Софийски университет

“Св. Климент Охридски”

Факултет по математика и

информатика

****

Проект по

Облачни приложения и архитектури

на тема

DryadLINQ

 Изготвил:

София 2011 Милен Борисов Станоев фн: 23483

 Специалност: Изкуствен интелект

Overview

DryadLINQ is a simple, powerful, and elegant programming environment for writing large-scale data parallel applications running on large PC clusters.

If you are interested in acquiring the DryadLINQ source for research purposes, please contact one of the project members below!

The goal of DryadLINQ is to make distributed computing on large compute cluster simple enough for every programmer. DryadLINQ combines two important pieces of Microsoft technology: the [Dryad](http://research.microsoft.com/en-us/projects/dryad/default.aspx)distributed execution engine and the .NET Language Integrated Query (LINQ).

Dryad provides reliable, distributed computing on thousands of servers for large-scale data parallel applications. LINQ enables developers to write and debug their applications in a SQL-like query language, relying on the entire .NET library and using Visual Studio.

Dryad

The Dryad Project is investigating programming models for writing parallel and distributed programs to scale from a small cluster to a large data-center.

If you are interested in acquiring the DryadLINQ source for research purposes, please contact one of the project members below!

Dryad is an infrastructure which allows a programmer to use the resources of a computer cluster or a data center for running data-parallel programs. A Dryad programmer can use thousands of machines, each of them with multiple processors or cores, without knowing anything about concurrent programming.

The Structure of Dryad Jobs



A Dryad programmer writes several sequential programs and connects them using one-way channels. The computation is structured as a directed graph: programs are graph vertices, while the channels are graph edges. A Dryad job is a graph generator which can synthesize any directed acyclic graph. These graphs can even change during execution, in response to important events in the computation.

Dryad is quite expressive. It completely subsumes other computation frameworks, such as Google's map-reduce, or the relational algebra. Moreover, Dryad handles job creation and management, resource management, job monitoring and visualization, fault tolerance, re-execution, scheduling, and accounting.

The Dryad Software Stack

As a proof of Dryad's versatility, a rich software ecosystem has been built on top Dryad:

* [SSIS](http://www.microsoft.com/technet/prodtechnol/sql/2005/intro2is.mspx) on Dryad executes many instances of SQL server, each in a separate Dryad vertex, taking advantage of Dryad's fault tolerance and scheduling. This system is currently deployed in a live production system as part of one of [Microsoft's AdCenter](http://advertising.microsoft.com/microsoft-adcenter) log processing pipelines.
* [DryadLINQ](http://research.microsoft.com/en-us/projects/dryadlinq/default.aspx) generates Dryad computations from the [LINQ](http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx) Language-Integrated Query extensions to C#.
* The distributed shell is a generalization of the pipe concept from the Unix shell in three ways. If Unix pipes allow the construction of one-dimensional (1-D) process structures, the distributed shell allows the programmer to build 2-D structures in a scripting language. The distributed shell generalizes Unix pipes in three ways:
	1. It allows processes to easily connect multiple file descriptors of each process -- hence the 2-D aspect.
	2. It allows the construction of pipes spanning multiple machines, across a cluster.
	3. It *virtualizes* the pipelines, allowing the execution of pipelines with many more processes than available machines, by time-multiplexing processors and buffering results.
* Several languages are compiled to distributed shell processes. PSQL is an early version, recently replaced with Scope.



.NET Language-Integrated Query

After two decades, the industry has reached a stable point in the evolution of object-oriented (**OO**) programming technologies. Programmers now take for granted features like classes, objects, and methods. In looking at the current and next generation of technologies, it has become apparent that the next big challenge in programming technology is to reduce the complexity of accessing and integrating information that is not natively defined using OO technology. The two most common sources of non-OO information are relational databases and XML.

Rather than add relational or XML-specific features to our programming languages and runtime, with the LINQ project we have taken a more general approach and are adding general-purpose query facilities to the .NET Framework that apply to all sources of information, not just relational or XML data. This facility is called .NET Language-Integrated Query (LINQ).

