DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language

Yuan Yu Michael Isard Dennis Fetterly Mihai Budiu
Úlfar Erlingsson[1] Pradeep Kumar Gunda Jon Currey
Microsoft Research Silicon Valley [1]*joint affiliation, Reykjavík University, Iceland*

Abstract

DryadLINQ is a system and a set of language extensions that enable a new programming model for large scale distributed computing. It generalizes previous execution environments such as SQL, MapReduce, and Dryad in two ways: by adopting an expressive data model of strongly typed .NET objects; and by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language.

A DryadLINQ program is a sequential program composed of LINQ expressions performing arbitrary side-effect-free transformations on datasets, and can be written and debugged using standard .NET development tools. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform. Dryad, which has been in continuous operation for several years on production clusters made up of thousands of computers, ensures efficient, reliable execution of this plan.

We describe the implementation of the DryadLINQ compiler and runtime. We evaluate DryadLINQ on a varied set of programs drawn from domains such as web-graph analysis, large-scale log mining, and machine learning. We show that excellent absolute performance can be attained—a general-purpose sort of 10^{12} Bytes of data executes in 319 seconds on a 240-computer, 960-disk cluster—as well as demonstrating near-linear scaling of execution time on representative applications as we vary the number of computers used for a job.

1 Introduction

The DryadLINQ system is designed to make it easy for a wide variety of developers to compute effectively on large amounts of data. DryadLINQ programs are written as imperative or declarative operations on datasets within a traditional high-level programming language, using an expressive data model of strongly typed .NET objects. The main contribution of this paper is a set of language extensions and a corresponding system that can automatically and transparently compile imperative programs in a general-purpose language into distributed computations that execute efficiently on large computing clusters.

Our goal is to give the programmer the illusion of writing for a single computer and to have the system deal with the complexities that arise from scheduling, distribution, and fault-tolerance. Achieving this goal requires a wide variety of components to interact, including cluster-management software, distributed-execution middleware, language constructs, and development tools. Traditional parallel databases (which we survey in Section 6.1) as well as more recent data-processing systems such as MapReduce [15] and Dryad [26] demonstrate that it is possible to implement high-performance large-scale execution engines at modest financial cost, and clusters running such platforms are proliferating. Even so, their programming interfaces all leave room for improvement. We therefore believe that the language issues addressed in this paper are currently among the most pressing research areas for data-intensive computing, and our work on the DryadLINQ system stems from this belief.

DryadLINQ exploits LINQ (Language INtegrated Query [2], a set of .NET constructs for programming with datasets) to provide a powerful hybrid of declarative and imperative programming. The system is designed to provide flexible and efficient distributed computation in any LINQ-enabled programming language including C#, VB, and F#. Objects in DryadLINQ datasets can be of any .NET type, making it easy to compute with data such as image patches, vectors, and matrices. DryadLINQ programs can use traditional structuring constructs such as functions, modules, and libraries, and express iteration using standard loops. Crucially, the distributed execution layer employs a fully functional, declarative description of the data-parallel component of the computation, which enables sophisticated rewritings and optimizations like those traditionally employed by parallel databases.

In contrast, parallel databases implement only declarative variants of SQL queries. There is by now a widespread belief that SQL is too limited for many applications [15, 26, 31, 34, 35]. One problem is that, in order to support database requirements such as in-place updates and efficient transactions, SQL adopts a very restrictive type system. In addition, the declarative "query-oriented" nature of SQL makes it difficult to express common programming patterns such as iteration [14]. Together, these make SQL unsuitable for tasks such as machine learning, content parsing, and web-graph analysis that increasingly must be run on very large datasets.

The MapReduce system [15] adopted a radically simplified programming abstraction, however even common operations like database Join are tricky to implement in this model. Moreover, it is necessary to embed MapReduce computations in a scripting language in order to execute programs that require more than one reduction or sorting stage. Each MapReduce instantiation is self-contained and no automatic optimizations take place across their boundaries. In addition, the lack of any type-system support or integration between the MapReduce stages requires programmers to explicitly keep track of objects passed between these stages, and may complicate long-term maintenance and re-use of software components.

Several domain-specific languages have appeared on top of the MapReduce abstraction to hide some of this complexity from the programmer, including Sawzall [32], Pig [31], and other unpublished systems such as Facebook's HIVE. These offer a limited hybridization of declarative and imperative programs and generalize SQL's stored-procedure model. Some whole-query optimizations are automatically applied by these systems across MapReduce computation boundaries. However, these approaches inherit many of SQL's disadvantages, adopting simple custom type systems and providing limited support for iterative computations. Their support for optimizations is less advanced than that in DryadLINQ, partly because the underlying MapReduce execution platform is much less flexible than Dryad.

DryadLINQ and systems such as MapReduce are also distinguished from traditional databases [25] by having *virtualized* expression plans. The planner allocates resources independent of the actual cluster used for execution. This means both that DryadLINQ can run plans requiring many more steps than the instantaneously-available computation resources would permit, and that the computational resources can change dynamically, e.g. due to faults—in essence, we have an extra degree of freedom in buffer management compared with traditional schemes [21, 24, 27, 28, 29]. A downside of virtualization is that it requires intermediate results to be stored to persistent media, potentially increasing computation latency.

This paper makes the following contributions to the literature:

- We have demonstrated a new hybrid of declarative and imperative programming, suitable for large-scale data-parallel computing using a rich object-oriented programming language.
- We have implemented the DryadLINQ system and validated the hypothesis that DryadLINQ programs can be automatically optimized and efficiently executed on large clusters.
- We have designed a small set of operators that improve LINQ's support for coarse-grain parallelization while preserving its programming model.

Section 2 provides a high-level overview of the steps involved when a DryadLINQ program is run. Section 3 discusses LINQ and the extensions to its programming model that comprise DryadLINQ along with simple illustrative examples. Section 4 describes the DryadLINQ implementation and its interaction with the low-level Dryad primitives. In Section 5 we evaluate our system using several example applications at a variety of scales. Section 6 compares DryadLINQ to related work and Section 7 discusses limitations of the system and lessons learned from its development.

2 System Architecture

DryadLINQ compiles LINQ programs into distributed computations running on the Dryad cluster-computing infrastructure [26]. A Dryad job is a directed acyclic graph where each vertex is a program and edges represent data channels. At run time, vertices are processes communicating with each other through the channels, and each channel is used to transport a finite sequence of data records. The data model and serialization are provided by higher-level software layers, in this case DryadLINQ.

Figure 1 illustrates the Dryad system architecture. The execution of a Dryad job is orchestrated by a centralized "job manager." The job manager is responsible for: (1) instantiating a job's dataflow graph; (2) scheduling processes on cluster computers; (3) providing fault-tolerance by re-executing failed or slow processes; (4) monitoring the job and collecting statistics; and (5) transforming the job graph dynamically according to user-supplied policies.

A cluster is typically controlled by a task scheduler, separate from Dryad, which manages a batch queue of jobs and executes a few at a time subject to cluster policy.

