, \emptyset, V_P \rangle$ where V_P is a sequence of "virtual" vertices corresponding to the partitions of the input. Similarly on job completion a set of output partitions can be logically concatenated to form a single named distributed file. An application will generally interrogate its input graphs to read the number of partitions at run time and automatically generate the appropriately replicated graph.

3.6 Job Stages

When the graph is constructed every vertex is placed in a "stage" to simplify job management. The stage topology can be seen as a "skeleton" or summary of the overall job, and the stage topology of our example Skyserver query application is shown in Figure 5. Each distinct type of vertex is grouped into a separate stage. Most stages are connected using the >= operator, while D is connected to M using the >> operator. The skeleton is used as a guide for generating summaries when monitoring a job, and can also be exploited by the automatic optimizations described in Section 5.2.

4. WRITING A VERTEX PROGRAM

The primary APIs for writing a Dryad vertex program are exposed through C++ base classes and objects. It was a design requirement for Dryad vertices to be able to incorporate legacy source and libraries, so we deliberately avoided adopting any Dryad-specific language or sandboxing restrictions. Most of the existing code that we anticipate integrating into vertices is written in C++, but it is straightforward to implement API wrappers so that developers can write vertices in other languages, for example C#. There is also significant value for some domains in being able to run unmodified legacy executables in vertices, and so we support this as explained in Section 4.2 below.

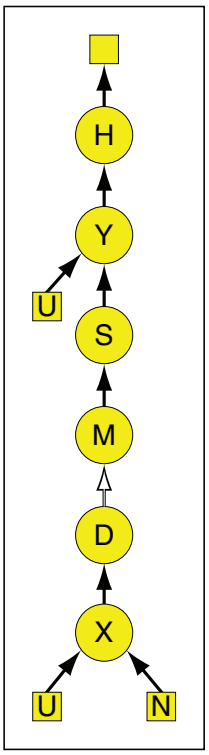

Figure 5: The stages of the Dryad computation from Figure 2. Section 3.6 has details.

4.1 Vertex execution

Dryad includes a runtime library that is responsible for setting up and executing vertices as part of a distributed computation. As outlined in Section 3.1 the runtime receives a closure from the job manager describing the vertex to be run, and URIs describing the input and output channels to connect to it. There is currently no type-checking for channels and the vertex must be able to determine, either statically or from the invocation parameters, the types of the items that it is expected to read and write

on each channel in order to supply the correct serialization routines. The body of a vertex is invoked via a standard `Main` method that includes channel readers and writers in its argument list. The readers and writers have a blocking interface to read or write the next item, which suffices for most simple applications. The vertex can report status and errors to the job manager, and the progress of channels is automatically monitored.

Many developers find it convenient to inherit from predefined vertex classes that hide the details of the underlying channels and vertices. We supply `map` and `reduce` classes with similar interfaces to those described in [16]. We have also written a variety of others including a general-purpose `distribute` that takes a single input stream and writes on multiple outputs, and `join`s that call a virtual method with every matching record tuple. These "classes" are simply vertices like any other, so it is straightforward to write new ones to support developers working in a particular domain.

4.2 Legacy executables

We provide a library "process wrapper" vertex that forks an executable supplied as an invocation parameter. The wrapper vertex must work with arbitrary data types, so its "items" are simply fixed-size buffers that are passed unmodified to the forked process using named pipes in the filesystem. This allows unmodified pre-existing binaries to be run as Dryad vertex programs. It is easy, for example, to invoke `perl` scripts or `grep` at some vertices of a Dryad job.

4.3 Efficient pipelined execution

Most Dryad vertices contain purely sequential code. We also support an event-based programming style, using a shared thread pool. The program and channel interfaces have asynchronous forms, though unsurprisingly it is harder to use the asynchronous interfaces than it is to write sequential code using the synchronous interfaces. In some cases it may be worth investing this effort, and many of the standard Dryad vertex classes, including non-deterministic merge, sort, and generic maps and joins, are built using the event-based programming style. The runtime automatically distinguishes between vertices which can use a thread pool and those that require a dedicated thread, and therefore encapsulated graphs which contain hundreds of asynchronous vertices are executed efficiently on a shared thread pool.

The channel implementation schedules read, write, serialization and deserialization tasks on a thread pool shared between all channels in a process, and a vertex can concurrently read or write on hundreds of channels. The runtime tries to ensure efficient pipelined execution while still presenting the developer with the simple abstraction of reading and writing a single record at a time. Extensive use is made of batching [28] to try to ensure that threads process hundreds or thousands of records at a time without touching a reference count or accessing a shared queue. The experiments in Section 6.2 substantiate our claims for the efficiency of these abstractions: even single-node Dryad applications have throughput comparable to that of a commercial database system.

5. JOB EXECUTION

The scheduler inside the job manager keeps track of the state and history of each vertex in the graph. At present if the job manager's computer fails the job is terminated, though 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 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. Each execution names its file-based output channels uniquely using its version number to avoid conflicts among versions. If the entire job completes successfully then each vertex selects a successful execution and renames its output files to their correct final forms.

When all of a vertex's input channels become ready a new execution record is created for the vertex and placed in a scheduling queue. A disk-based channel is considered to be ready when the entire file is present. A channel that is a TCP pipe or shared-memory FIFO is ready when the predecessor vertex has at least one running execution record.

