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

Michael Isard
Microsoft Research Silicon Valley
misard@microsoft.com

Yuan Yu
Microsoft Research Silicon Valley
yuanbyu@microsoft.com

ABSTRACT

The Dryad and DryadLINQ systems offer a new programming model for large scale data-parallel computing. They generalize previous execution environments such as SQL and MapReduce in three ways: by providing a general-purpose distributed execution engine for data-parallel applications; 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 operations 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 on a large compute cluster.

This paper describes the programming model, provides a high-level overview of the design and implementation of the Dryad and DryadLINQ systems, and discusses the tradeoffs and connections to parallel and distributed databases.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Performance

Keywords

Distributed programming, cloud computing, concurrency

1. INTRODUCTION

Large-scale data intensive computation has recently attracted a tremendous amount of attention both in the research community and in industry. A primary goal of our research is to give the programmer the illusion of writing for a single computer and to have the system deal automatically 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 engine, language constructs, parallel compilers, and development tools.

Parallel databases have for some time been able to store and process large volumes of data in scalable distributed systems, and meet the above goal of providing the programmer with a single-computer interface. They are however specialized to a subset of fairly simple computations for which SQL is an effective language. SQL does not make it easy to operate on complex types or data structures, and lacks good support for language constructs such as libraries, modules and loops that simplify the management and long-term maintainability of large software projects.

The application domain for data-intensive computation is moving away from traditional relational data-processing tasks towards areas such as graph analysis, image processing, machine learning, and a spectrum of eScience applications such as hydrology and astronomy. Consequently, recent systems have focused on integrating distributed execution engines with a standard high-level language such as C++, Java or C#. At the same time they have abandoned standard database features such as transactions and tables, which are less relevant to these newer applications.

MapReduce [7] implements a very restricted execution engine and programming model, but has proved influential due to its simplicity and scalability at low cost. An open-source implementation of MapReduce called Hadoop [3] has been widely adopted, and a language layer, PigLatin [14], was added to Hadoop to hide the complexity of programming the MapReduce interfaces directly. This paper outlines Dryad [13] and DryadLINQ [15]. Dryad is an execution engine that allows more general execution plans than MapReduce, and is therefore more efficient for some applications. DryadLINQ is a language layer that targets Dryad, but unlike PigLatin or SCOPE [5], which introduce new domain-specific languages, DryadLINQ is embedded as constructs within existing .NET programming languages.

Most parallel databases tightly integrate their storage, execution and language layers. Hadoop/PigLatin and Dryad/-

DryadLINQ both contain three distinct layers in their software stacks: a standalone distributed file system; an execution engine; and a language that targets the execution engine. This paper describes the abstractions we chose for Dryad and DryadLINQ, and Section 5 discusses some lessons learned from these choices.

The structure of this paper is as follows: Section 2 introduces the LINQ programming model. Section 3 provides a high-level overview of the Dryad execution engine and Section 4 describes the extensions that DryadLINQ adds to LINQ, and sketches some features of its implementation. Section 5 discusses the tradeoffs and connections to database systems, and outlines future research directions. Section 6 draws conclusions from the development of the systems.

2. PROGRAMMING WITH LINQ

We use LINQ [2] as the foundation for our programming model. LINQ adds high level data access to traditional .NET programming languages. It does so by defining a base type for a dataset, and then implementing a comprehensive set of relational operators including filtering (`Where`), equijoin (`Join`), and custom aggregation (`Aggregate`). These operators act on entire datasets and are typically used in place of loops over data items. As a simple example, consider the following code sequence to merge a set of partial sums and output their average:

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

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

Using LINQ constructs, this merge method might be replaced by the following:

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

which is both more concise and easier for an intelligent system to optimize. In this fragment, `x => x.sum` is an example of a C# *lambda expression.*

While the set of LINQ operators is reminiscent of those provided by SQL, the objects in a LINQ dataset may be of any .NET type including nested datasets and complex object graphs. In addition the operators are embedded directly in high level .NET languages, giving developers access to all the .NET libraries as well structuring concepts such as loops, classes, and modules. LINQ relies heavily on the use of lambda expressions and generics, and on static type inference, to ensure that the operators can be used effectively without imposing an unreasonable syntactic burden on the developer. It is this integration of operators and language syntax that makes LINQ a convenient and concise programming model.