We use the term *language-integrated query* to indicate that query is an integrated feature of the developer's primary programming languages (for example, Visual C#, Visual Basic). Language-integrated query allows *query expressions* to benefit from the rich metadata, compile-time syntax checking, static typing and IntelliSense that was previously available only to imperative code. Language-integrated query also allows a single general purpose declarative query facility to be applied to all in-memory information, not just information from external sources.

.NET Language-Integrated Query defines a set of general purpose *standard query operators* that allow traversal, filter, and projection operations to be expressed in a direct yet declarative way in any .NET-based programming language. The standard query operators allow queries to be applied to any **IEnumerable<T>**-based information source. LINQ allows third parties to augment the set of standard query operators with new domain-specific operators that are appropriate for the target domain or technology. More importantly, third parties are also free to replace the standard query operators with their own implementations that provide additional services such as remote evaluation, query translation, optimization, and so on. By adhering to the conventions of the *LINQ pattern*, such implementations enjoy the same language integration and tool support as the standard query operators.

The extensibility of the query architecture is used in the LINQ project itself to provide implementations that work over both XML and SQL data. The query operators over XML (LINQ to XML) use an efficient, easy-to-use, in-memory XML facility to provide XPath/XQuery functionality in the host programming language. The query operators over relational data (LINQ to SQL) build on the integration of SQL-based schema definitions into the common language runtime (CLR) type system. This integration provides strong typing over relational data while retaining the expressive power of the relational model and the performance of query evaluation directly in the underlying store.

DryadLINQ

DryadLINQ translates LINQ programs into distributed Dryad computations:

* C# and LINQ data objects become distributed partitioned files.
* LINQ queries become distributed Dryad jobs.
* C# methods become code running on the vertices of a Dryad job.



DryadLINQ has the following features:

* **Declarative programming**: computations are expressed in a high-level language similar to SQL
* **Automatic parallelization**: from sequential declarative code the DryadLINQ compiler generates highly parallel query plans spanning large computer clusters. For exploiting multi-core parallelism on each machine DryadLINQ relies on the[PLINQ](http://msdn2.microsoft.com/en-us/magazine/cc163329.aspx) parallelization framework.
* **Integration with Visual Studio**: programmers in DryadLINQ take advantage of the comprehensive VS set of tools: Intellisense, code refactoring, integrated debugging, build, source code management.
* **Integration with .Net**: all .Net libraries, including Visual Basic, and dynamic languages are available.
* **Type safety**: distributed computations are statically type-checked.
* **Automatic serialization**: data transport mechanisms automatically handle all .Net object types.
* **Job graph optimizations**
	+ static: a rich set of term-rewriting query optimization rules is applied to the query plan, optimizing locality and improving performance.
	+ dynamic: run-time query plan optimizations automatically adapt the plan taking into account the statistics of the data set processed.
* **Conciseness**: the following line of code is a complete implementation of the Map-Reduce computation framework in DryadLINQ:

public static IQueryable<R>
  MapReduce<S,M,K,R>(this IQueryable<S> source,
                                 Expression<Func<S,IEnumerable<M>>> mapper,
                                 Expression<Func<M,K>> keySelector,
                                 Expression<Func<K,IEnumerable<M>,R>> reducer)
{
    return source.SelectMany(mapper).GroupBy(keySelector, reducer);
}

System Architecture

DryadLINQ compiles LINQ programs into distributed computations running on the Dryad cluster-computing infrastructure.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. 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 faulttolerance by re-executing failed or slow processes; (4) monitoring the job and collecting statistics; and (5) transforming the job graph dynamically according to usersupplied 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

DryadLINQ Execution Overview



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, DryadLINQspeciﬁc, 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.

Watching the Remote Program

While your program is running the remote query you will see on the output console some cryptic strings. This information comes from the Dryad Job Manager, which oversees the execution of the program on the cluster. While an explanation of Dryad is out of the scope of this document it is useful to know the basics of Dryad for debugging, visualization and performance tuning purposes.

Figure 2: Execution stages of a Dryad Job.

Figure 2 shows the stages of a Dryad job. The job is coordinated by a Job Manager; the manager is the *brain,* while all the work is performed by theworkers*,* which arecalled *vertices*. The manager starts first, and it creates vertices on the cluster, using a remote execution service. It monitors the vertices’ progress and gathers execution statistics. The manager prints periodic summaries of the state of the computation, to the console.