A vertex and any of its channels may each specify a "hard-constraint" or a "preference" listing the set of computers on which it would like to run. The constraints are combined and attached to the execution record when it is added to the scheduling queue and they allow the application writer to require that a vertex be co-located with a large input file, and in general let the scheduler preferentially run computations close to their data.

At present the job manager performs greedy scheduling based on the assumption that it is the only job running on the cluster. When an execution record is paired with an available computer the remote daemon is instructed to run the specified vertex, and during execution the job manager receives periodic status updates from the vertex. If every vertex eventually completes then the job is deemed to have completed successfully. If any vertex is re-run more than a set number of times then the entire job is failed.

Files representing temporary channels are stored in directories managed by the daemon and cleaned up after the job completes, and vertices are killed by the daemon if their "parent" job manager crashes. We have a simple graph visualizer suitable for small jobs that shows the state of each vertex and the amount of data transmitted along each channel as the computation progresses. A web-based interface shows regularly-updated summary statistics of a running job and can be used to monitor large computations. The statistics include the number of vertices that have completed or been re-executed, the amount of data transferred across channels, and the error codes reported by failed vertices. Links are provided from the summary page that allow a developer to download logs or crash dumps for further debugging, along with a script that allows the vertex to be re-executed in isolation on a local machine.

5.1 Fault tolerance policy

Failures are to be expected during the execution of any distributed application. Our default failure policy is suitable for the common case that all vertex programs are deterministic.[1] Because our communication graph is acyclic, it is relatively straightforward to ensure that every terminating execution of a job with immutable inputs will compute the

[1]The definition of job completion and the treatment of job outputs above also implicitly assume deterministic execution.

same result, regardless of the sequence of computer or disk failures over the course of the execution.

When a vertex execution fails for any reason the job manager is informed. If the vertex reported an error cleanly the process forwards it via the daemon before exiting; if the process crashes the daemon notifies the job manager; and if the daemon fails for any reason the job manager receives a heartbeat timeout. If the failure was due to a read error on an input channel (which is be reported cleanly) the default policy also marks the execution record that generated that version of the channel as failed and terminates its process if it is running. This will cause the vertex that created the failed input channel to be re-executed, and will lead in the end to the offending channel being re-created. Though a newly-failed execution record may have non-failed successor records, errors need not be propagated forwards: since vertices are deterministic two successors may safely compute using the outputs of different execution versions. Note however that under this policy an entire connected component of vertices connected by pipes or shared-memory FIFOs will fail as a unit since killing a running vertex will cause it to close its pipes, propagating errors in both directions along those edges. Any vertex whose execution record is set to failed is immediately considered for re-execution.

As Section 3.6 explains, each vertex belongs to a "stage," and each stage has a manager object that receives a callback on every state transition of a vertex execution in that stage, and on a regular timer interrupt. Within this callback the stage manager holds a global lock on the job manager data-structures and can therefore implement quite sophisticated behaviors. For example, the default stage manager includes heuristics to detect vertices that are running slower than their peers and schedule duplicate executions. This prevents a single slow computer from delaying an entire job and is similar to the backup task mechanism reported in [16]. In future we may allow non-deterministic vertices, which would make fault-tolerance more interesting, and so we have implemented our policy via an extensible mechanism that allows non-standard applications to customize their behavior.

5.2 Run-time graph refinement

We have used the stage-manager callback mechanism to implement run-time optimization policies that allow us to scale to very large input sets while conserving scarce network bandwidth. Some of the large clusters we have access to have their network provisioned in a two-level hierarchy, with a dedicated mini-switch serving the computers in each rack, and the per-rack switches connected via a single large core switch. Therefore where possible it is valuable to schedule vertices as much as possible to execute on the same computer or within the same rack as their input data.

If a computation is associative and commutative, and performs a data reduction, then it can benefit from an aggregation tree. As shown in Figure 6, a logical graph connecting a set of inputs to a single downstream vertex can be refined by inserting a new layer of internal vertices, where each internal vertex reads data from a subset of the inputs that are close in network topology, for example on the same computer or within the same rack. If the internal vertices perform a data reduction, the overall network traffic between racks will be reduced by this refinement. A typical application would be a histogramming operation that takes as input a set of partial histograms and outputs their union. The implementation in

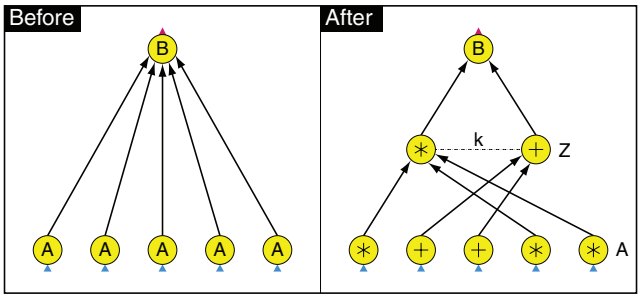

Figure 6: A dynamic refinement for aggregation. The logical graph on the left connects every input to the single output. The locations and sizes of the inputs are not known until run time when it is determined which computer each vertex is scheduled on. At this point the inputs are grouped into subsets 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.

Dryad simply attaches a custom stage manager to the input layer. As this aggregation manager receives callback notifications that upstream vertices have completed, it rewrites the graph with the appropriate refinements.