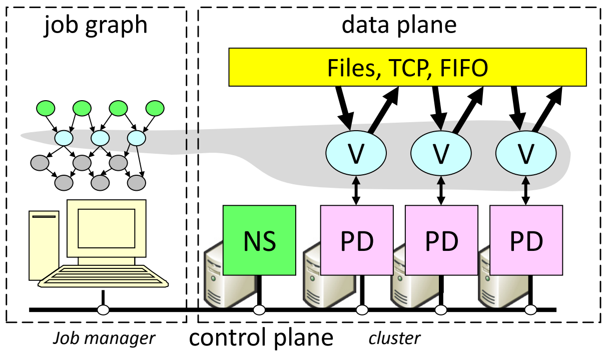

Figure 1: *Dryad system architecture. NS is the name server which maintains the cluster membership. The job manager is responsible for spawning vertices (V) on available computers with the help of a remote-execution and monitoring daemon (PD). Vertices exchange data through files, TCP pipes, or shared-memory channels. The grey shape indicates the vertices in the job that are currently running and the correspondence with the job execution graph.*

2.1 DryadLINQ Execution Overview

Figure 2 shows the flow of execution when a program is executed by DryadLINQ.

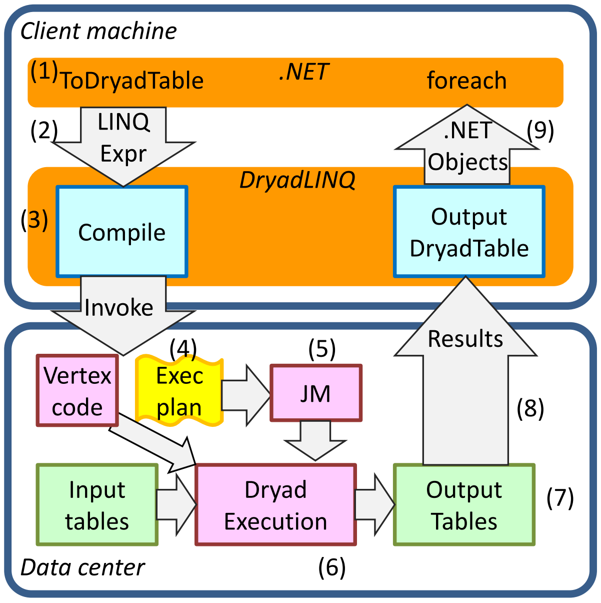

Figure 2: *LINQ-expression execution in DryadLINQ.*

Step 1. A .NET user application runs. It creates a DryadLINQ expression object. Because of LINQ's deferred evaluation (described in Section 3), the actual execution of the expression has not occurred.

Step 2. The application calls `ToDryadTable` triggering a data-parallel execution. The expression object is handed to DryadLINQ.

Step 3. DryadLINQ compiles the LINQ expression into a distributed Dryad execution plan. It performs: (a) the decomposition of the expression into subexpressions, each to be run in a separate Dryad vertex; (b) the generation of code and static data for the remote Dryad vertices; and (c) the generation of serialization code for the required data types. Section 4 describes these steps in detail.

Step 4. DryadLINQ invokes a custom, DryadLINQ-specific, Dryad job manager. The job manager may be executed behind a cluster firewall.

Step 5. The job manager creates the job graph using the plan created in Step 3. It schedules and spawns the vertices as resources become available. See Figure 1.

Step 6. Each Dryad vertex executes a vertex-specific program (created in Step 3b).

Step 7. When the Dryad job completes successfully it writes the data to the output table(s).

Step 8. The job manager process terminates, and it returns control back to DryadLINQ. DryadLINQ creates the local `DryadTable` objects encapsulating the outputs of the execution. These objects may be used as inputs to subsequent expressions in the user program. Data objects within a `DryadTable` output are fetched to the local context only if explicitly dereferenced.

Step 9. Control returns to the user application. The iterator interface over a `DryadTable` allows the user to read its contents as .NET objects.

Step 10. The application may generate subsequent DryadLINQ expressions, to be executed by a repetition of Steps 2–9.

3 Programming with DryadLINQ

In this section we highlight some particularly useful and distinctive aspects of DryadLINQ. More details on the programming model may be found in LINQ language reference [2] and materials on the DryadLINQ project website [1] including a language tutorial. A companion technical report [38] contains listings of some of the sample programs described below.

3.1 LINQ

The term LINQ [2] refers to a set of .NET constructs for manipulating sets and sequences of data items. We describe it here as it applies to C# but DryadLINQ programs have been written in other .NET languages including F#. The power and extensibility of LINQ derive from a set of design choices that allow the programmer to express complex computations over datasets while giving the runtime great leeway to decide how these computations should be implemented.

The base type for a LINQ collection is `IEnumerable<T>`. From a programmer's perspective, this is an abstract dataset of objects of type T that is accessed using an iterator interface. LINQ also defines the IQueryable<T> interface which is a subtype of IEnumerable<T> and represents an (unevaluated) expression constructed by combining LINQ datasets using LINQ operators. We need make only two observations about these types: (a) in general the programmer neither knows nor cares what concrete type implements any given dataset's IEnumerable interface; and (b) DryadLINQ composes all LINQ expressions into IQueryable objects and defers evaluation until the result is needed, at which point the expression graph within the IQueryable is optimized and executed in its entirety on the cluster. Any IQueryable object can be used as an argument to multiple operators, allowing efficient re-use of common subexpressions.

LINQ expressions are statically strongly typed through use of nested generics, although the compiler hides much of the type-complexity from the user by providing a range of "syntactic sugar." Figure 3 illustrates LINQ's syntax with a fragment of a simple example program that computes the top-ranked results for each query in a stored corpus. Two versions of the same LINQ expression are shown, one using a declarative SQL-like syntax, and the second using the object-oriented style we adopt for more complex programs.

The program first performs a Join to "look up" the static rank of each document contained in a scoreTriples tuple and then computes a new rank for that tuple, combining the query-dependent score with the static score inside the constructor for QueryScoreDocIDTriple. The program next groups the resulting tuples by query, and outputs the top-ranked results for each query. The full example program is included in [38].

The second, object-oriented, version of the example illustrates LINQ's use of C#'s lambda expressions. The Join method, for example, takes as arguments a dataset to perform the Join against (in this case staticRank) and three functions. The first two functions describe how to determine the keys that should be used in the Join. The third function describes the Join function itself. Note that the compiler performs static type inference to determine the concrete types of var objects and anonymous lambda expressions so the programmer need not remember (or even know) the type signatures of many subexpressions or helper functions.

3.2 DryadLINQ Constructs

DryadLINQ preserves the LINQ programming model and extends it to data-parallel programming by defining a small set of new operators and datatypes.

The DryadLINQ data model is a distributed implementation of LINQ collections. Datasets may still contain arbitrary .NET types, but each DryadLINQ dataset is in general distributed across the computers of a cluster, partitioned into disjoint pieces as shown in Figure 4. The partitioning strategies used—hash-partitioning, range-partitioning, and round-robin—are familiar from parallel databases [18]. This dataset partitioning is managed transparently by the system unless the programmer explicitly overrides the optimizer's choices.

