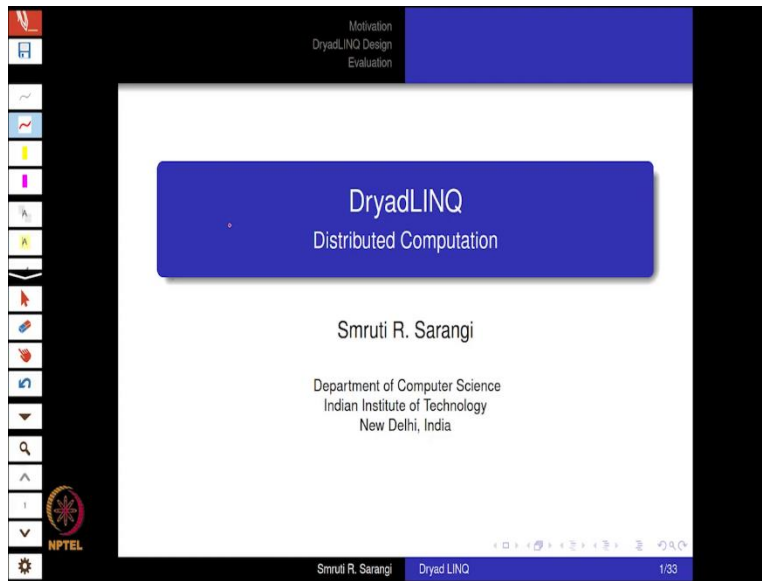


Advanced Distributed System
Professor Smruti R. Sarangi
Department Of Computer Science And Engineering
Indian Institute Of Technology, Delhi
Lecture 25
DryadLINQ: Distributed Computing With High Level Languages

(Refer Slide Time: 00:17)



In this lecture, we will discuss the DryadLINQ system, which is Microsoft's engine for distributed computation. So, prerequisite for this lecture is the lecture on Condor on the Condor distributed batch processing system, which is also there in the same lecture. So, it is there in the same playlist.

(Refer Slide Time: 00:38)

Motivation
Dryad/LINQ Design
Evaluation

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 Dryad/LINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

Smruti R. Sarangi Dryad/LINQ 2/33

Motivation
Dryad/LINQ Design
Evaluation

Dryad-LINQ

- Current programming models for large scale distributed programming
 - Map-Reduce
 - MPI
 - Microsoft Dryad
- A Dryad/LINQ program contains LINQ expressions that:
 - Use LINQ expressions to specify side effect free transformations to data $g(f(\rightarrow))$
 - The Dryad system parallelizes portions of the program, and runs the program on thousands of machines
- A sort of a terabyte level data set takes 319 seconds, on a 240 node system.
** functional programming (ocaml, ml scheme)*
** CONDOR*

Basic Idea
Related Work

Smruti R. Sarangi Dryad/LINQ 4/33

So, we will first discuss the basic idea and the related work. Then we will go through the design and the architecture of the Dryad/LINQ system. And then look at two specific benchmarks Terasort, which is pretty much sought a terabyte of data, and the SkyServer benchmark, which is an astronomical, a benchmark for processing astronomical data.

So, the basic idea is that many of the current programming models for large scale distributed programming like Map-Reduce, MPI, Microsoft Dryad, are essentially of the same type. So, they take they put a lot of load on the user. So, they are kind of based on explicit parallelism. Where pretty much a large part of the planning and the scheduling.

So, I am not discussing planning and scheduling once again, because this was discussed in the Condor lecture. So, where we basically said planning is the mapping of jobs to pools of machines, and scheduling is running a job within a pool of machines the time aspect of it.

So, Map-Reduce, just a quick overview, it essentially takes a large program a large problem, divides it into a set of smaller problems, distributes the small problems to different machines, and again collects and collates the results. MPI is at an even lower level where machines just send messages to each other.

And based on these messages, the final output of a program is computed. Microsoft Dryad is Microsoft distributed execution system where again a program is specified as a directed acyclic graph of jobs or tasks. So, where your Task 1 and Task 2, Task 3 and so on and these are mapped to different machines.

But here again, there is a huge the onus is on the programmer to actually divide a program into tasks specify the dependencies between them, when it comes to the dependencies are specified as a directed acyclic graph and these run on the system, these are kind of extensions of Condor. So, DryadLINQ kind of extends the system.

So, it takes an existing LINQ expression language integrated query expression. So, these are essentially expressions programming language expressions in a .Net framework for each expression is a side effect free transformation of the data. The side effect free is where essentially we can take a data and we can apply a set of functions to it.

So, then of course, the data keeps on changing, but it is not accessing any external data. For example, it is not changing a file, not sending a message over a network or not accessing any other piece of memory that is not contained in here. So, this is a side effect free transformation, which is heavily used in functional programming.

So, LINQ expressions pretty much use that and once a program is written using these LINQ expressions, the Dryad system paralyzes portions of the program and runs it on 1000s of machines. So, when you combine the existing Microsoft's distributed execution engine, Microsoft Dryad with LINQ.