The operation in Figure 6 can be performed recursively to generate as many layers of internal vertices as required. We have also found a "partial aggregation" operation to be very useful. This refinement is shown in Figure 7; having grouped the inputs into k sets, the optimizer replicates the downstream vertex k times to allow all of the sets to be processed in parallel. Optionally, the partial refinement can be made to propagate through the graph so that an entire pipeline of vertices will be replicated k times (this behavior is not shown in the figure). An example of the application of this technique is described in the experiments in Section 6.3. Since the aggregation manager is notified on the completion of upstream vertices, it has access to the size of the data written by those vertices as well as its location. A typical grouping heuristic ensures that a downstream vertex has no more than a set number of input channels, or a set volume of input data. A special case of partial refinement can be performed at startup to size the initial layer of a graph so that, for example, each vertex processes multiple inputs up to some threshold with the restriction that all the inputs must lie on the same computer.

Because input data can be replicated on multiple computers in a cluster, the computer on which a graph vertex is scheduled is in general non-deterministic. Moreover the

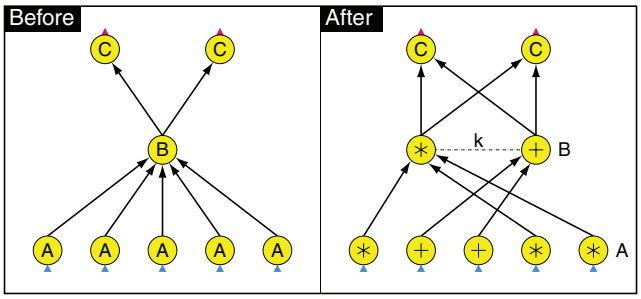

Figure 7: A partial aggregation refinement. Following an input grouping as in Figure 6 into k sets, the successor vertex is replicated k times to process all the sets in parallel.

amount of data written in intermediate computation stages is typically not known before a computation begins. Therefore dynamic refinement is often more efficient than attempting a static grouping in advance.

Dynamic refinements of this sort emphasize the power of overlaying a physical graph with its "skeleton." For many applications, there is an equivalence class of graphs with the same skeleton that compute the same result. Varying the number of vertices in each stage, or their connectivity, while preserving the graph topology at the stage level, is merely a (dynamic) performance optimization.

6. EXPERIMENTAL EVALUATION

Dryad has been used for a wide variety of applications, including relational queries, large-scale matrix computations, and many text-processing tasks. For this paper we examined the effectiveness of the Dryad system in detail by running two sets of experiments. The first experiment takes the SQL query described in Section 2.1 and implements it as a Dryad application. We compare the Dryad performance with that of a traditional commercial SQL server, and we analyze the Dryad performance as the job is distributed across different numbers of computers. The second is a simple map-reduce style data-mining operation, expressed as a Dryad program and applied to 10.2 TBytes of data using a cluster of around 1800 computers.

The strategies we adopt to build our communication flow graphs are familiar from the parallel database literature [18] and include horizontally partitioning the datasets, exploiting pipelined parallelism within processes and applying exchange operations to communicate partial results between the partitions. None of the application-level code in any of our experiments makes explicit use of concurrency primitives.

6.1 Hardware

The SQL query experiments were run on a cluster of 10 computers in our own laboratory, and the data-mining tests were run on a cluster of around 1800 computers embedded in a data center. Our laboratory computers each had 2 dual-core Opteron processors running at 2 GHz (i.e., 4 CPUs total), 8 GBytes of DRAM (half attached to each processor chip), and 4 disks. The disks were 400 GByte Western Digital WD40 00YR-01PLB0 SATA drives, connected through a Silicon Image 3114 PCI SATA controller (66MHz, 32-bit). Network connectivity was by 1 Gbit/sec Ethernet links connecting into a single non-blocking switch. One of our laboratory computers was dedicated to running SQLServer and its data was stored in 4 separate 350 GByte NTFS volumes, one on each drive, with SQLServer configured to do its own data striping for the raw data and for its temporary tables. All the other laboratory computers were configured with a single 1.4 TByte NTFS volume on each computer, created by software striping across the 4 drives. The computers in the data center had a variety of configurations, but were typically roughly comparable to our laboratory equipment. All the computers were running Windows Server 2003 Enterprise x64 edition SP1.

6.2 SQL Query

The query for this experiment is described in Section 2.1 and uses the Dryad communication graph shown in Figure 2. SQLServer 2005's execution plan for this query was very close to the Dryad computation, except that it used an external hash join for "Y" in place of the sort-merge we chose for Dryad. SQLServer takes slightly longer if it is forced by a query hint to use a sort-merge join.

For our experiments, we used two variants of the Dryad graph: "in-memory" and "two-pass." In both variants communication from each M_i through its corresponding S_i to Y is by a shared-memory FIFO. This pulls four sorters into the same process to execute in parallel on the four CPUs in each computer. In the "in-memory" variant only, communication from each D_i to its four corresponding M_j vertices is also by a shared-memory FIFO and the rest of the $D_i \rightarrow M_k$ edges use TCP pipes. All other communication is through NTFS temporary files in both variants.