The inputs and outputs of a DryadLINQ computation are represented by objects of type DryadTable<T>, which is a subtype of IQueryable<T>. Subtypes of DryadTable<T> support underlying storage providers that include distributed filesystems, collections of NTFS files, and sets of SQL tables. DryadTable objects may include metadata read from the file system describing table properties such as schemas for the data items contained in the table, and partitioning schemes which the DryadLINQ optimizer can use to generate more efficient executions. These optimizations, along with issues such

```
// SQL-style syntax to join two input sets:
// scoreTriples and staticRank
var adjustedScoreTriples =
  from d in scoreTriples
  join r in staticRank on d.docID equals r.key
  select new QueryScoreDocIDTriple(d, r);
var rankedQueries =
  from s in adjustedScoreTriples
  group s by s.query into g
  select TakeTopQueryResults(g);

// Object-oriented syntax for the above join
var adjustedScoreTriples =
  scoreTriples.Join(staticRank,
    d => d.docID, r => r.key,
    (d, r) => new QueryScoreDocIDTriple(d, r));
var groupedQueries =
  adjustedScoreTriples.GroupBy(s => s.query);
var rankedQueries =
  groupedQueries.Select(
    g => TakeTopQueryResults(g));
```

Figure 3: *A program fragment illustrating two ways of expressing the same operation. The first uses LINQ's declarative syntax, and the second uses object-oriented interfaces. Statements such as* r => r.key *use C#'s syntax for anonymous lambda expressions.*

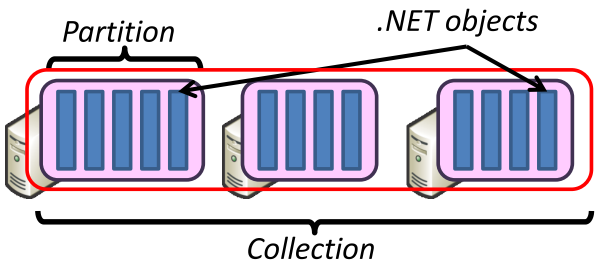

Figure 4: *The DryadLINQ data model: strongly-typed collections of .NET objects partitioned on a set of computers.*

as data serialization and compression, are discussed in Section 4.

The primary restriction imposed by the DryadLINQ system to allow distributed execution is that *all the functions called in DryadLINQ expressions must be side-effect free*. Shared objects can be referenced and read freely and will be automatically serialized and distributed where necessary. However, if any shared object is modified, the result of the computation is undefined. DryadLINQ does not currently check or enforce the absence of side-effects.

The inputs and outputs of a DryadLINQ computation are specified using the `GetTable<T>` and `ToDryadTable<T>` operators, e.g.:

```
var input = GetTable<LineRecord>("file://in.tbl");
var result = MainProgram(input, ...);
var output = ToDryadTable(result, "file://out.tbl");
```

Tables are referenced by URI strings that indicate the storage system to use as well as the name of the partitioned dataset. Variants of `ToDryadTable` can simultaneously invoke multiple expressions and generate multiple output `DryadTables` in a single distributed Dryad job. This feature (also encountered in parallel databases such as Teradata) can be used to avoid recomputing or materializing common subexpressions.

DryadLINQ offers two data re-partitioning operators: `HashPartition<T,K>` and `RangePartition<T,K>`. These operators are needed to enforce a partitioning on an output dataset and they may also be used to override the optimizer's choice of execution plan. From a LINQ perspective, however, they are no-ops since they just reorganize a collection without changing its contents. The system allows the implementation of additional re-partitioning operators, but we have found these two to be sufficient for a wide class of applications.

The remaining new operators are `Apply` and `Fork`, which can be thought of as an "escape-hatch" that a programmer can use when a computation is needed that cannot be expressed using any of LINQ's built-in operators. `Apply` takes a function `f` and passes to it an iterator over the entire input collection, allowing arbitrary streaming computations. As a simple example, `Apply` can be used to perform "windowed" computations on a sequence, where the ith entry of the output sequence is a function on the range of input values $[i, i + d]$ for a fixed window of length d. The applications of `Apply` are much more general than this and we discuss them further in Section 7. The `Fork` operator is very similar to `Apply` except that it takes a single input and generates multiple output datasets. This is useful as a performance optimization to eliminate common subcomputations, e.g. to implement a document parser that outputs both plain text and a bibliographic entry to separate tables.

If the DryadLINQ system has no further information about `f`, `Apply` (or `Fork`) will cause all of the computation to be serialized onto a single computer. More often, however, the user supplies annotations on `f` that indicate conditions under which `Apply` can be parallelized. The details are too complex to be described in the space available, but quite general "conditional homomorphism" is supported—this means that the application can specify conditions on the partitioning of a dataset under which `Apply` can be run independently on each partition. DryadLINQ will automatically re-partition the data to match the conditions if necessary.

DryadLINQ allows programmers to specify annotations of various kinds. These provide manual hints to guide optimizations that the system is unable to perform automatically, while preserving the semantics of the program. As mentioned above, the `Apply` operator makes use of annotations, supplied as simple .NET attributes, to indicate opportunities for parallelization. There are also `Resource` annotations to discriminate functions that require constant storage from those whose storage grows along with the input collection size—these are used by the optimizer to determine buffering strategies and decide when to pipeline operators in the same process. The programmer may also declare that a dataset has a particular partitioning scheme if the file system does not store sufficient metadata to determine this automatically.

The DryadLINQ optimizer produces good automatic execution plans for most programs composed of standard LINQ operators, and annotations are seldom needed unless an application uses complex `Apply` statements.

3.3 Building on DryadLINQ

Many programs can be directly written using the DryadLINQ primitives. Nevertheless, we have begun to build libraries of common subroutines for various application domains. The ease of defining and maintaining such libraries using C#'s functions and interfaces highlights the advantages of embedding data-parallel constructs within a high-level language.

The MapReduce programming model from [15] can be compactly stated as follows (eliding the precise type signatures for clarity):

```
public static MapReduce( // returns set of Rs
    source, // set of Ts
    mapper, // function from T → Ms
    keySelector, // function from M → K
    reducer // function from (K,Ms) → Rs
) {
    var mapped = source.SelectMany(mapper);
    var groups = mapped.GroupBy(keySelector);
    return groups.SelectMany(reducer);
}
```

Section 4 discusses the execution plan that is automatically generated for such a computation by the DryadLINQ optimizer.

We built a general-purpose library for manipulating numerical data to use as a platform for implementing machine-learning algorithms, some of which are described in Section 5. The applications are written as traditional programs calling into library functions, and make no explicit reference to the distributed nature of the computation. Several of these algorithms need to iterate over a data transformation until convergence. In a traditional database this would require support for recursive expressions, which are tricky to implement [14]; in DryadLINQ it is trivial to use a C# loop to express the iteration. The companion technical report [38] contains annotated source for some of these algorithms.

4 System Implementation

This section describes the DryadLINQ parallelizing compiler. We focus on the generation, optimization, and execution of the distributed execution plan, corresponding to step 3 in Figure 2. The DryadLINQ optimizer is similar in many respects to classical database optimizers [25]. It has a static component, which generates an execution plan, and a dynamic component, which uses Dryad policy plug-ins to optimize the graph at run time.

4.1 Execution Plan Graph

When it receives control, DryadLINQ starts by converting the raw LINQ expression into an execution plan graph (EPG), where each node is an operator and edges represent its inputs and outputs. The EPG is closely related to a traditional database query plan, but we use the more general terminology of execution plan to encompass computations that are not easily formulated as "queries." The EPG is a directed acyclic graph—the existence of common subexpressions and operators like `Fork` means that EPGs cannot always be described by trees. DryadLINQ then applies term-rewriting optimizations on the EPG. The EPG is a "skeleton" of the Dryad data-flow graph that will be executed, and each EPG node is replicated at run time to generate a Dryad "stage" (a collection of vertices running the same computation on different partitions of a dataset). The optimizer annotates the EPG with metadata properties. For edges, these include the .NET type of the data and the compression scheme, if any, used after serialization. For nodes, they include details of the partitioning scheme used, and ordering information within each partition. The output of a node, for example, might be a dataset that is hash-partitioned by a particular key, and sorted according to that key within each partition; this information can be used by subsequent `OrderBy` nodes to choose an appropriate distributed sort algorithm as described below in Section 4.2.3. The properties are seeded from the LINQ expression tree and the input and output tables' metadata, and propagated and updated during EPG rewriting.