The fact that the arguments to LINQ operators are expressions means that it supports higher-order functions. The fact that LINQ is embedded in .NET languages with full-fledged type systems means that it is straightforward to build up libraries of higher-order subcomputations. For example, MapReduce can be represented in (slightly abridged) LINQ syntax as

```
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);
}
```

where any functions with the appropriate type signatures can be passed in to perform the Map and Reduce operations.

The LINQ framework provides an interface that allows developers to introduce new implementations of the operators by subclassing the base dataset type. There is an implementation, PLINQ [10], that executes operators in parallel on a shared-memory multi-processor, and another, LINQ-to-SQL [2], that allows a .NET program to operate directly on records that are stored in a SQL database. DryadLINQ is implemented as another such "LINQ provider," so is directly compatible with existing code written to the LINQ programming model. A given method that calls into LINQ operators will be executed by different implementations depending on the concrete types of the datasets which are passed in as arguments, which greatly simplifies debugging since a distributed program can be tested locally without changing any of its code except the type of the input dataset.

A crucial aspect of the LINQ design is that implementations can support lazy evaluation. In this case, when an operator is applied to a dataset the runtime simply adds a node to an in-memory expression tree. Only when a dataset element is accessed, or an expression is explicitly materialized, does the implementation execute the desired computation in its entirety. This hybrid of imperative and declarative programming styles allows the system to perform powerful optimizations on complex expressions while retaining for the programmer the convenience and familiarity of imperative languages.

3. EXECUTION ENGINE

In this section we provide a high-level overview of the design and implementation of the Dryad execution engine. More details can be found in [13].

3.1 Computational model

A Dryad 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 Dryad runtime. In particular, there may be many more vertices in the graph than compute nodes in the cluster.

At run time, vertices are processes communicating with each other through the channels, and each channel is used

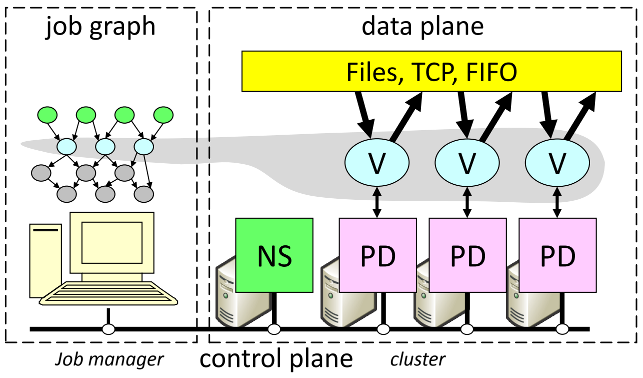

Figure 1: Dryad system architecture. NS is the name server which maintains the cluster membership. The job manager is responsible for spawning vertices (V) on available computers with the help of a remote-execution and monitoring daemon (PD). Vertices exchange data through files, TCP pipes, or shared-memory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

to transport a finite sequence of data records. The channel abstraction supports multiple implementations that use shared memory, TCP pipes, or disk files.

3.2 System architecture

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) determining constraints and hints to guide scheduling so that vertices execute on computers that are close to their input data in network topology; (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 Dryad job manager may contain its own internal scheduler that chooses which computer each vertex should be executed on, or it may send its list of ready vertices and their constraints to a centralized scheduler that optimizes placement across multiple jobs running concurrently. All channel data communication occurs directly between vertices and thus the job manager is only responsible for control decisions and is not a bottleneck for any data transfers.

A Dryad cluster has a name server (NS) that can be used to discover all the available compute nodes. The name server also exposes the location of each cluster machine within the network so that scheduling decisions can take better account of locality. There is a simple daemon (D) running on each cluster machine that is responsible for creating processes on behalf of the job manager. The first time a vertex (V) is executed on a machine its code is sent from the job manager to the daemon, or copied from a nearby computer that is executing the same job, and it is cached for subsequent uses. The daemon acts as a proxy so that the job manager can talk to the remote vertices and monitor the state and progress of the computation.