There is good spatial locality in the query, which improves as the number of partitions (n) decreases: for $n = 40$ an average of 80% of the output of D_i goes to its corresponding M_i, increasing to 88% for $n = 6$. In either variant n must be large enough that every sort executed by a vertex S_i will fit into the computer's 8 GBytes of DRAM (or else it will page). With the current data, this threshold is at $n = 6$.

Note that the non-deterministic merge in M randomly permutes its output depending on the order of arrival of items on its input channels and this technically violates the requirement that all vertices be deterministic. This does not cause problems for our fault-tolerance model because the sort S_i "undoes" this permutation, and since the edge from M_i to S_i is a shared-memory FIFO within a single process the two vertices fail (if at all) in tandem and the non-determinism never "escapes."

The in-memory variant requires at least n computers since otherwise the S vertices will deadlock waiting for data from an X vertex. The two-pass variant will run on any number of computers. One way to view this trade-off is that by adding the file buffering in the two-pass variant we in effect converted to using a two-pass external sort. Note that the conversion from the in-memory to the two-pass program simply involves changing two lines in the graph construction code, with no modifications to the vertex programs.

We ran the two-pass variant using $n = 40$, varying the number of computers from 1 to 9. We ran the in-memory variant using $n = 6$ through $n = 9$, each time on n computers. As a baseline measurement we ran the query on a reasonably well optimized SQLServer on one computer. Table 2 shows the elapsed times in seconds for each experiment. On repeated runs the times were consistent to within 3.4% of their averages except for the single-computer two-pass case, which was within 9.4%. Figure 8 graphs the inverse of these times, normalized to show the speed-up factor relative to the two-pass single-computer case.

The results are pleasantly straightforward. The two-pass Dryad job works on all cluster sizes, with close to linear speed-up. The in-memory variant works as expected for

Computers	1	2	3	4	5	6	7	8	9
SQLServer	3780								
Two-pass	2370	1260	836	662	523	463	423	346	321
In-memory						217	203	183	168

Table 2: Time in seconds to process an SQL query using different numbers of computers. The SQLServer implementation cannot be distributed across multiple computers and the in-memory experiment can only be run for 6 or more computers.

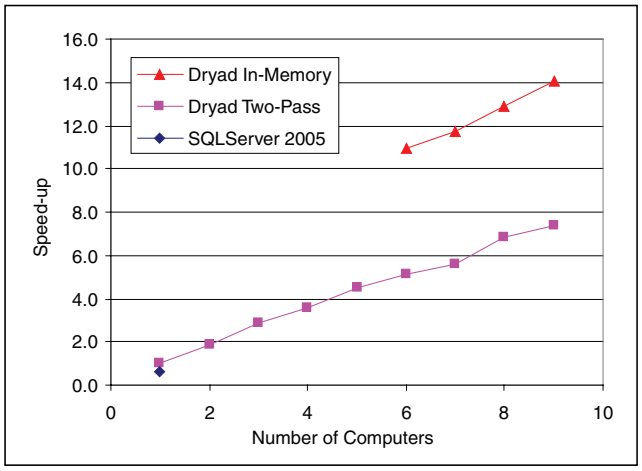

Figure 8: The speedup of the SQL query computation is near-linear in the number of computers used. The baseline is relative to Dryad running on a single computer and times are given in Table 2.

$n = 6$ and up, again with close to linear speed-up, and approximately twice as fast as the two-pass variant. The SQLServer result matches our expectations: our specialized Dryad program runs significantly, but not outrageously, faster than SQLServer's general-purpose query engine. We should note of course that Dryad simply provides an execution engine while the database provides much more functionality, including logging, transactions, and mutable relations.

6.3 Data mining

The data-mining experiment fits the pattern of map then reduce. 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 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 communication graph is shown in Figure 9. The log files are partitioned and replicated across the computers' disks. The P vertices each read their part of the log files using library newline-delimited text items, and parse them to extract the query strings. Subsequent items are all library tuples containing a query string, a count, and a hash of the string. Each D vertex distributes to k outputs based on the query string hash; S performs an in-memory sort. C accumulates total counts for each query and MS performs a streaming merge-sort. S and MS come from a vertex library and take a comparison function as a parameter; in this example they sort based on the query hash. We have encapsulated the simple vertices into subgraphs denoted by diamonds in order to reduce the total number of vertices in the job (and hence the overhead associated with process start-up) and the volume of temporary data written to disk.

The graph shown in Figure 9 does not scale well to very large datasets. It is wasteful to execute a separate Q vertex for every input partition. Each partition is only around 100 MBytes, and the P vertex performs a substantial data reduction, so the amount of data which needs to be sorted by the S vertices is very much less than the total RAM on a computer. Also, each R subgraph has n inputs, and when n grows to hundreds of thousands of partitions, it becomes unwieldy to read in parallel from so many channels.