Propagating these properties is substantially harder in the context of DryadLINQ than for a traditional database. The difficulties stem from the much richer data model and expression language. Consider one of the simplest operations: `input.Select(x => f(x))`. If `f` is a simple expression, e.g. `x.name`, then it is straightforward for DryadLINQ to determine which properties can be propagated. However, for arbitrary `f` it is in general impossible to determine whether this transformation preserves the partitioning properties of the input.

Fortunately, DryadLINQ can usually infer properties in the programs typical users write. Partition and sort key properties are stored as expressions, and it is often feasible to compare these for equality using a combination of static typing, static analysis, and reflection. The system also provides a simple mechanism that allows users to assert properties of an expression when they cannot be determined automatically.

4.2 DryadLINQ Optimizations

DryadLINQ performs both static and dynamic optimizations. The static optimizations are currently greedy heuristics, although in the future we may implement cost-based optimizations as used in traditional databases. The dynamic optimizations are applied during Dryad job execution, and consist in rewriting the job graph depending on run-time data statistics. Our optimizations are sound in that a failure to compute properties simply results in an inefficient, though correct, execution plan.

4.2.1 Static Optimizations

DryadLINQ's static optimizations are conditional graph rewriting rules triggered by a predicate on EPG node properties. Most of the static optimizations are focused on minimizing disk and network I/O. The most important are:

Pipelining: Multiple operators may be executed in a single process. The pipelined processes are themselves LINQ expressions and can be executed by an existing single-computer LINQ implementation.

Removing redundancy: DryadLINQ removes unnecessary hash- or range-partitioning steps.

Eager Aggregation: Since re-partitioning datasets is expensive, down-stream aggregations are moved in front of partitioning operators where possible.

I/O reduction: Where possible, DryadLINQ uses Dryad's TCP-pipe and in-memory FIFO channels instead of persisting temporary data to files. The system by default compresses data before performing a partitioning, to reduce network traffic. Users can manually override compression settings to balance CPU usage with network load if the optimizer makes a poor decision.

4.2.2 Dynamic Optimizations

DryadLINQ makes use of hooks in the Dryad API to dynamically mutate the execution graph as information from the running job becomes available. Aggregation gives a major opportunity for I/O reduction since it can be optimized into a tree according to locality, aggregating data first at the computer level, next at the rack level, and finally at the cluster level. The topology of such an aggregation tree can only be computed at run time, since it is dependent on the dynamic scheduling decisions which allocate vertices to computers. DryadLINQ automatically uses the dynamic-aggregation logic present in Dryad [26].

Dynamic data partitioning sets the number of vertices in each stage (i.e., the number of partitions of each dataset) at run time based on the size of its input data. Traditional databases usually estimate dataset sizes statically, but these estimates can be very inaccurate, for example in the presence of correlated queries. DryadLINQ supports dynamic hash and range partitions—for range partitions both the number of partitions and the partitioning key ranges are determined at run time by sampling the input dataset.

4.2.3 Optimizations for `OrderBy`

DryadLINQ's logic for sorting a dataset `d` illustrates many of the static and dynamic optimizations available to the system. Different strategies are adopted depending on `d`'s initial partitioning and ordering. Figure 5 shows the evolution of an `OrderBy` node `O` in the most complex case, where `d` is not already range-partitioned by the correct sort key, nor are its partitions individually ordered by the key. First, the dataset must be re-partitioned. The `DS` stage performs deterministic sampling of the input dataset. The samples are aggregated by a histogram vertex `H`, which determines the partition keys as a function of data distribution (load-balancing the computation in the next stage). The `D` vertices perform the actual repartitioning, based on the key ranges computed by `H`. Next, a merge node `M` interleaves the inputs, and a `S` node sorts them. `M` and `S` are pipelined in a single process, and communicate using iterators. The number of partitions in the `DS+H+D` stage is chosen at run time

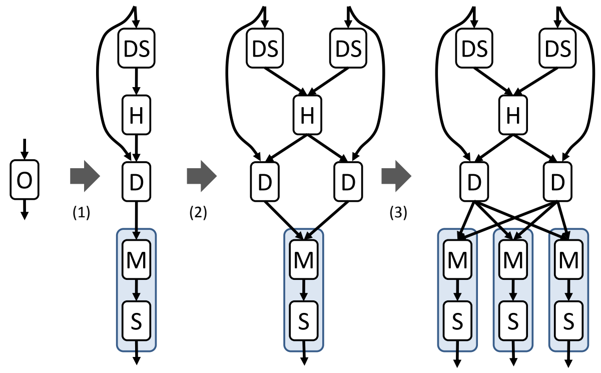

Figure 5: *Distributed sort optimization described in Section 4.2.3. Transformation (1) is static, while (2) and (3) are dynamic.*

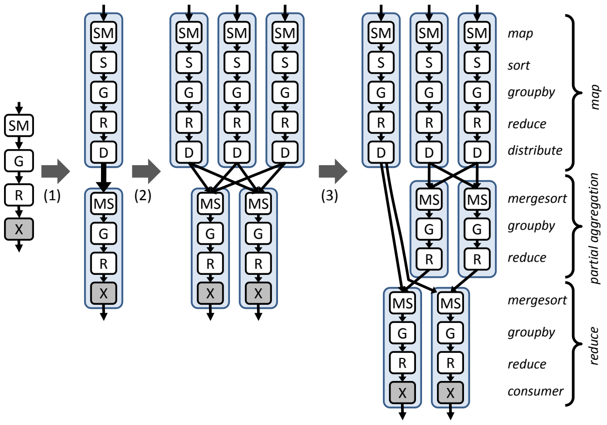

Figure 6: *Execution plan for MapReduce, described in Section 4.2.4. Step (1) is static, (2) and (3) are dynamic based on the volume and location of the data in the inputs.*

based on the number of partitions in the preceding computation, and the number of partitions in the `M+S` stage is chosen based on the volume of data to be sorted (transitions (2) and (3) in Figure 5).

4.2.4 Execution Plan for MapReduce

This section analyzes the execution plan generated by DryadLINQ for the MapReduce computation from Section 3.3. Here, we examine only the case when the input to `GroupBy` is not ordered and the reduce function is determined to be associative and commutative. The automatically generated execution plan is shown in Figure 6. The plan is statically transformed (1) into a Map and a Reduce stage. The Map stage applies the `SelectMany` operator (`SM`) and then sorts each partition (`S`), performs a local GroupBy (`G`) and finally a local reduction (`R`). The `D` nodes perform a hash-partition. All these operations are pipelined together in a single process. The Reduce stage first merge-sorts all the incoming data streams (`MS`). This is followed by another `GroupBy` (`G`) and the final reduction (`R`). All these Reduce stage operators are pipelined in a single process along with the subsequent operation in the computation (`X`). As with the sort plan in Section 4.2.3, at run time (2) the number of Map instances is automatically determined using the number of input partitions, and the number of Reduce instances is chosen based on the total volume of data to be Reduced. If necessary, DryadLINQ will insert a dynamic aggregation tree (3) to reduce the amount of data that crosses the network. In the figure, for example, the two rightmost input partitions were processed on the same computer, and their outputs have been pre-aggregated locally before being transferred across the network and combined with the output of the leftmost partition.