3.3 Runtime policies

Fault-tolerance is achieved by re-executing failed vertices. For example, if a vertex fails due to a read error on an input channel, the job manager can simply mark the upstream vertex that generated that version of the channel as failed. 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. Dryad guarantees that every successful execution of a job with immutable inputs and deterministic vertex programs will always return the same result, regardless of the number of machine or network failures over the course of the execution. The assumption that the dataflow graph is acyclic considerably simplifies the implementation, but could be removed if needed.

Dryad also supports a callback mechanism that can be used to implement runtime optimization policies by dynamically mutating the execution plan graph. In Dryad, each vertex belongs to a "stage" and each stage has a manager interface that receives a callback on every state transition in that stage. A variety of dynamic optimizations can be supported by implementing stage manager classes that export the callback interface. For example, it is often useful to choose the degree of parallelism of a stage of a Dryad job based on the amount of input data to be consumed by that stage. This data size may not be known before the job starts running, since it may be a complex function of the inputs. In this case Dryad can initially represent the whole stage by a single "proxy" vertex. As upstream computations complete and their output data sizes become known, the stage manager is informed of these sizes using callbacks. It can then rewrite the graph, replacing the initial proxy vertex by an appropriately-sized set of identical vertices to achieve the desired degree of parallelism.

3.4 Data model

The inputs to a Dryad job are typically stored as partitioned files in a distributed file system. Partitions are represented as source vertices in the job graph, and any processing vertex that is connected to such a vertex reads the entire partition, sequentially, through its input channel. The degree of parallelism of early stages of a Dryad job is therefore typically determined by the number of partitions in its inputs. A Dryad job stage may write a partitioned file to the distributed file system. Each vertex in the stage writes one or more partitions into the file system (by writing sequential data through a channel to a virtual sink vertex), and when the job completes successfully these partitions are concatenated atomically, in a metadata-only operation, by the file system to form a single distributed file.

Dryad communicates with distributed file systems through a narrow interface, and has been ported to several underlying storage implementations. The only operations it requires are: file lookup, to map from a name in the distributed file system to a set of partitions indicating the location and size of each partition; sequential read and write of an individual partition; and concatenation of partitions into a file. We have used Dryad with file systems that store partitions as flat files in the local NTFS file systems of the cluster computers and as tables in SQLServer databases running on the cluster computers. In the latter case Dryad can implement some of the functions of a non-transactional parallel database.

4. PROGRAMMING MODEL

While several language layers have been written to target the Dryad execution engine, we believe that DryadLINQ [15]

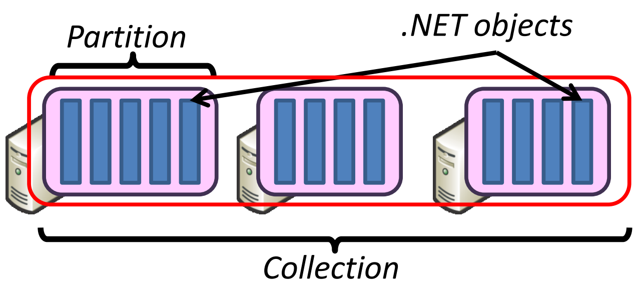

Figure 2: The DryadLINQ data model: strongly-typed collections of .NET objects partitioned on a set of computers.

is the most sophisticated and easiest to use. The most distinctive feature of DryadLINQ is its deep integration of data-parallel programming into traditional high-level programming languages. This section highlights some important aspects of DryadLINQ. More details on the programming model may be found in the LINQ language reference [2] and materials on the DryadLINQ project website [1]. A companion technical report [16] contains some working sample programs.

4.1 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. A DryadLINQ dataset is still a sequence of objects of an arbitrary .NET type, but it is in general distributed across the computers of a cluster, partitioned into disjoint pieces as shown in Figure 2. The partitioning strategies used — hash-partitioning, range-partitioning, and round-robin — are familiar from parallel databases [9]. The system transparently manages dataset partitioning unless the programmer explicitly overrides the optimizer's choices.