The job manager itself may be running on a remote machine (depending on configuration parameters).

* If your job manager is running on the local machine then you can visualize your job’s state interactively using internet explorer.
* If your job manager is running on the cluster using the cluster scheduler, then you can visualize the job via the cluster’s web server. The visualization is not interactive.

By default, DryadLINQ will run the job manager locally. If you are running on a cluster that has a job scheduler installed, you can configure DryadLINQ to instead submit the job to the job scheduler by adding the usejobscheduler attribute to the <Cluster> element in your DryadLinqConfig.xml:

 <Cluster name=" MyClusterName "
 …

 usejobscheduler=" true " />

Partitioned Files

The Match program that we wrote can be effectively parallelized for scanning a large amount of data. It is sufficient to cut the data into pieces (preserving line boundaries) and run the scan in parallel on all pieces. The hardest part to do is to describe the file pieces. For this purpose **DryadLINQ** provides a datatype PartitionedFile. A partitioned file on disk is composed of two parts:

1. The pieces themselves and
2. The metadata: a textual description of all the pieces of a file which has been split. Figure 3 shows how the metadata is organized:
* The first line indicates the name prefix of each piece. The pieces *must* all be placed in the same directory on all the machines. In this example each file will be in the \mydata directory, and its name will have the form Piece.XXXXXXXX. Here XXXXXXXX is an 8-digit hexadecimal number.
* The second line is the number of pieces, in this example 4.
* Each line that follows describes a piece:
	+ The piece number, in decimal.
	+ The piece size in bytes.
	+ Finally, a comma-separated list of machines. A piece may be replicated on several machines, for fault-tolerance.



Figure 3: Partitioned File Structure

The description in Figure 3 corresponds to the following pieces:

* \\m1\mydata\Piece.00000000
* \\m2\mydata\Piece.00000001
* \\m3\mydata\Piece.00000001
* \\m3\mydata\Piece.00000002
* \\m4\mydata\Piece.00000003

Piece.00000001 is present on two machines.

Once you have partitioned your data in this way, you only need to make a tiny change to enable your computation to use the partitioned table:

public static IQueryable<string> Match(string directory,

 string filename,

 string tosearch)

{

DryadDataContext ddc = new DryadDataContext("file://" + directory);

DryadTable<LineRecord> table = ddc.GetPartitionedTable<LineRecord>(filename);

return table.Select(s => s.line).Where(s => s.IndexOf(tosearch) >= 0);

}

When running this job, the job will operate in parallel on all four partitions:



Figure 4: The program operating on a partitioned file with 4 partitions.

The throughput of this computation will be increased by a factor of 4 (assuming the cluster contains at least 4 machines). If the input partitions are on different machines, they can be read all in parallel. The file output by the program also contains four partitions. (However, the C# program still uses one iterator to read all four output partitions.)

Reductions (Aggregations)

One of the most useful operations that can be performed on data is *reduction*, also called *aggregation*. By definition, an aggregation takes a lot of data values and collapses them to a single value. A typical example would be the sum of a stream of numbers. A somewhat less obvious example is the count.

**LINQ** contains lots of operators – see 4.1. But, as you may expect by now, **LINQ** provides a generic aggregation operator which relies on a delegate to transform two values into one.

public static TAccumulate Aggregate<TSource, TAccumulate>(

 this IQueryable<TSource> source,

 TAccumulate seed,

 Expression<Func<TAccumulate, TSource, TAccumulate>> func)

We can sum up the values in a partitioned file in this way:

var result = input.Aggregate((x,y) => x+y);

The distributed computation which aggregates an input with two partitions looks as follows:

Read input

Add

Figure 6: Aggregation is done after collecting all inputs in a single vertex.

The *Aggregate* vertex collects all the data from the two input readers and sums it up.

However, if the aggregating function is associative, a much better parallel computation plan is possible by aggregating each partition of the data separately and then combining the results at the end. By adding an [Associative] annotation to a function, you can enable **DryadLINQ** to generate a much better plan.

[Associative]

int Add(int x, int y);

var sum = input.Aggregate((x,y)=>Add(x,y));

The generated plan looks much better:

Read & Add

Add

Figure 7: Aggregation of associative function is done using multiple machines.