The resulting execution plan is very similar to the manually constructed plans reported for Google's MapReduce [15] and the Dryad histogram computation in [26]. The crucial point to note is that in DryadLINQ MapReduce is not a primitive, hard-wired operation, and *all* user-specified computations gain the benefits of these powerful automatic optimization strategies.

4.3 Code Generation

The EPG is used to derive the Dryad execution plan after the static optimization phase. While the EPG encodes all the required information, it is not a runnable program. DryadLINQ uses dynamic code generation to automatically synthesize LINQ code to be run at the Dryad vertices. The generated code is compiled into a .NET assembly that is shipped to cluster computers at execution time. For each execution-plan stage, the assembly contains two pieces of code:

(1) The code for the LINQ subexpression executed by each node.

(2) Serialization code for the channel data. This code is much more efficient than the standard .NET serialization methods since it can rely on the contract between the reader and writer of a channel to access the same statically known datatype.

The subexpression in a vertex is built from pieces of the overall EPG passed in to DryadLINQ. The EPG is created in the original client computer's execution context, and may depend on this context in two ways:

(1) The expression may reference variables in the local context. These references are eliminated by partial evaluation of the subexpression at code-generation time. For primitive values, the references in the expressions are replaced with the actual values. Object values are serialized to a resource file which is shipped to computers in the cluster at execution time.

(2) The expression may reference .NET libraries. .NET reflection is used to find the transitive closure of all non-system libraries referenced by the executable, and these are shipped to the cluster computers at execution time.

4.4 Leveraging Other LINQ Providers

One of the greatest benefits of using the LINQ framework is that DryadLINQ can leverage other systems that use the same constructs. DryadLINQ currently gains most from the use of PLINQ [19], which allows us to run the code within each vertex in parallel on a multi-core server. PLINQ, like DryadLINQ, attempts to make the process of parallelizing a LINQ program as transparent as possible, though the systems' implementation strategies are quite different. Space does not permit a detailed explanation, but PLINQ employs the iterator model [25] since it is better suited to fine-grain concurrency in a shared-memory multi-processor system. Because both PLINQ and DryadLINQ use expressions composed from the same LINQ constructs, it is straightforward to combine their functionality. DryadLINQ's vertices execute LINQ expressions, and in general the addition by the DryadLINQ code generator of a single line to the vertex's program triggers the use of PLINQ, allowing the vertex to exploit all the cores in a cluster computer. We note that this remarkable fact stems directly from the careful design choices that underpin LINQ.

We have also added interoperation with the LINQ-to-SQL system which lets DryadLINQ vertices directly access data stored in SQL databases. Running a database on each cluster computer and storing tables partitioned across these databases may be much more efficient than using flat disk files for some applications. DryadLINQ programs can use "partitioned" SQL tables as input and output. DryadLINQ also identifies and ships some subexpressions to the SQL databases for more efficient execution.

Finally, the default single-computer LINQ-to-Objects implementation allows us to run DryadLINQ programs on a single computer for testing on small inputs under the control of the Visual Studio debugger before executing on a full cluster dataset.

4.5 Debugging

Debugging a distributed application is a notoriously difficult problem. Most DryadLINQ jobs are long running, processing massive datasets on large clusters, which could make the debugging process even more challenging. Perhaps surprisingly, we have not found debugging the correctness of programs to be a major challenge when using DryadLINQ. Several users have commented that LINQ's strong typing and narrow interface have turned up many bugs before a program is even executed. Also, as mentioned in Section 4.4, DryadLINQ supports a straightforward mechanism to run applications on a single computer, with very sophisticated support from the .NET development environment.

Once an application is running on the cluster, an individual vertex may fail due to unusual input data that manifests problems not apparent from a single-computer test. A consequence of Dryad's deterministic-replay execution model, however, is that it is straightforward to re-execute such a vertex in isolation with the inputs that caused the failure, and the system includes scripts to ship the vertex executable, along with the problematic partitions, to a local computer for analysis and debugging.

Performance debugging is a much more challenging problem in DryadLINQ today. Programs report summary information about their overall progress, but if particular stages of the computation run more slowly than expected, or their running time shows surprisingly high variance, it is necessary to investigate a collection of disparate logs to diagnose the issue manually. The centralized nature of the Dryad job manager makes it straightforward to collect profiling information to ease this task, and simplifying the analysis of these logs is an active area of our current research.

5 Experimental Evaluation

We have evaluated DryadLINQ on a set of applications drawn from domains including web-graph analysis, large-scale log mining, and machine learning. All of our performance results are reported for a medium-sized private cluster described in Section 5.1. Dryad has been in continuous operation for several years on production clusters made up of thousands of computers so we are confident in the scaling properties of the underlying execution engine, and we have run large-scale DryadLINQ programs on those production clusters.

5.1 Hardware Configuration

The experiments described in this paper were run on a cluster of 240 computers. Each of these computers was running the Windows Server 2003 64-bit operating system. The computers' principal components were two dual-core AMD Opteron 2218 HE CPUs with a clock speed of 2.6 GHz, 16 GBytes of DDR2 random access memory, and four 750 GByte SATA hard drives. The computers had two partitions on each disk. The first, small, partition was occupied by the operating system on one disk and left empty on the remaining disks. The remaining partitions on each drive were striped together to form a large data volume spanning all four disks. The computers were each connected to a Linksys SRW2048 48-port full-crossbar GBit Ethernet local switch via GBit Ethernet. There were between 29 and 31 computers connected to each local switch. Each local switch was in turn connected to a central Linksys SRW2048 switch, via 6 ports aggregated using 802.3ad link aggregation. This gave each local switch up to 6 GBits per second of full duplex connectivity. Note that the switches are commodity parts purchased for under $1000 each.

5.2 Terasort

In this experiment we evaluate DryadLINQ using the Terasort benchmark [3]. The task is to sort 10 billion 100-Byte records using case-insensitive string comparison on a 10-Byte key. We use the data generator described in [3]. The DryadLINQ program simply defines the record type, creates a `DryadTable` for the partitioned inputs, and calls `OrderBy`; the system then automatically generates an execution plan using dynamic range-partitioning as described in Section 4.2.3 (though for the purposes of running a controlled experiment we manually set the number of partitions for the sorting stage).

For this experiment, each computer in the cluster stored a partition of size around 3.87 GBytes ($4,166,666,600$ Bytes). We varied the number of computers used, so for an execution using n computers, the total data sorted is $3.87n$ GBytes. On the largest run $n = 240$ and we sort 10^{12} Bytes of data. The most time-consuming phase of this experiment is the network read to range-partition the data. However, this is overlapped with the sort, which processes inputs in batches in parallel and generates the output by merge-sorting the sorted batches. DryadLINQ automatically compresses the data before it is transferred across the network—when sorting 10^{12} Bytes of data, 150 GBytes of compressed data were transferred across the network.

Table 1 shows the elapsed times in seconds as the number of machines increases from 1 to 240, and thus the data sorted increases from 3.87 GBytes to 10^{12} Bytes. On repeated runs the times were consistent to within 5% of their averages. Figure 7 shows the same information in graphical form. For the case of a single partition, DryadLINQ uses a very different execution plan, skipping the sampling and partitioning stages. It thus reads the input data only once, and does not perform any network transfers. The single-partition time is therefore the baseline time for reading a partition, sorting it, and writing the output. For $2 \leq n \leq 20$ all computers were connected to the same local switch, and the elapsed time stays fairly constant. When $n > 20$ the elapsed time seems to be approaching an asymptote as we increase the number of computers. We interpret this to mean that the cluster is well-provisioned: we do not saturate the core

Computers	1	2	10	20	40	80	240
Time	119	241	242	245	271	294	319

Table 1: Time in seconds to sort different amounts of data. The total data sorted by an n-machine experiment is around $3.87n$ GBytes, or 10^{12} Bytes when $n = 240$.