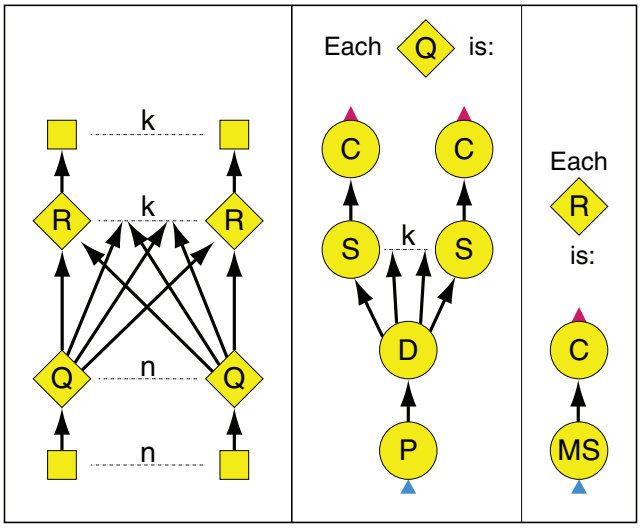

Figure 9: The communication graph to compute a query histogram. Details are in Section 6.3. This figure shows the first cut "naive" encapsulated version that doesn't scale well.

After trying a number of different encapsulation and dynamic refinement schemes we arrived at the communication graphs shown in Figure 10 for our experiment. Each subgraph in the first phase now has multiple inputs, grouped automatically using the refinement in Figure 7 to ensure they all lie on the same computer. The inputs are sent to the parser P through a non-deterministic merge vertex M. The distribution (vertex D) has been taken out of the first phase to allow another layer of grouping and aggregation (again using the refinement in Figure 7) before the explosion in the number of output channels.

We ran this experiment on 10,160,519,065,748 Bytes of input data in a cluster of around 1800 computers embedded in a data center. The input was divided into 99,713 partitions replicated across the computers, and we specified that the application should use 450 R subgraphs. The first phase grouped the inputs into at most 1 GBytes at a time, all lying on the same computer, resulting in 10,405 Q' subgraphs that wrote a total of 153,703,445,725 Bytes. The outputs from the Q' subgraphs were then grouped into sets of at most 600 MBytes on the same local switch resulting in 217 T subgraphs. Each T was connected to every R subgraph, and they wrote a total of 118,364,131,628 Bytes. The total output from the R subgraphs was 33,375,616,713 Bytes, and the end-to-end computation took 11 minutes and 30 seconds. Though this experiment only uses 11,072 vertices, intermediate experiments with other graph topologies confirmed that Dryad can successfully execute jobs containing hundreds of thousands of vertices.

We would like to emphasize several points about the optimization process we used to arrive at the graphs in Figure 10:

1. At no point during the optimization did we have to modify any of the code running inside the vertices: we were simply manipulating the graph of the job's communication flow, changing tens of lines of code.

2. This communication graph is well suited to any map-reduce computation with similar characteristics: i.e. that the map phase (our P vertex) performs substan-

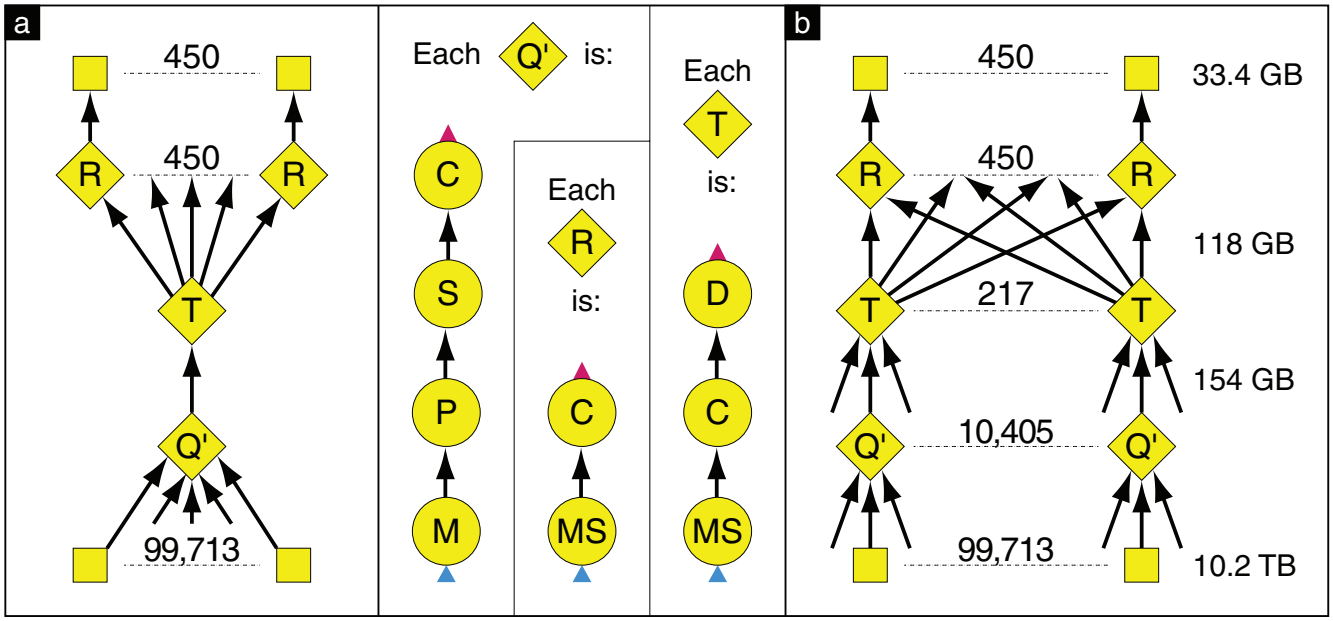

Figure 10: Rearranging the vertices gives better scaling performance compared with Figure 9. The user supplies graph (a) specifying that 450 buckets should be used when distributing the output, and that each Q' vertex may receive up to 1GB of input while each T may receive up to 600MB. The number of Q' and T vertices is determined at run time based on the number of partitions in the input and the network locations and output sizes of preceding vertices in the graph, and the refined graph (b) is executed by the system. Details are in Section 6.3.

tial data reduction and the reduce phase (our C vertex) performs some additional relatively minor data reduction. A different topology might give better performance for a map-reduce task with different behavior; for example if the reduce phase performed substantial data reduction a dynamic merge tree as described in Figure 6 might be more suitable.