Each machine does pipelined reading followed by local aggregation on its own data, and then a global stage combines the partial results.

Apply

**DryadLINQ** takes one IQueryable object and transforms it into a distributed network of processes (vertices). Each of the vertices manipulates only a partition of the data. In the generated code, each vertex executes an independent **LINQ** program. ***The inputs and outputs to each vertex are all IEnumerable objects***. Thus each vertex takes automatically advantage of the lazy evaluation and pipelining provided by the iterator model.

Most **LINQ** methods are “stateless”: they operate on each value in a collection independently on its neighbors. However, some very useful types of computations need to see multiple values at once. A typical example is a sliding-window (e.g., convolution) computation. **DryadLINQ** extends **LINQ** with several powerful operators.The complete list of **DryadLINQ** operators is given in Section 4.1 DryadLINQ Operators.

The Apply operator is a new addition. It corresponds roughly to Select: it has a delegate argument, which produces the output by transforming the input. Unlike Select, the input to the Apply delegate is the whole input stream, and the output is *a complete stream*.



Figure 8: The Select delegate receives each element individually, while the one of Apply receives the whole stream.

In other words, in Figure 8: The Select delegate receives each element individually, while the one of Apply receives the whole stream.Figure 8 the type of f is Expression<Func<T,S>>, while the type of g is Expression<Func<IEnumerable<T>,IEnumerable<S>>>.

There exists a binary version of Apply, which operates on two input streams:

public static IQueryable<T3>
Apply<T1, T2, T3>(this IQueryable<T1> source1,
 IQueryable<T2> source2,
 Expression<Func<IEnumerable<T1>,
 IEnumerable<T2>,
 IEnumerable<T3>>> procFunc);

Unfortunately, there is no binary version of Select. But we can build one using on Apply. For example, here is how to implement a binary Select-like operator which adds the corresponding numbers in two streams (the two streams must have the same length). First, we write the per-vertex transformation, which operates on IEnumerable inputs:

public static IEnumerable<int>

addeach(IEnumerable<int> left, IEnumerable<int> right)

{

 IEnumerator<int> left\_enu = left.GetEnumerator();

 IEnumerator<int> right\_enu = right.GetEnumerator();

 while (true)

 {

 bool more\_left = left\_enu.MoveNext();

 bool more\_right = right\_enu.MoveNext();

 if (more\_left != more\_right)

 {

 throw new Exception("Streams with different lengths");

 }

 if (!more\_left) yield break; // both are finished

 int l = left\_enu.Current;

 int r = right\_enu.Current;

 int q = l + r;

 yield return q;

 }

}

The addeach function is hopefully obvious: it iterates over two streams in parallel using two iterators (it uses the MoveNext() and Current stream operators rather than foreach).

To create the IQueryable version of addition it is just enough to invoke addeach on the two inputs:

public static IQueryable<int>
Add(IQueryable<int> left,
 IQueryable<int> right)

{

 return left.Apply<int, int, int>(right, (x,y) => addeach(x,y));

}

(It is surprisingly harder to write a generic Select, which takes an arbitrary delegate; this is covered in Section Advanced Topic: Higher-Order Query Operations.)

If we run this query, for example by supplying both inputs from a single partitioned file:

Add(input, input).ShowOnConsole();

we will have an unpleasant surprise: the executed plan is quite inefficient:



Figure 9: Naive Plan for the pairwise addition.

(Each in[] vertex reads one partition and then broadcasts the data to two consumers using a Tee vertex. A Tee vertex stands for “broadcast”: all outputs correspond to the same input. The red edges in the figure are implemented using FIFO channels; this means that all three vertices *Merge* and *Apply* run as separate threads in a single process on the same machine, and just pass pointers to objects to each other.)

The plan generated performs all the additions in a single vertex, labeled “Apply\_\_0” in the figure. For this purposes it merges (by concatenating) the two entire input streams. It is obvious to us that the additions could be done in parallel in each partition, but since the addeach function claims it needs to see the entire input stream, **DryadLINQ** obliges and builds it before passing it to the Apply.