The inputs and outputs of a DryadLINQ computation refer to files in the distributed file systems supported by the underlying Dryad execution engine. Each partition in a distributed file contains a serialization of a subset of the objects in a DryadLINQ dataset, and objects cannot cross partition boundaries. Thus when a Dryad vertex reads a partition from the file system, the DryadLINQ process running in that vertex will operate on the corresponding subset of the overall dataset.

DryadLINQ stores additional information alongside each file, including schemas for the objects in the dataset and details of the partitioning scheme used, e.g. the key selector and range boundaries for a range-partitioned file. Thus DryadLINQ datasets may be self-describing, however this is not strictly enforced as it would be in a traditional database system.

DryadLINQ includes operators that materialize a dataset to a named file in the storage system, and it is possible to simultaneously invoke multiple expressions and generate multiple output tables 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.

4.2 DryadLINQ operators

DryadLINQ supports almost all of the LINQ operators. It also introduces a small set of custom operators specifically targeted to data-parallel programming. The new operators are integrated into the language using LINQ extensibility, so can be used in programs the same way as standard constructs. As far as possible we have avoided introducing DryadLINQ-specific extensions since we would like to maintain portability across LINQ implementations.

The first class of new operators allows the user to control the partitioning of a dataset, for example by supplying a custom hash function or range boundaries, or overriding the default number of partitions. It is natural that LINQ does not natively support partitioning operators because they are idempotent in the abstract LINQ programming model. Nevertheless, because the partitioning of a dataset directly affects the degree of parallelism of the distributed computation, allowing user control of partitions can be essential to get good performance in cases where the optimizer is unable to automatically determine an appropriate scheme. This may occur, for example, if the standard hash function induces a very skewed distribution. User-defined partitions can also be used to ensure that an output dataset is in the correct format, for example if it is going to be used by another system with its own partitioning scheme.

While the partitioning operators are currently DryadLINQ-specific, control over the granularity of partitioning can also be useful to improve the performance of shared-memory distributed implementations such as PLINQ. We are therefore working with other groups in Microsoft to define a consistent set of operators that may eventually be included in the LINQ standard.

The most significant other operator we introduce is `Apply`. It takes a function `f` and passes to it an iterator over an entire input dataset, allowing arbitrary streaming computations that sequentially process every object in the dataset. 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.

`Apply` can be thought of as an "escape-hatch" that a programmer can use when a computation cannot be expressed using any of LINQ's built-in operators, but it can incur a serious performance penalty since the whole dataset must be streamed to a single Dryad vertex if `f` is to process every element. However by adding a suitable annotation to `f` (see below), the user can instruct DryadLINQ to execute a separate vertex for each partition of the input dataset, so each instance of `f` "sees" the subset of elements of the set contained in a single partition. This partitioned form of `Apply` is frequently used to process input datasets that have complex custom formats, where for example each partition must be read in its entirety in order to parse the elements contained in it.

The last new operator is `Fork`, which 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. For example, a document parser could be implemented using `Fork` to output both plain text and a bibliographic entry to separate tables.

4.3 Annotations

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. Annotations in DryadLINQ are simple .NET attributes on classes and methods. For example annotations can be used to inform the system that a user-defined function is associative or commutative, and thus enable optimizations such as eager aggregation. The correctness of annotations is not enforced by the system, so developers are responsible for ensuring that they do not introduce semantic errors through faulty annotations.

4.4 Building on DryadLINQ

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

As noted above, the MapReduce programming model can be concisely stated as a static function. DryadLINQ includes a small library that defines a number of such functions. Applications are written as traditional programs calling into library functions, and make no explicit reference to the distributed nature of the computation. A general-purpose library for manipulating numerical data has also been written, and it has been used as a platform to implement machine-learning algorithms such as linear regression, Expectation–Maximization (E–M) for a mixture of gaussians, principal component analysis, probabilistic latent semantic indexing, and epitome extraction.