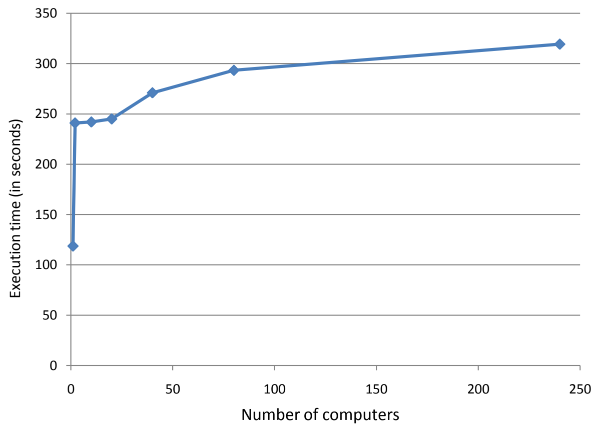

Figure 7: *Sorting increasing amounts of data while keeping the volume of data per computer fixed. The total data sorted by an n-machine experiment is around $3.87n$ GBytes, or 10^{12} Bytes when $n = 240$.*

Computers	1	5	10	20	40
Dryad	2167	451	242	135	92
DryadLINQ	2666	580	328	176	113

Table 2: Time in seconds to process skyserver Q18 using different number of computers.

network even when performing a dataset repartitioning across all computers in the cluster.

5.3 SkyServer

For this experiment we implemented the most time-consuming query (Q18) from the Sloan Digital Sky Survey database [23]. The query identifies a "gravitational lens" effect by comparing the locations and colors of stars in a large astronomical table, using a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. In this experiment, we compare the performance of the two-pass variant of the Dryad program described in [26] with that of DryadLINQ. The Dryad program is around 1000 lines of C++ code whereas the corresponding DryadLINQ program is only around 100 lines of C#. The input tables were manually range-partitioned into 40 partitions using the same keys. We varied n, the number of computers used, to investigate the scaling performance. For a given n we ensured that the tables were distributed such that each computer had approximately $40/n$ partitions of each, and that for a given partition key-range the data from the two tables was stored on the same computer.

Table 2 shows the elapsed times in seconds for the native Dryad and DryadLINQ programs as we varied n between 1 and 40. On repeated runs the times were consistent to within 3.5% of their averages. The DryadLINQ implementation is around 1.3 times slower than the native Dryad job. We believe the slowdown is mainly due

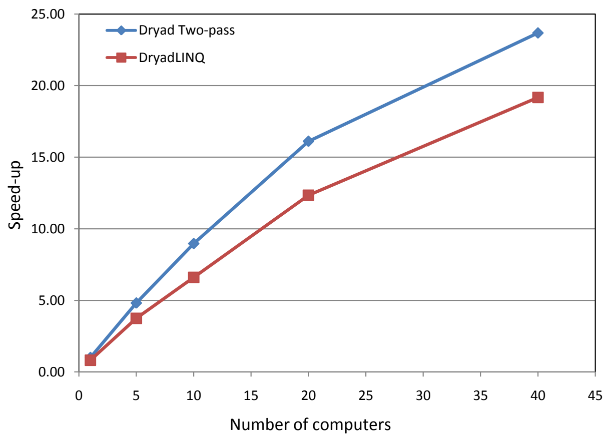

Figure 8: *The speed-up of the Skyserver Q18 computation as the number of computers is varied. The baseline is relative to DryadLINQ job running on a single computer and times are given in Table 2.*

to a hand-tuned sort strategy used by the Dryad program, which is somewhat faster than DryadLINQ's automatic parallel sort implementation. However, the DryadLINQ program is written at a much higher level. It abstracts much of the distributed nature of the computation from the programmer, and is only 10% of the length of the native code.

Figure 8 graphs the inverse of the running times, normalized to show the speed-up factor relative to the two-pass single-computer Dryad version. For $n \leq 20$ all computers were connected to the same local switch, and the speedup factor is approximately proportional to the number of computers used. When $n = 40$ the computers must communicate through the core switch and the scaling becomes sublinear.

5.4 PageRank

We also evaluate the performance of DryadLINQ at performing PageRank calculations on a large web graph. PageRank is a conceptually simple iterative computation for scoring hyperlinked pages. Each page starts with a real-valued score. At each iteration every page distributes its score across its outgoing links and updates its score to the sum of values received from pages linking to it. Each iteration of PageRank is a fairly simple relational query. We first Join the set of links with the set of ranks, using the source as the key. This results in a set of scores, one for each link, that we can accumulate using a GroupBy-Sum with the link's destinations as keys. We compare two implementations: an initial "naive" attempt and an optimized version.

Our first DryadLINQ implementation follows the outline above, except that the links are already grouped by source (this is how the crawler retrieves them). This makes the Join less complicated—once per page rather than once per link—but requires that we follow it with a SelectMany, to produce the list of scores to aggregate. This naive implementation takes 93 lines of code, including 35 lines to define the input types.

The naive approach scales well, but is inefficient because it must reshuffle data proportional to the number of links to aggregate the transmitted scores. We improve on it by first HashPartitioning the link data by a hash of the hostname of the source, rather than a hash of the page name. The result is that most of the rank updates are written back locally—80%-90% of web links are host-local—and do not touch the network. It is also possible to cull leaf pages from the graph (and links to them); they do not contribute to the iterative computation, and needn't be kept in the inner loop. Further performance optimizations, like pre-grouping the web pages by host (+7 LOC), rewriting each of these host groups with dense local names (+21 LOC), and pre-aggregating the ranks from each host (+18 LOC) simplify the computation further and ease the memory footprint. The complete source code for our implementation of PageRank is contained in the companion technical report [38].

We evaluate both of these implementations (running on 240 computers) on a large web graph containing 954M pages and 16.5B links, occupying 1.2 TB compressed on disk. The naive implementation, including pre-aggregation, executes 10 iterations in 12,792 seconds. The optimized version, which further compresses the graph down to 116 GBytes, executes 10 iterations in 690 seconds.

It is natural to compare our PageRank implementation with similar implementations using other platforms. MapReduce, Hadoop, and Pig all use the MapReduce computational framework, which has trouble efficiently implementing Join due to its requirement that all input (including the web graph itself) be output of the previous stage. By comparison, DryadLINQ can partition the web graph once, and reuse that graph in multiple stages without moving any data across the network. It is important to note that the Pig language masks the complexity of Joins, but they are still executed as MapReduce computations, thus incurring the cost of additional data movement. SQL-style queries can permit Joins, but suffer from their rigid data types, preventing the pre-grouping of links by host and even by page.

5.5 Large-Scale Machine Learning

We ran two machine-learning experiments to investigate DryadLINQ's performance on iterative numerical algorithms.

The first experiment is a clustering algorithm for detecting botnets. We analyze around 2.1 GBytes of data, where each datum is a three-dimensional vector summarizing salient features of a single computer, and group them into 3 clusters. The algorithm was written using the machine-learning framework described in Section 3.3 in 160 lines of C#. The computation has three stages: (1) parsing and re-partitioning the data across all the computers in the cluster; (2) counting the records; and (3) performing an iterative E–M computation. We always perform 10 iterations (ignoring the convergence criterion) grouped into two blocks of 5 iterations, materializing the results every 5 iterations. Some stages are CPU-bound (performing matrix algebra), while other are I/O bound. The job spawns about 10,000 processes across the 240 computers, and completes end-to-end in 7 minutes and 11 seconds, using about 5 hours of effective CPU time.