Fortunately, there is a way out: by adding an appropriate *annotation* to the addeach function you can indicate that it can be safely applied to each partition independently:

[Homomorphic]

public static IEnumerable<int>

addeach(IEnumerable<int> left, IEnumerable<int> right)

While homomorphic is a mouthful, it just means that the function operates correctly partitionwise (i.e., it distributes with respect to partition concatenation:
concatenate(add(a,c),add(b,d)) = add(concatenate(a,b), concatenate(c,d) ). With this annotation, the execution plan uses two separate vertices to perform the addition, one operating on each partition:



Figure 10: using delegate annotations can improve plans.

MapReduce

Dryad and DryadLINQ were designed to address some of the limitations of databases and MapReduce. Dryad is a distributed execution engine that lies between databases and MapReduce: it abandons much of the traditional functionality of a database (transactions, in-place updates, etc.) while providing fault-tolerant execution of complex query plans on large-scale clusters. DryadLINQ is a language layer built on top of Dryad that tightly integrates distributed queries into high level .NET programming languages. It provides a unied data model and programming language for relational queries and user-dened functions. Dryad and DryadLINQ are an attractive research platform because Dryad supports complex execution plans that cannot be performed by a system such as Hadoop, while the Dryad- LINQ source is available for modication, unlike that of most parallel databases. This paper explains in detail how distributed aggregation can be treated eciently by the Dryad-LINQ optimization phase, and extends the DadLINQ programming interface as well as the set of optimizations the system may apply.

Often the partial aggregation of a subsequence r is much

smaller than r itself: in the case of average for example

the partial sum is just two values, regardless of the number

of integers that have been processed. When there is such

substantial data reduction, partial aggregation can be introduced

both as part of the initial Map phase and in an aggregation

tree, as shown in Figure 2, to greatly reduce network

tra\_c. In order to decompose a user-de\_ned aggregation using

partial aggregation it is necessary to introduce auxiliary

functions, called \Combiners" in [10], that synthesize the intermediate

results into the \_nal output. The MapReduce

system described in [10] can perform partial aggregation on

each local computer before transmitting data across the network,

but does not use an aggregation tree.

In order to enable partial aggregation a user of MapReduce

must supply three functions:

1. InitialReduce: hK; Sequence of Ri ! hK;Xi which

takes a sequence of records of type R, all with the

same key of type K, and outputs a partial aggregation

encoded as the key of type K and an intermediate type

X.

2. Combine: hK; Sequence of Xi ! hK;Xi which takes

a sequence of partial aggregations of type X, all with

the same key of type K, and outputs a new, combined,

partial aggregation again encoded as an object of type

X with the shared key of type K.

3. FinalReduce: hK; Sequence of Xi ! Sequence of S

which takes a sequence of partial aggregations of type

X, all with the same key of type K, and outputs zero

or more records of type S.

In simple cases such as Sum or Min the types R, X and S are

all the same, and InitialReduce, Combine and FinalReduce

can all be computed using the same function. Three separate

functions are needed even for straightforward computations

such as integer average:

Partial InitialReduce(Key k,

Sequence<int> recordSequence) {

Partial p = { 0, 0 };

foreach (r in recordSequence) {

p.partialSum += r;

++p.partialCount;

}

return <k, p>;

}

Partial Combine(Key k,

Sequence<Partial> partialSequence) {

Partial p = { 0, 0 };

foreach (r in partialSequence) {

p.partialSum += r.partialSum;

p.partialCount += r.partialCount;

}

return <k, p>;

}

double FinalReduce(Key k,

Sequence<Partial> partialSequence)

{

// key is ignored

Partial p = Combine(k, partialSequence);

return (double)p.partialSum/(double)p.partialCount;

}

PageRank

The \_nal example performs an iterative PageRank computation

on a web graph. For clarity we present a simpli\_ed

implementation of PageRank but interested readers can \_nd

more highly optimized implementations in [27] and [28].

var ranks = pages.Select(p => new Rank(p.name, 1.0));

for (int i = 0; i < interations; i++)

{

// join pages with ranks, and disperse updates

var updates =

from p in pages

join rank in ranks on p.name equals rank.name

select p.Distribute(rank);

// re-accumulate.

ranks = from list in updates

from rank in list

group rank.rank by rank.name into g

select new Rank(g.Key, g.Sum());