Several of those machine-learning 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 [6]. In DryadLINQ it is trivial to use a C# loop to express the iteration. Our users commonly execute loops containing tens of iterations, unfolding to hundreds of computation stages in a single distributed job. There is no current support for data-dependent loop termination, e.g. running a loop until convergence, so users typically write an outer while loop that repeatedly runs a fixed number of iterations and checks for the convergence condition. Each iteration of the outer loop invokes a new distributed computation. Dryad's graph-rewriting primitives would support dynamic unrolling at run time, and we intend to experiment with support for this to determine whether it is worth adding to the language.

4.5 DryadLINQ implementation

The DryadLINQ system consists of two main components: a parallel compiler and a runtime. The compiler compiles DryadLINQ programs to distributed execution plans and the runtime provides an implementation of the DryadLINQ operators. Figure 3 shows the flow of execution when a program is executed by DryadLINQ.

In Steps 1–2, a .NET user application runs. Because of LINQ's deferred execution, expressions are accumulated in a DryadLINQ expression object, and their actual execution is only triggered by the application invoking a method that materializes the output dataset. At this point, DryadLINQ takes over and compiles the LINQ expression into a distributed execution plan that Dryad understands (Step 3). This step performs most of the compilation work, including (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. DryadLINQ then invokes a custom, DryadLINQ-specific, Dryad job manager (Step 4).

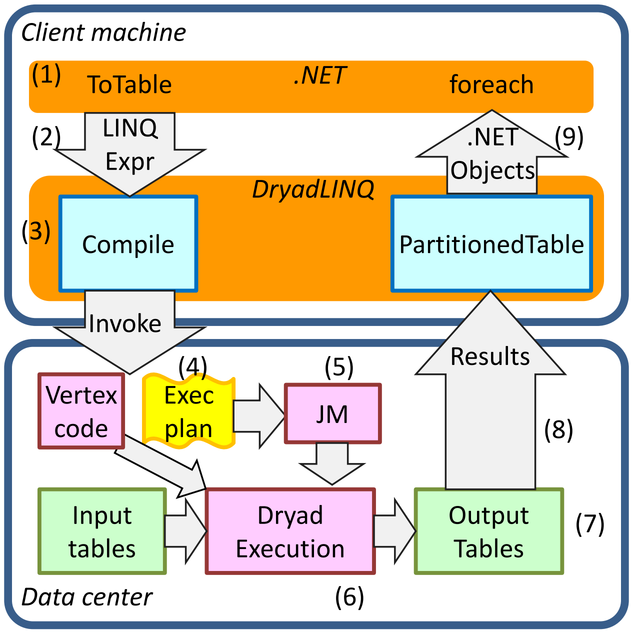

Figure 3: LINQ-expression execution in DryadLINQ.

Dryad takes over at Step 5. It creates the job graph using the plan created in Step 3, and schedules and spawns the vertices as cluster resources become available. Each Dryad vertex executes a vertex-specific program created by DryadLINQ in Step 3. When the Dryad job completes successfully, it writes the data to the output table(s), and returns the control back to DryadLINQ at Step 8.

DryadLINQ then creates the local dataset objects encapsulating the outputs of the execution, and returns control back to the user application. The output datasets may be used as inputs to subsequent expressions in the user program. Data objects within a dataset are fetched to the local context only if explicitly dereferenced.

At the heart of the DryadLINQ system is the parallel compiler. It first turns a raw LINQ expression into an execution plan graph (EPG), and goes through several phases of semantics-preserving graph rewriting to optimize the execution plan. The EPG is closely related to a traditional database query plan, but we use the more general form of execution plan to encompass computations that are not easily formulated as "query trees." The EPG is a directed acyclic graph, since the existence of common subexpressions and operators like `Fork` makes it difficult to express the execution plans as trees. The optimizer uses many traditional database optimization techniques, both static and dynamic. It annotates the EPG with metadata properties. For edges, these include the .NET type of the data, the partitioning scheme, and the compression scheme. For nodes, they include details of the operations to be performed. The properties are seeded from the LINQ expression tree and the input and output tables' metadata, and propagated and updated during EPG rewriting.