We also used DryadLINQ to apply statistical inference algorithms [33] to automatically discover network-wide relationships between hosts and services on a medium-size network (514 hosts). For each network host the algorithms compose a dependency graph by analyzing timings between input/output packets. The input is processed header data from a trace of 11 billion packets (180 GBytes packed using a custom compression format into 40 GBytes). The main body of this DryadLINQ program is just seven lines of code. It hash-partitions the data using the pair `(host,hour)` as a key, applies a doubly-nested E–M algorithm and hypothesis testing (which takes 95% of the running time), partitions again by hour, and finally builds graphs for all 174,588 active host hours. The computation takes 4 hours and 22 minutes, and more than 10 days of effective CPU time.

6 Related Work

DryadLINQ builds on decades of previous work in distributed computation. The most obvious connections are with parallel databases, grid and cluster computing, parallel and high-performance computation, and declarative programming languages.

Many of the salient features of DryadLINQ stem from the high-level system architecture. In our model of cluster computing the three layers of storage, execution, and application are decoupled. The system can make use of a variety of storage layers, from raw disk files to distributed filesystems and even structured databases. The Dryad distributed execution environment provides generic distributed execution services for acyclic networks of processes. DryadLINQ supplies the application layer.

6.1 Parallel Databases

Many of the core ideas employed by DryadLINQ (such as shared-nothing architecture, horizontal data partitioning, dynamic repartitioning, parallel query evaluation, and dataflow scheduling), can be traced to early research projects in parallel databases [18], such as Gamma [17], Bubba [8], and Volcano [22], and found in commercial products for data warehousing such as Teradata, IBM DB2 Parallel Edition [4], and Tandem SQL/MP [20].

Although DryadLINQ builds on many of these ideas, it is not a traditional database. For example, DryadLINQ provides a generalization of the concept of query language, but it does not provide a data definition language (DDL) or a data management language (DML) and it does not provide support for in-place table updates or transaction processing. We argue that the DDL and DML belong to the storage layer, so they should not be a first-class part of the application layer. However, as Section 4.2 explains, the DryadLINQ optimizer does make use of partitioning and typing information available as metadata attached to input datasets, and will write such metadata back to an appropriately configured storage layer.

Traditional databases offer extensibility beyond the simple relational data model through embedded languages and stored procedures. DryadLINQ (following LINQ's design) turns this relationship around, and embeds the expression language in the high-level programming language. This allows DryadLINQ to provide very rich native datatype support: almost all native .NET types can be manipulated as typed, first-class objects.

In order to enable parallel expression execution, DryadLINQ employs many traditional parallelization and query optimization techniques, centered on horizontal data partitioning. As mentioned in the Introduction, the expression plan generated by DryadLINQ is virtualized. This virtualization underlies DryadLINQ's dynamic optimization techniques, which have not previously been reported in the literature [16].

6.2 Large Scale Data-Parallel Computation Infrastructure

The last decade has seen a flurry of activity in architectures for processing very large datasets (as opposed to traditional high-performance computing which is typically CPU-bound). One of the earliest commercial generic platforms for distributed computation was the Teoma Neptune platform [13], which introduced a map-reduce computation paradigm inspired by MPI's Reduce operator. The Google MapReduce framework [15] slightly extended the computation model, separated the execution layer from storage, and virtualized the execution. The Hadoop open-source port of MapReduce uses the same architecture. NetSolve [5] proposed a grid-based architecture for a generic execution layer. DryadLINQ has a richer set of operators and better language support than any of these other proposals.

At the storage layer a variety of very large scale simple databases have appeared, including Google's BigTable [11], Amazon's Simple DB, and Microsoft SQL Server Data Services. Architecturally, DryadLINQ is just an application running on top of Dryad and generating distributed Dryad jobs. We can envision making it interoperate with any of these storage layers.

6.3 Declarative Programming Languages

Notable research projects in parallel declarative languages include Parallel Haskell [37], Cilk [7], and NESL [6].

There has also been a recent surge of activity on layering distributed and declarative programming language on top of distributed computation platforms. For example, Sawzall [32] is compiled to MapReduce applications, while Pig [31] programs are compiled to the Hadoop infrastructure. The MapReduce model is extended to support Joins in [12]. Other examples include Pipelets [9], HIVE (an internal Facebook language built on Hadoop), and Scope [10], Nebula [26], and PSQL (internal Microsoft languages built on Dryad).

Grid computing usually provides workflows (and not a programming language interface), which can be tied together by a user-level application. Examples include Swift [39] and its scripting language, Taverna [30], and Triana [36]. DryadLINQ is a higher-level language, which better conceals the underlying execution fabric.

7 Discussion and Conclusions

DryadLINQ has been in use by a small community of developers for over a year, resulting in tens of large applications and many more small programs. The system was recently released more widely within Microsoft and our experience with it is rapidly growing as a result. Feedback from users has generally been very positive. It is perhaps of particular interest that most of our users manage small private clusters of, at most, tens of computers, and still find substantial benefits from DryadLINQ.

Of course DryadLINQ is not appropriate for all distributed applications, and this lack of generality arises from design choices in both Dryad and LINQ.

The Dryad execution engine was engineered for batch applications on large datasets. There is an overhead of at least a few seconds when executing a DryadLINQ EPG which means that DryadLINQ would not currently be well suited to, for example, low-latency distributed database lookups. While one could imagine re-engineering Dryad to mitigate some of this latency, an effective solution would probably need to adopt different strategies for, at least, resource virtualization, fault-tolerance, and code generation and so would look quite different to our current system.

The question of which applications are suitable for parallelization by DryadLINQ is more subtle. In our experience, the main requirement is that the program can be written using LINQ constructs: users generally then find it straightforward to adapt it to distributed execution using DryadLINQ—and in fact frequently no adaptation is necessary. However, a certain change in outlook may be required to identify the data-parallel components of an algorithm and express them using LINQ operators. For example, the PageRank computation described in Section 5 uses a Join operation to implement a subroutine typically specified as matrix multiplication.

Dryad and DryadLINQ are also inherently specialized for streaming computations, and thus may appear very inefficient for algorithms which are naturally expressed using random-accesses. In fact for several workloads including breadth-first traversal of large graphs we have found DryadLINQ outperforms specialized random-access infrastructures. This is because the current performance characteristics of hard disk drives ensures that sequential streaming is faster than small random-access reads even when greater than 99% of the streamed data is discarded. Of course there will be other workloads where DryadLINQ is much less efficient, and as more storage moves from spinning disks to solid-state (e.g. flash memory) the advantages of streaming-only systems such as Dryad and MapReduce will diminish.