}

Each element p of the collection pages contains a unique

identi\_er p.name and a list of identi\_ers specifying all the

pages in the graph that p links to. Elements of ranks are

pairs specifying the identi\_er of a page and its current estimated

rank. The \_rst statement initializes ranks with a

default rank for every page in pages. Each iteration then

calls a method on the page object p to distribute p's current

rank evenly along its outgoing edges: Distribute returns a

list of destination page identi\_ers each with their share of p's

rank. Finally the iteration collects these distributed ranks,

accumulates the incoming total for each page, and generates

a new estimated rank value for that page. One iteration is

analogous to a step of MapReduce in which the \Map" is actually

a Join pipelined with the distribution of scores, and

the \Reduce" is used to re-aggregate the scores. The \_nal

select is associative-decomposable so once more DryadLINQ

uses the optimized execution plan in Figure 2.

The collection pages has been pre-partitioned according

to a hash of p.name, and the initialization of ranks causes

that collection to inherit the same partitioning. Figure 7

shows the execution plan for multiple iterations of Page-

Rank. Each iteration computes a new value for ranks. Because

DryadLINQ knows that ranks and pages have the

same partitioning, the Join in the next iteration can be computed

on the partitions of pages and ranks pairwise without

any data re-partitioning. A well-designed parallel database

would also be able to automatically select a plan that avoids

re-partitioning the datasets across iterations. However, because

MapReduce does not natively support multi-input operators

such as Join, it is unable to perform a pipelined iterative

computation such as PageRank that preserves data

locality, leading to much larger data transfer volumes for

this type of computation when executed on a system such

as Hadoop.

Development

After nearly six years of research into Dryad and DryadLINQ—as well as its use in-house on Microsoft projects such as [Kinect](http://www.xbox.com/en-US/kinect) and [Bing](http://www.bing.com/)—Dryad and DryadLINQ are entering commercial use. Starting Jan. 26, a technology preview of Dryad and DryadLINQ will be built into the [Windows HPC Server 2008 R2](http://www.microsoft.com/hpc/en/us/default.aspx) high-performance computing line and eventually will be integrated with [Microsoft SQL Server](http://www.microsoft.com/sqlserver/en/us/default.aspx) and[Windows Azure](http://www.microsoft.com/windowsazure/). HPC Server is designed to give customers tremendous computing power and an easy management experience, all using off-the-shelf hardware.

[Michael Isard](http://research.microsoft.com/en-us/people/misard/), a [Microsoft Research Silicon Valley](http://research.microsoft.com/en-us/labs/siliconvalley/default.aspx) principal researcher instrumental in launching the Dryad project, says the new technology is an excellent example of how Microsoft views computing.

“This is an opportunity to democratize large-scale, data-intensive computing,” he says. “In areas such as customer-relationship management, business intelligence, planning, and infrastructure—all those tasks where companies now have access to a vast amount of data—Dryad and DryadLINQ can make sense of that data.”

How Dryad Works

The Dryad project consists of two key components. The Dryad tool itself provides reliable computing across thousands of servers. DryadLINQ, built on Microsoft’s[.NET Language Integrated Query](http://msdn.microsoft.com/en-us/library/bb308959.aspx) (LINQ), enables developers to write their applications in a SQL-like query language, using familiar programming tools such as[Microsoft Visual Studio](http://www.microsoft.com/visualstudio/en-us/). Most programmers will work only with DryadLINQ; once they have launched their application into the cloud, Dryad will do the rest, invisibly.

A third piece, the [Distributed Storage Catalog](http://research.microsoft.com/en-us/projects/tidyfs/) (DSC), is a distributed file system built for Dryad. It manages the data that Dryad is processing, keeping it stored reliably and safely with user-configurable redundancy. The DSC also keeps the data close to the servers processing it, so time is not wasted transmitting the data to a server.

Dryad and DryadLINQ make it easier for programmers to take advantage of the power of parallel computing, in which rows of servers or multicore processors within a single machine tackle a single computing problem. Such computing is extremely powerful, especially with so-called “unstructured” data such as information on buying habits that a retailer might collect from tens of thousands of customers but that has not been tagged or annotated, in contrast to structured data found, for instance, in a SQL database.