The EPG represents a "skeleton" of the Dryad data-flow graph to be executed, and each EPG node is expanded at run time into a set of Dryad vertices running the same computation on different partitions of a dataset. As mentioned in Section 3.3, the number of vertices in an EPG node may be determined dynamically based on the amount of data output by upstream computations. Other dynamic optimizations include the construction of network-aware aggregation and broadcast trees to reduce bandwidth. The topology of these trees is determined only after the vertices have been bound to specific computers for execution.

The DryadLINQ runtime is fairly simple, consisting mostly of: a) an implementation of the new DryadLINQ operators; and b) serialization and deserialization code for channel data. In many cases the vertex code generated by DryadLINQ is simply a LINQ query, in which case the runtime can take advantage of an existing LINQ implementation such as PLINQ to allow the vertex code to be automatically parallelized on a multi-core cluster computer.

4.6 Query Optimizations

The DryadLINQ optimizer is similar in many respects to classical database optimizers [12]. It has a static component, which performs conditional graph rewrite rules on the EPG, and a dynamic component, which uses Dryad's stage callbacks to optimize the execution at run time. Many of the core ideas employed by Dryad and DryadLINQ (such as shared-nothing architecture, horizontal data partitioning, dynamic repartitioning, parallel query evaluation, and dataflow scheduling), have their roots in parallel database systems [9], such as Gamma [8], Bubba [4], and Volcano [11].

There are also significant differences. First, our EPG is a directed acyclic graph, a more general representation than the query tree. This is needed in order to handle vertices with multiple outputs. Second, many optimizations rely on the semantics of the operators involved. However, inferring these semantics is substantially harder in the context of DryadLINQ than a tranditional database. The difficulties stem from the much richer data model and programming language.

5. DISCUSSION

Dryad has been in continuous use on small, medium and large clusters for nearly three years. Since we operate clusters and also program to Dryad's interfaces (for example to implement DryadLINQ, or to experiment with scheduling policies), we have extensive experience with its pros and cons. DryadLINQ has been available to developers for well over a year, but it is still too early to have definitive feedback from users. Early adopters are self-selecting since they seek out the system to solve particular problems that no existing system is adequately addressing. Also, once diverse groups start to program with DryadLINQ it becomes much harder to gather feedback. Users may simply abandon the system without telling us if they find that it doesn't meet their needs; or if our documentation is poor they may struggle with issues that are easily resolved. Nevertheless we have strongly positive initial impressions, as we discuss below.

5.1 The Dryad system

We believe that our decision to separate the execution engine into a standalone layer, independent of program semantics, has been very successful. The Dryad layer has only low-level information about the job that it is executing: it knows the graph topology and the quantity of data being transferred along each channel, and can be given simple hints about the amount of memory, CPU and disk bandwidth that each vertex is likely to consume. These hints have turned out to be enough to guide scheduling and placement decisions. By withholding from the Dryad layer any information about the semantics of the programs running at the vertices we have been able to keep the scheduling and fault-tolerance state machines uniform and thus fairly simple.

The contrast with MapReduce is instructive. MapReduce tightly couples the semantics of its processing vertices with the execution policy. There are exactly two types of process: Mappers and Reducers. Mappers read only from replicated storage which is assumed to be reliable. Reducers read from the output of the Mappers, so a failure in a Reducer may trigger the re-execution of a Mapper if the required data is no longer available—hence the fault tolerance strategy is different for the two types of vertex. The scheduling policy is also quite different, since each Mapper reads from a single replicated input file and has substantial benefit from being executed close to a replica, while each Reducer reads from all the Mappers. As a consequence of hard-wiring the semantics of its processes into the system, MapReduce can specialize its policies for those two cases, but specialization also makes it harder to implement more general graphs.

The lack of generality has at least two unfortunate side effects for MapReduce. First, some computations such as Join are painful to implement because each Mapper can only read a single input, while Join naturally consumes two streams in parallel. Second, MapReduce's specialization forces Reducers to write to replicated storage even when their output will immediately be consumed by a subsequent Map stage in a chained computation. This is necessary because Mappers rely for their fault tolerance on their inputs being stored reliably, while Dryad can backtrack through its execution graph to regenerate intermediate data as necessary. In the case of failure Dryad must do more work, but it incurs much less network and disk traffic in the common case since it does not need to replicate the output of every second stage of computation. In practice we find the probability of any given vertex encountering a failure is small, so optimizing for performance in the non-failure case has proved a successful strategy. Of course Dryad can if necessary bound the amount of backtracking that will be done in case of failure by replicating intermediate results at strategic points in the computation.