3. When scaling up another order of magnitude or two, we might change the topology again, e.g. by adding more layers of aggregation between the T and R stages. Such re-factoring is easy to do.

4. Getting good performance for large-scale data-mining computations is not trivial. Many novel features of the Dryad system, including subgraph encapsulation and dynamic refinement, were used. These made it simple to experiment with different optimization schemes that would have been difficult or impossible to implement using a simpler but less powerful system.

7. BUILDING ON DRYAD

As explained in the introduction, we have targeted Dryad at developers who are experienced at using high-level compiled programming languages. In some domains there may be great value in making common large-scale data processing tasks easier to perform, since this allows non-developers to directly query the data store [33]. We designed Dryad to be usable as a platform on which to develop such more restricted but simpler programming interfaces, and two other groups within Microsoft have already prototyped systems to address particular application domains.

7.1 The "Nebula" scripting language

One team has layered a scripting interface on top of Dryad. It allows a user to specify a computation as a series of stages (corresponding to the Dryad stages described in Section 3.6), each taking inputs from one or more previous stages or the file system. Nebula transforms Dryad into a generalization of the Unix piping mechanism and it allows programmers to write giant acyclic graphs spanning many computers. Often a Nebula script only refers to existing executables such as `perl` or `grep`, allowing a user to write an entire complex distributed application without compiling any code. The Nebula layer on top of Dryad, together with some `perl` wrapper functions, has proved to be very successful for large-scale text processing, with a low barrier to entry for users. Scripts typically run on thousands of computers and contain 5–15 stages including multiple projections, aggregations and joins, often combining the information from multiple input sets in sophisticated ways.

Nebula hides most of the details of the Dryad program from the developer. Stages are connected to preceding stages using operators that implicitly determine the number of vertices required. For example, a "Filter" operation creates one new vertex for every vertex in its input list, and connects them pointwise to form a pipeline. An "Aggregate" operation can be used to perform exchanges and merges. The implementation of the Nebula operators makes use of dynamic optimizations like those described in Section 5.2 however the operator abstraction allows users to remain unaware of the details of these optimizations. All Nebula vertices execute the process wrapper described in Section 4.2, and the vertices in a given stage all run the same executable and command-line, specified using the script. The Nebula system defines conventions for passing the names of the input and output pipes to the vertex executable command-line.

There is a very popular "front-end" to Nebula that lets the user describe a job using a combination of: fragments of `perl` that parse lines of text from different sources into structured records; and a relational query over those structured records expressed in a subset of SQL that includes `select`, `project` and `join`. This job description is converted into a Nebula script and executed using Dryad. The `perl` pars-

ing fragments for common input sources are all in libraries, so many jobs using this front-end are completely described using a few lines of SQL.

7.2 Integration with SSIS

SQL Server Integration Services (SSIS) [6] supports workflow-based application programming on a single instance of SQLServer. The AdCenter team in MSN has developed a system that embeds local SSIS computations in a larger, distributed graph with communication, scheduling and fault tolerance provided by Dryad. The SSIS input graph can be built and tested on a single computer using the full range of SQL developer tools. These include a graphical editor for constructing the job topology, and an integrated debugger.

When the graph is ready to run on a larger cluster the system automatically partitions it using heuristics and builds a Dryad graph that is then executed in a distributed fashion. Each Dryad vertex is an instance of SQLServer running an SSIS subgraph of the complete job. This system is currently deployed in a live production system as part of one of AdCenter's log processing pipelines.

7.3 Distributed SQL queries

One obvious additional direction would be to adapt a query optimizer for SQL or LINQ [4] queries to compile plans directly into a Dryad flow graph using appropriate parameterized vertices for the relational operations. Since our fault-tolerance model only requires that inputs be immutable over the duration of the query, any underlying storage system that offers lightweight snapshots would suffice to allow us to deliver consistent query results. We intend to pursue this as future work.

8. RELATED WORK

Dryad is related to a broad class of prior literature, ranging from custom hardware to parallel databases, but we believe that the ensemble of trade-offs we have chosen for its design, and some of the technologies we have deployed, make it a unique system.

Hardware Several hardware systems use stream programming models similar to Dryad, including Intel IXP [2], Imagine [26], and SCORE [15]. Programmers or compilers represent the distributed computation as a collection of independent subroutines residing within a high-level graph.

Click A similar approach is adopted by the Click modular router [27]. The technique used to encapsulate multiple Dryad vertices in a single large vertex, described in section 3.4, is similar to the method used by Click to group the elements (equivalent of Dryad vertices) in a single process. However, Click is always single-threaded, while Dryad encapsulated vertices are designed to take advantage of multiple CPU cores that may be available.