It is difficult, though, to harness the power afforded by parallel computing. Most programmers are more familiar with writing sequential programs, in which Action A is followed by Action B, then Action C. It is challenging to think and program in parallel.

While DryadLINQ enables developers to write their applications in a query language using Visual Studio, Dryad breaks up the program and assigns it across clusters of servers or processors. In effect, Dryad acts as a computing traffic cop, sending data down potentially millions of computing pathways. It helps make sure that when one piece of data is modified, other servers don’t also change that data. It balances the computing load between many computers, and it re-routes computing traffic if an error or communications problem temporarily takes one or even several servers offline.

That removes a huge burden from programmers and lets them focus on the problem they are trying to solve, not how the computers will act in parallel.

“We want programmers to be able to write their programs without having to think about things like fault tolerance [a byproduct of parallel computing’s complexity],” says [Yuan Yu](http://research.microsoft.com/en-us/people/yuanbyu/), a principal researcher at Microsoft Research Silicon Valley who led the creation of the DryadLINQ component.

“We want them to be able to write sequential and declarative code, and then, that same code can be run on a single machine, on a multicore machine, or on a cluster of machines. That’s the beauty of the DryadLINQ programming model.”

A second benefit is that Dryad gives programmers supercomputer-level power with everyday programming tools and relatively inexpensive hardware.

“This is a much cheaper way of doing things,” Yu says. “Everything is a commodity—a commodity operating system, using commodity servers and switches. Dryad deals with the reliability and the bandwidth issues.”

Dryad also utilizes Microsoft’s big investment in the cloud. As Dryad is integrated with Azure, all a programmer will need to take advantage of Dryad is a client and an Azure connection. Whether they are working on a cluster or the cloud, programmers can store their data and then manipulate it through their DryadLINQ-written applications. On a cluster, the DSC unit manages the data to keep it close to the processors working on it, so time is not lost in communicating data between servers.

“The only thing we’ll give the customer is some client software for writing DryadLINQ programs,” Isard says. “They’ll basically write the program on their machine and submit it to Windows Azure, where Dryad is running internally.”

The Evolution of Dryad

Dryad had its roots in an idea developed in October 2004 by Isard—then working on search for Microsoft—when he recognized the need for a large-scale data-intensive computation platform and began discussions with researchers at Microsoft to build on the idea.

Not long afterward, the newly created Dryad came into widespread use within Microsoft’s search offering, where it was used on thousands of servers. But while the tool worked well, the programming interface was awkward. Yu recognized the potential of LINQ to serve as the front-end programming tool for Dryad, and started the DryadLINQ project in September 2006. By early 2008, the Dryad/DryadLINQ combination was made available within Microsoft. A release to a small collection of academic researchers followed. Dryad also was adopted as a key tool for the development of the [Xbox 360](http://www.xbox.com/en-US/xbox360) Kinect gaming device.  The DryadLINQ research paper won a best-paper award in 2008 during the eighth USENIX Symposium on Operating Systems Design and Implementation.

“It was easily the largest project in our lab,” Yu says. “And this was a long-term project, so management had to believe in it. But they said, ‘We believe in you guys, so here is the money you need to build a server cluster to do the research.’ Also, the entire lab was very supportive—we built the (Dryad) system, and many researchers are using it for real work. Their feedback, in particular, has been invaluable in refining the DryadLINQ programming model.”

Isard adds that while it might seem Dryad had a long gestation, the market time for its release is right.

“I think the HPC product group moved at the right time—when they saw the opportunity,” he says. “We were a year or two ahead of the curve on the research side, but we were ready when the product group saw a need for it.”

Dryad Enters the Market

A big step is coming, as Dryad and DryadLINQ become fully productized as part of the Microsoft HPC Server suite. It also will be integrated with Microsoft SQL Server and Windows Azure to give customers from academia to the business community a new, powerful computing tool.

Isard is confident that Dryad’s ease of use and familiar Microsoft tools will win over developers.

 “Dryad will particularly appeal to customers who would love to keep using Windows and Excel and Visual Studio and all the tools they already use,” he says, “and need a technology for unstructured data analysis that really scales.”