We have revisited a few aspects of the Dryad design since it was initially deployed. We originally supported multiple concurrent jobs by giving each job its own standalone scheduler, and allowing these schedulers to gain some visibility about the overall state of the system by inspecting the queues at the process daemons. If one job had scheduled a vertex on a particular computer, other jobs could discover this and might probe the cluster to look for an idle resource to use. When we eventually implemented a centralized scheduler we discovered that it was much more efficient to perform a global optimization across jobs, giving our system improved throughput and fairness. Fortunately, since the Dryad job manager was originally written with the intention of experimenting with different scheduling policies, all scheduling actions are accessed through a well-defined interface and moving to a central scheduler involved a fairly small amount of work.

We initially thought that Dryad jobs would commonly stream data through TCP-based channels for performance reasons. In practice, however, we have found that on medium and large clusters they are almost never used since it is simpler to always buffer the output of channels to disk. We have not yet implemented cut-through to allow a consumer vertex to start executing before the producer has completed, but this is an obvious optimization. We have recently gained several customers who are trying to implement numerically-intensive computations that require frequent communication between vertices, and we are revisiting our support for TCP channels to see if they are well suited to this type of workload.

The weakest part of the Dryad design is the support for dynamic optimization. As explained in Section 3.3, components such as DryadLINQ can modify the job graph during execution. This has turned out to be extremely powerful and useful, however it is very low level. In our current implementation the interaction between different optimizations (for example choosing the number of vertices in a processing stage at runtime, and building a dynamic aggregation tree) is ad hoc and leads to complex code that is unwieldy to maintain.

5.2 Data-parallel programming models

We have been very fortunate that LINQ maps so effectively to the requirements of a distributed data-parallel language. We were initially attracted to the idea of extending LINQ rather than designing a new domain-specific language partly because we believe that good language design is difficult, and best left to experienced professionals. The decision to use an existing language framework has been invaluable for us as system builders, since we do not need to waste engineering effort reinventing parsers or development environments, or worry about the complexity of integrating with the type system of an underlying language for stored procedures. We have also been able to leverage other LINQ implementations very effectively, for example by using PLINQ to execute our distributed computations efficiently and in parallel on multi-core computers. We believe that the integration of distributed computation inside existing languages has also been very beneficial for our users. Most developers using DryadLINQ are already familiar with C# and the Visual Studio development environment, and find it easy to get started using our system.

Most systems addressing the same application domain as DryadLINQ, including SQL databases, PigLatin [14], and SCOPE [5], take the approach of embedding user-defined functions within an outer query or script written in a simplified domain-specific language. This seems attractive for the case of simple computations where essentially all of the logic can be contained in the scripting language. Many databases have built-in procedural languages that integrate seamlessly with the SQL type system, and if programmers remain within the scope of these built-in languages, things stay simple. In the case that a program literally consists entirely of a single Map function followed by a single Reduce function, native MapReduce is very attractive.

The problem with a two-layer approach arises when the outer scripting language needs to call into user-defined procedures in a general-purpose language in order to perform more sophisticated processing. For example, databases often support stored procedures written in C or C++, PigLatin calls into Java methods, and SCOPE calls into C#. If a program is complex enough to require user-defined functions in a general-purpose language then we believe it is preferable to remain within the general-purpose language throughout. There is generally a mismatch between type systems at the boundary between a scripting language and a full programming language, which either requires marshaling or restricts the types that can be accessed in user-defined code. Also, if control begins in a scripting language then even simple tasks like reading initialization parameters from a configuration file and passing them to user-defined processing functions can become difficult to implement. Most importantly, though, complex programs benefit from unifying structure, and it is a mistake to force the user to use two different mechanisms to organize code and datastructures within a single application.