Dataflow The overall structure of a Dryad application is closely related to large-grain dataflow techniques used in e.g. LGDF2 [19], CODE2 [31] and P-RIO [29]. These systems were not designed to scale to large clusters of commodity computers, however, and do not tolerate machine failures or easily support programming very large graphs. Paralex [9] has many similarities to Dryad, but in order to provide automatic fault-tolerance sacrifices the vertex programming model, allowing only pure-functional programs.

Parallel databases Dryad is heavily indebted to the traditional parallel database field [18]: e.g., Vulcan [22], Gamma [17], RDb [11], DB2 parallel edition [12], and many others. Many techniques for exploiting parallelism, including data partitioning; pipelined and partitioned parallelism; and hash-based distribution are directly derived from this work. We can map the whole relational algebra on top of Dryad, however Dryad is not a database engine: it does not include a query planner or optimizer; the system has no concept of data schemas or indices; and Dryad does not support transactions or logs. Dryad gives the programmer more control than SQL via C++ programs in vertices and allows programmers to specify encapsulation, transport mechanisms for edges, and callbacks for vertex stages. Moreover, the graph builder language allows Dryad to express irregular computations.

Continuous Query systems There are some superficial similarities between CQ systems (e.g. [25, 10, 34]) and Dryad, such as some operators and the topologies of the computation networks. However, Dryad is a batch computation system, not designed to support real-time operation which is crucial for CQ systems since many CQ window operators depend on real-time behavior. Moreover, many datamining Dryad computations require extremely high throughput (tens of millions of records per second per node), which is much greater than that typically seen in the CQ literature.

Explicitly parallel languages like Parallel Haskell [38], Cilk [14] or NESL [13] have the same emphasis as Dryad on using the user's knowledge of the problem to drive the parallelization. By relying on C++, Dryad should have a faster learning curve than that for functional languages, while also being able to leverage commercial optimizing compilers. There is some appeal in these alternative approaches, which present the user with a uniform programming abstraction rather than our two-level hierarchy. However, we believe that for data-parallel applications that are naturally written using coarse-grain communication patterns, we gain substantial benefit by letting the programmer cooperate with the system to decide on the granularity of distribution.

Grid computing [1] and projects such as Condor [37] are clearly related to Dryad, in that they leverage the resources of many workstations using batch processing. However, Dryad does not attempt to provide support for wide-area operation, transparent remote I/O, or multiple administrative domains. Dryad is optimized for the case of a very high-throughput LAN, whereas in Condor bandwidth management is essentially handled by the user job.

Google MapReduce The Dryad system was primarily designed to support large-scale data-mining over clusters of thousands of computers. As a result, of the recent related systems it shares the most similarities with Google's MapReduce [16, 33] which addresses a similar

problem domain. The fundamental difference between the two systems is that a Dryad application may specify an arbitrary communication DAG rather than requiring a sequence of map/distribute/sort/reduce operations. In particular, graph vertices may consume multiple inputs, and generate multiple outputs, of different types. For many applications this simplifies the mapping from algorithm to implementation, lets us build on a greater library of basic subroutines, and, together with the ability to exploit TCP pipes and shared-memory for data edges, can bring substantial performance gains. At the same time, our implementation is general enough to support all the features described in the MapReduce paper.

Scientific computing Dryad is also related to high-performance computing platforms like MPI [5], PVM [35], or computing on GPUs [36]. However, Dryad focuses on a model with no shared-memory between vertices, and uses no synchronization primitives.

NOW The original impetus for employing clusters of workstations with a shared-nothing memory model came from projects like Berkeley NOW [7, 8], or TACC [20]. Dryad borrows some ideas from these systems, such as fault-tolerance through re-execution and centralized resource scheduling, but our system additionally provides a unified, simple high-level programming language layer.

Log datamining Addamark, now renamed SenSage [32] has successfully commercialized software for log datamining on clusters of workstations. Dryad is designed to scale to much larger implementations, up to thousands of computers.

9. DISCUSSION

With the basic Dryad infrastructure in place, we see a number of interesting future research directions. One fundamental question is the applicability of the programming model to general large-scale computations beyond text processing and relational queries. Of course, not all programs are easily expressed using a coarse-grain data-parallel communication graph, but we are now well positioned to identify and evaluate Dryad's suitability for those that are.

Section 3 assumes an application developer will first construct a static job graph, then pass it to the runtime to be executed. Section 6.3 shows the benefits of allowing applications to perform automatic dynamic refinement of the graph. We plan to extend this idea and also introduce interfaces to simplify dynamic modifications of the graph according to application-level control flow decisions. We are particularly interested in data-dependent optimizations that might pick entirely different strategies (for example choosing between in-memory and external sorts) as the job progresses and the volume of data at intermediate stages becomes known. Many of these strategies are already described in the parallel database literature, but Dryad gives us a flexible testbed for exploring them at very large scale. At the same time, we must ensure that any new optimizations can be targeted by higher-level languages on top of Dryad, and we plan to implement comprehensive support for relational queries as suggested in Section 7.3.