We have learned a lot from our users' experience of the `Apply` operator. Many DryadLINQ beginners find it easier to write custom code inside `Apply` than to determine the equivalent native LINQ expression. `Apply` is therefore helpful since it lowers the barrier to entry to use the system. However, the use of `Apply` "pollutes" the relational nature of LINQ and can reduce the system's ability to make high-level program transformations. This tradeoff between purity and ease of use is familiar in language design. As system builders we have found one of the most useful properties of `Apply` is that sophisticated programmers can use it to manually implement optimizations that DryadLINQ does not perform automatically. For example, the optimizer currently implements all reductions using partial sorts and groupings as shown in Figure 6. In some cases operations such as Count are much more efficiently implemented using hash tables and accumulators, and several developers have independently used `Apply` to achieve this performance improvement. Consequently we plan to add additional reduction patterns to the set of automatic DryadLINQ optimizations. This experience strengthens our belief that, at the current stage in the evolution of the system, it is best to give users flexibility and suffer the consequences when they use that flexibility unwisely.

DryadLINQ has benefited tremendously from the design choices of LINQ and Dryad. LINQ's extensibility, allowing the introduction of new execution implementations and custom operators, is the key that allows us to achieve deep integration of Dryad with LINQ-enabled programming languages. LINQ's strong static typing is extremely valuable when programming large-scale computations—it is much easier to debug compilation errors in Visual Studio than run-time errors in the cluster. Likewise, Dryad's flexible execution model is well suited to the static and dynamic optimizations we want to implement. We have not had to modify any part of Dryad to support DryadLINQ's development. In contrast, many of our optimizations would have been difficult to express using a more limited computational abstraction such as MapReduce.

Our current research focus is on gaining more understanding of what programs are easy or hard to write with DryadLINQ, and refining the optimizer to ensure it deals well with common cases. As discussed in Section 4.5, performance debugging is currently not well supported. We are working to improve the profiling and analysis tools that we supply to programmers, but we are ultimately more interested in improving the system's ability to get good performance automatically. We are also pursuing a variety of cluster-computing projects that are enabled by DryadLINQ, including storage research tailored to the workloads generated by DryadLINQ applications.

Our overall experience is that DryadLINQ, by combining the benefits of LINQ—a high-level language and rich data structures—with the power of Dryad's distributed execution model, proves to be an amazingly simple, useful and elegant programming environment.

Acknowledgements

We would like to thank our user community for their invaluable feedback, and particularly Frank McSherry for the implementation and evaluation of PageRank. We also thank Butler Lampson, Dave DeWitt, Chandu Thekkath, Andrew Birrell, Mike Schroeder, Moises Goldszmidt, and Kannan Achan, as well as the OSDI review committee and our shepherd Marvin Theimer, for many helpful comments and contributions to this paper.

References

[1] The DryadLINQ project. http://research.microsoft.com/research/sv/DryadLINQ/.

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

[3] Sort benchmark. http://research.microsoft.com/barc/SortBenchmark/.

[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND, G. P., AND WIL-

SON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2, 1995.

[5] BECK, M., DONGARRA, J., AND PLANK, J. S. NetSolve/D: A massively parallel grid execution system for scalable data intensive collaboration. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[6] BLELLOCH, G. E. Programming parallel algorithms. *Communications of the ACM (CACM) 39*, 3, 1996.

[7] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[8] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng. 2*, 1, 1990.

[9] CARNAHAN, J., AND DECOSTE, D. Pipelets: A framework for distributed computation. In *W4: Learning in Web Search*, 2005.

[10] CHAIKEN, R., JENKINS, B., LARSON, P.-Å., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *International Conference of Very Large Data Bases (VLDB)*, 2008.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. BigTable: A distributed storage system for structured data. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[12] CHIH YANG, H., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD international conference on Management of data*, 2007.

[13] CHU, L., TANG, H., YANG, T., AND SHEN, K. Optimizing data aggregation for cluster-based internet services. In *Symposium on Principles and practice of parallel programming (PPoPP)*, 2003.

[14] CRUANES, T., DAGEVILLE, B., AND GHOSH, B. Parallel SQL execution in Oracle 10g. In *ACM SIGMOD*, 2004.

[15] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[16] DESHPANDE, A., IVES, Z., AND RAMAN, V. Adaptive query processing. *Foundations and Trends in Databases 1*, 1, 2007.

[17] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., HSIAO, H., BRICKER, A., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2*, 1, 1990.

[18] DEWITT, D., AND GRAY, J. Parallel database systems: The future of high performance database processing. *Communications of the ACM 36*, 6, 1992.

[19] DUFFY, J. A query language for data parallel programming. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, 2007.

[20] ENGLERT, S., GLASSTONE, R., AND HASAN, W. Parallelism and its price : A case study of nonstop SQL/MP. In *Sigmod Record*, 1995.

[21] FENG, L., LU, H., TAY, Y. C., AND TUNG, A. K. H. Buffer management in distributed database systems: A data mining based approach. In *International Conference on Extending Database Technology*, 1998, H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds., vol. 1377 of *Lecture Notes in Computer Science*.

[22] GRAEFE, G. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD International Conference on Management of data*, 1990.

[23] GRAY, J., SZALAY, A., THAKAR, A., KUNSZT, P., STOUGHTON, C., SLUTZ, D., AND VANDENBERG, J. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, 2002.

[24] HASAN, W., FLORESCU, D., AND VALDURIEZ, P. Open issues in parallel query optimization. *SIGMOD Rec. 25*, 3, 1996.

[25] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2, 2007.

[26] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2007.

[27] KABRA, N., AND DEWITT, D. J. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD International Conference on Management of Data*, 1998.

[28] KOSSMANN, D. The state of the art in distributed query processing. *ACM Comput. Surv. 32*, 4, 2000.

[29] MORVAN, F., AND HAMEURLAIN, A. Dynamic memory allocation strategies for parallel query execution. In *Symposium on Applied computing (SAC)*, 2002.

[30] OINN, T., GREENWOOD, M., ADDIS, M., FERRIS, J., GLOVER, K., GOBLE, C., HULL, D., MARVIN, D., LI, P., LORD, P., POCOCK, M. R., SENGER, M., WIPAT, A., AND WROE, C. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18*, 10, 2005.

[31] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (Industrial Track) (SIGMOD)*, 2008.

[32] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4, 2005.

[33] SIMMA, A., GOLDSZMIDT, M., MACCORMICK, J., BARHAM, P., BLACK, R., ISAACS, R., AND MORTIER, R. CT-NOR: representing and reasoning about events in continuous time. In *International Conference on Uncertainty in Artificial Intelligence*, 2008.

[34] STONEBRAKER, M., BEAR, C., ÇETINTEMEL, U., CHERNIACK, M., GE, T., HACHEM, N., HARIZOPOULOS, S., LIFTER, J., ROGERS, J., AND ZDONIK, S. One size fits all? Part 2: Benchmarking results. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[35] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era (it's time for a complete rewrite). In *International Conference of Very Large Data Bases (VLDB)*, 2007.

[36] TAYLOR, I., SHIELDS, M., WANG, I., AND HARRISON, A. *Workflows for e-Science*. 2007, ch. The Triana Workflow Environment: Architecture and Applications, pp. 320–339.

[37] TRINDER, P., LOIDL, H.-W., AND POINTON, R. Parallel and distributed Haskells. *Journal of Functional Programming 12*, (4&5), 2002.

[38] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., CURREY, J., MCSHERRY, F., AND ACHAN, K. Some sample programs written in DryadLINQ. Tech. Rep. MSR-TR-2008-74, Microsoft Research, 2008.

[39] ZHAO, Y., HATEGAN, M., CLIFFORD, B., FOSTER, I., VON LASZEWSKI, G., NEFEDOVA, V., RAICU, I., STEF-PRAUN, T., AND WILDE, M. Swift: Fast, reliable, loosely coupled parallel computation. *IEEE Congress on Services*, 2007.