We believe that the approach adopted in DryadLINQ is simpler than previous approaches when applications involve even moderately complex user code. We have found that our users almost always write programs that involve more than a single Map function followed by a single Reduce, and usually the application logic cannot be expressed in a single SQL query. We have already seen users benefit from libraries of subroutines such as the linear algebra package mentioned in Section 4.4, and we anticipate that as we gain more eScience users, this type of library will become much more common. It is very useful to blur the boundary between the code that runs locally on the user's computer and that which is distributed, since as a program evolves subcomputations can migrate between the two. It is also very convenient to be able to seamlessly inspect the results of one distributed computation, manipulate them within a single language and type system, and pass them on to subsequent processing stages.

One potential benefit of a two-language approach to distributed computing is that it might make it easier to restrict the programs that users can write. It can be desirable, when many users share a large computing resource, to ensure that a poorly-written program cannot accidentally overwhelm the system. In practice this kind of enforcement is straightforward with DryadLINQ as well because the compiler generates the execution plan to send to Dryad, and the runtime acts as a sandbox for all user-supplied code.

5.3 Limitations of Dryad and DryadLINQ

Parallel databases offer features such as transactions that are not supported by systems using MapReduce or Dryad for their execution engine. Databases also offer much better support for very short queries, since both Dryad and MapReduce incur overhead of a second or more in distributing code and starting processes before computation can begin. Therefore, databases are a better solution for most transaction-processing workloads. However, it is worth noting that DryadLINQ can interoperate with SQL databases using the LINQ-to-SQL provider and in future we may be able to support a limited notion of transactional update.

DryadLINQ does not currently support all state of the art database optimization techniques. This is partly because, as mentioned in Section 4.6, it can be difficult to automatically infer properties of user-defined operators. We are pursuing ways to perform more sophisticated inference, for example using static program analysis techniques such as program slicing. We also currently perform only rudimentary cost-based optimizations, and are working to improve this area of the system.

6. CONCLUSIONS

The ultimate goal of our research is to make data-parallel cluster computing as easy as writing programs for a single computer. We think Dryad and DryadLINQ represent a significant step towards this goal. In Dryad, we have built a general-purpose, high performance distributed execution engine that is considerably more expressive and flexible than previous execution engines. In DryadLINQ, we have achieved seamless integration of data-parallel computing into traditional high-level programming languages. We were able to achieve deep language integration mainly due to the extensibility of the LINQ framework and the flexibility of the Dryad execution engine. 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 bugs in the cloud.

Dryad has been in continuous operation for several years on production clusters made up of thousands of computers. DryadLINQ has been in use by a small community of developers for over a year. The systems have been used on a wide variety of applications, including relational queries, large-scale log mining, web-graph analysis, and machine learning. The systems were recently released for academic use to a number of universities, and the Microsoft External Research team has been preparing for a formal release that includes comprehensive programming manuals and tutorials. To facilitate collaborations, the source code of DryadLINQ is included in our academic release.

Our experience so far is that DryadLINQ, by combining the benefits of LINQ with the efficient and scalable Dryad execution engine, has proved to be an amazingly simple, useful and elegant programming environment.

7. ACKNOWLEDGMENTS

We would like to thank the members of the Dryad and DryadLINQ projects for their very substantial contributions to this work, and acknowledge the numerous helpful suggestions and comments of our colleagues throughout Microsoft.

8. REFERENCES

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

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

[3] Hadoop wiki. http://wiki.apache.org/hadoop/, April 2008.

[4] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, 1990.

[5] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *International Conference of Very Large Data Bases (VLDB)*, August 2008.

[6] T. Cruanes, B. Dageville, and B. Ghosh. Parallel SQL execution in Oracle 10g. In *ACM SIGMOD*, pages 850–854, Paris, France, 2004. ACM.

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

[8] 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.

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

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

[11] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD International Conference on Management of data*, pages 102–111, New York, NY, USA, 1990. ACM Press.

[12] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)*, pages 59–72, March 2007.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (Industrial Track) (SIGMOD)*, Vancouver, Canada, June 2008.

[15] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, December 8-10 2008.

[16] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, J. Currey, F. McSherry, and K. Achan. Some sample programs written in DryadLINQ. Technical Report MSR-TR-2008-74, Microsoft Research, May 2008.