The job manager described in this paper assumes it has exclusive control of all of the computers in the cluster, and this makes it difficult to efficiently run more than one job at a time. We have completed preliminary experiments with a new implementation that allows multiple jobs to cooperate when executing concurrently. We have found that this makes much more efficient use of the resources of a large cluster, but are still exploring variants of the basic design. A full analysis of our experiments will be presented in a future publication.

There are many opportunities for improved performance monitoring and debugging. Each run of a large Dryad job generates statistics on the resource usage of thousands of executions of the same program on different input data. These statistics are already used to detect and re-execute slow-running "outlier" vertices. We plan to keep and analyze the statistics from a large number of jobs to look for patterns that can be used to predict the resource needs of vertices before they are executed. By feeding these predictions to our scheduler, we may be able to continue to make more efficient use of a shared cluster.

Much of the simplicity of the Dryad scheduler and fault-tolerance model come from the assumption that vertices are deterministic. If an application contains non-deterministic vertices then we might in future aim for the guarantee that every terminating execution produces an output that *some* failure-free execution could have generated. In the general case where vertices can produce side-effects this might be very hard to ensure automatically.

The Dryad system implements a general-purpose data-parallel execution engine. We have demonstrated excellent scaling behavior on small clusters, with absolute performance superior to a commercial database system for a hand-coded read-only query. On a larger cluster we have executed jobs containing hundreds of thousands of vertices, processing many terabytes of input data in minutes, and we can automatically adapt the computation to exploit network locality. We let developers easily create large-scale distributed applications without requiring them to master any concurrency techniques beyond being able to draw a graph of the data dependencies of their algorithms. We sacrifice some architectural simplicity compared with the MapReduce system design, but in exchange we release developers from the burden of expressing their code as a strict sequence of map, sort and reduce steps. We also allow the programmer the freedom to specify the communication transport which, for suitable tasks, delivers substantial performance gains.

Acknowledgements

We would like to thank all the members of the Cosmos team in Windows Live Search for their support and collaboration, and particularly Sam McKelvie for many helpful design discussions. Thanks to Jim Gray and our anonymous reviewers for suggestions on improving the presentation of the paper.

10. REFERENCES

[1] Global grid forum. `http://www.gridforum.org/`.
[2] Intel IXP2XXX product line of network processors. `http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm`.
[3] Intel platform 2015. `http://www.intel.com/technology/architecture/platform2015/`.

[4] The LINQ project. http://msdn.microsoft.com/netframework/future/linq/.

[5] Open MPI. http://www.open-mpi.org/.

[6] SQL Server Integration Services. http://www.microsoft.com/sql/technologies/integration/default.mspx.

[7] Thomas E. Anderson, David E. Culler, David A. Patterson, and NOW Team. A case for networks of workstations: NOW. *IEEE Micro*, pages 54–64, February 1995.

[8] Remzi H. Arpaci-Dusseau. Run-time adaptation in River. *Transactions on Computer Systems (TOCS)*, 21(1):36–86, 2003.

[9] Özalp Babaoğlu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. Paralex: an environment for parallel programming in distributed systems. pages 178–187, New York, NY, USA, 1992. ACM Press.

[10] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Mike Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD*, Baltimore, MD, June 2005.

[11] Tom Barclay, Robert Barnes, Jim Gray, and Prakash Sundaresan. Loading databases using dataflow parallelism. *SIGMOD Rec.*, 23(4):72–83, 1994.

[12] Chaitanya Baru and Gilles Fecteau. An overview of DB2 parallel edition. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 460–462, New York, NY, USA, 1995. ACM Press.

[13] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 19-21 1995.

[15] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Yury Markovskiy, André DeHon, and John Wawrzynek. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. In *FPL*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[16] Jeff Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, December 2004.

[17] D. DeWitt, S. Ghandeharizadeh, D. Schneider, H. Hsiao, A. Bricker, and R. Rasmussen. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[18] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.

[19] D. C. DiNucci and R. G. Babb II. Design and implementation of parallel programs with LGDF2. In *Digest of Papers from Compcon '89*, pages 102–107, 1989.

[20] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 78–91, New York, NY, USA, 1997. ACM Press.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[22] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 102–111, New York, NY, USA, 1990. ACM Press.

[23] J. Gray, A.S. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. Vandenberg. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, pages 189–210, Paris, France, March 2002. Carleton Scientific. also as MSR-TR-2002-01.

[24] Jim Gray and Alex Szalay. Science in an exponential world. *Nature*, 440(23), March 23 2006.

[25] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. A comparison of stream-oriented high-availability algorithms. Technical Report TR-03-17, Computer Science Department, Brown University, September 2003.

[26] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[27] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[28] James Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *Usenix Annual Technical Conference*, June 2002.

[29] Orlando Loques, Julius Leite, and Enrique Vinicio Carrera E. P-RIO: A modular parallel-programming environment. *IEEE Concurrency*, 6(1):47–57, 1998.

[30] William Mark, Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[31] P. Newton and J.C. Browne. The CODE 2.0 graphical parallel programming language. pages 167 – 177, Washington, D. C., United States, July 1992.

[32] Ken Phillips. SenSage ESA. *SC Magazine*, March 1 2006.

[33] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[34] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838, New York, NY, USA, 2004. ACM Press.

[35] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[36] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data-parallelism to program GPUs for general-purpose uses. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, October 2006. also as MSR-TR-2005-184.

[37] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[38] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4&5):469–510, 2002.