[John Dunagan](http://research.microsoft.com/en-us/people/jdunagan/), a principal architect for Microsoft’s High Performance Computing group, thinks HPC Server customers who use Dryad will find that they now can solve problems that had been challenging.

“We’re convinced that we will delight our customers, both with the pure capability of the system, as well as its ease of use,” he says. “What I really like about Dryad is that is not just about handling a problem in a better way, it is also about new possibilities in computing that you couldn’t imagine before.”

The Microsoft Research team that worked on Dryad is pleased to see its project in a position to seek a larger audience.

“Offering an easy-to-use but powerful, data-intensive computing tool is exciting to see,” Isard says. “It will benefit a whole new set of Microsoft customers.”

Windows Azure and DryadLINQ

**What is Windows Azure?**
Windows Azure is a platform for building scalable, highly reliable, multi-tiered web service applications. It is hosted on Microsoft’s large data centers in the United States, Europe, and Asia. Windows Azure has both compute and data resources. The compute resources are designed to allow applications to scale to thousands of servers and data resources. [Read more about Windows Azure](http://www.microsoft.com/windowsazure/).

**On This Page**

Windows Azure as a Platform for Research in Cloud Computing

**What types of research projects are well suited to Azure?**Windows Azure can be an excellent research platform for many types of research. However, it is designed to support scalable web services, so projects that play to this strength will have the most success. One area of particular interest is computational models and techniques that augment the capabilities of client devices, ranging from feature rich desktop and laptop computers to cell phones of other mobile devices with data and computation resources in the cloud. How can we make the cloud into a transparent extension and experience amplifier of our client-based research tools?

Others interesting areas include:

* Evaluation of cloud computing for a spectrum of algorithms, computational models, and programming patterns.
* Programming abstractions for cloud platforms that offer higher level abstractions and guarantees to application developers.
* Scaling conventional algorithms to cloud scale, coping with load, and dealing with faults.
* Research into scientific data management, data analysis, and new services in the cloud to support data intensive research.
* Data parallel program frameworks, going beyond the map reduce model.
* Experimental platforms for distributed agent technology.
* Compiling declarative and/or functional languages for cloud platforms.
* Sustainable science gateways and virtual organizations in the cloud for web-based application and collaboration frameworks.
* Research to support digital data libraries in the cloud that contain scientific data, not just the metadata, and support integration with published literature.
* New techniques for data visualization of cloud scale data sets, including visual analytics.
* Frameworks to support sensor-based science in the cloud, from services to process data streaming and from sensors to the storage and curation of raw and derived data products.
* Machine learning techniques for data understanding and data generation.
* Research to support intelligent interactions leveraging web data and domain knowledge.

**Will Hadoop or Dryad/LINQ be available on Azure?**There is no port of Hadoop or Dryad/LINQ currently available. However, Windows Azure is an excellent platform for experimenting with new variations on large-scale map-reduce algorithms, as these patterns are easily coded as worker role networks.

**Can I run my MPI HPC applications on Windows Azure?**Windows Azure is not designed to replace the traditional HPC supercomputer. In its current data center configuration it does not have the high-bandwidth, low-latency communication model that is appropriate for tightly-coupled MPI jobs. However, Windows Azure can be used to host large parallel computations that do not require MPI messaging, such as ensemble or parameter sweep studies.

**Can Azure be useful as an experimental host for distributed computing research?**Yes. Windows Azure worker roles have access to standard TCP/IP sockets on each virtual machine (VM) in which they run. Hence it is possible to use a large number of worker roles to experiment with distributed computing algorithms and protocols.

**Can Azure be used to support collaborations and “science gateways”?**Yes. Windows Azure is an excellent platform for sharing “community” data and data analysis tools. Most science gateways are built as web portals and Windows Azure is ideally suited for this task.

**What data collections will be made available?**We will be very interested in suggestions from researchers about important community data collections and tools that can be hosted. We currently have data collections from the NCBI genome databases, oceanographic instrument data, and some MODIS satellite data. We also are providing access to [web scale n-grams](http://research.microsoft.com/en-us/projects/azure/faq.aspx#Web-N-gram-Services) via a service. However, our goal is to let the research community help us define a sustainable collection of shared resources and analysis tools.