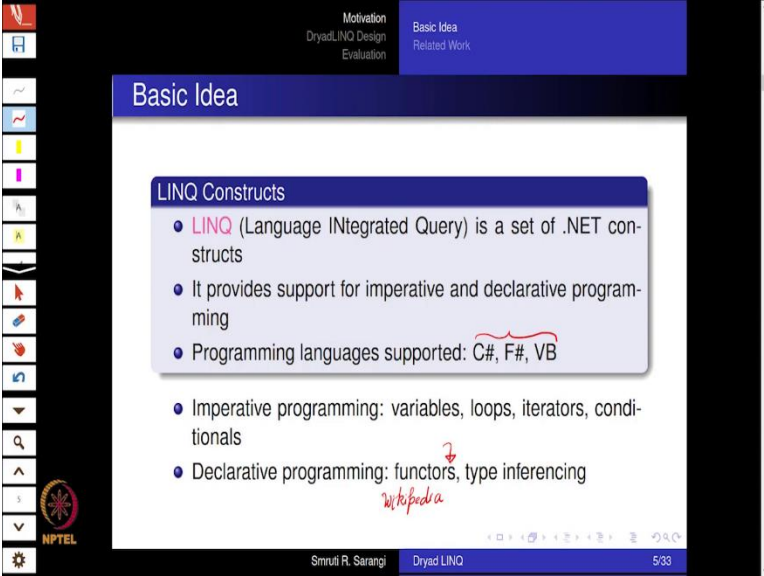
Which is a functional programming paradigm for so, when both of them are combined, you get DryadLINQ. And so this is kind of a state of the art system. So, using this, it is possible to write very simple code to sort a terabyte level data set in just 319 seconds, in just about 5 minutes on a 240 node system, which is not much so, for at 240 nodes.

This is actually a very good result. So, a certain prerequisites over here. Before we proceed, user should have some idea of functional programming particularly when you know a programming that does not use the notion of permanent state. So, it does not use files and networks and so, it does not use this, so even arrays. So, it does not have pointers.

So, very simply have data and you just apply a set of functions to it. So, the notion of functional programming should be clear, otherwise, viewers will not be able to appreciate this lecture. So, functional programming here we are talking of languages like OCAML, MI, scheme and so on. And of course, they should have some idea of a distributed batch processing system.

Where we have a large number of tasks, the relationships, the dependencies between the tasks are specified as directed acyclic graphs, and the lecture on condor which is there in the same playlist would kind of set viewers up for what is coming in this lecture.

(Refer Slide Time: 06:09)



The screenshot shows a presentation slide with a blue header and a white content area. The header contains navigation icons and the text 'Motivation DryadLINQ Design Evaluation Basic Idea Related Work'. The slide title is 'Basic Idea'. The main content is a box titled 'LINQ Constructs' containing a bulleted list. The first bullet point states that LINQ (Language INtegrated Query) is a set of .NET constructs. The second bullet point says it provides support for imperative and declarative programming. The third bullet point lists supported programming languages: C#, F#, and VB. Below this, there are two more bullet points: 'Imperative programming: variables, loops, iterators, conditionals' and 'Declarative programming: functors, type inferencing'. A red bracket is drawn under 'C#, F#, VB' and a red arrow points from 'functors' to 'type inferencing'. A red 'Wikipedia' watermark is visible at the bottom of the slide content. The footer of the slide includes the NPTEL logo, the name 'Smruti R. Sarangi', the title 'Dryad LINQ', and the slide number '5/33'.

So, LINQ, as I mentioned, is Language Integrated Query, which is a set of kind of imperative and declarative constructs in almost all of the .Net languages, which is C sharp, F sharp and VB Visual

Basic. So, there are some, so this is not purely functional, because a pure functional language will not have many of the imperative directives. So, it is kind of like semi functional and semi imperative. So, the imperative directives are, of course, you have variables, loops, and iterators, and conditionals.

For loops, if statements, they are there. And we have standard, so we have a heavy. So, we have a very strong notion of types, very strong notion of inferencing types is a very strongly typed languages. And we have functors. So, I will not describe the background of functors. But I would request the readers to look up the Wikipedia articles on functors.

And in general, in a treating functions as high level objects. So, during functions as separate objects of their pluses, this notion should be clear to viewers before they actually proceed. So, a quick scan of these concepts on Wikipedia would set up users for the rest of the 28 slides in this talk.

(Refer Slide Time: 07:36)

Motivation
DryadLINQ Design
Evaluation

Basic Idea
Related Work

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

NPTEL

Snruti R. Sarangi Dryad LINQ 6/33

Motivation DryadLINQ Design Evaluation Basic Idea Related Work

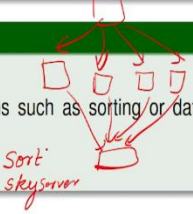
Related Work

Parallel Databases

- Implement only declarative variants of SQL Queries
- The query oriented nature of SQL makes it hard to specify typical programming constructs

Map Reduce

- Not very flexible.
- Hard to perform operations such as sorting or database joins
- Lack of type support




Sort skyscraper

NPTEL Smrutil R. Sarangi Dryad LINQ 7/33

Motivation DryadLINQ Design Evaluation Basic Idea Related Work

Related Work - II

- Domain specific languages on top of MapReduce – Sawzall, Pig (Yahoo), Hive (Facebook)
- They are a combination of declarative constructs, and iterative constructs
- However, they are not very flexible since their pattern is inherently based on SQL
- How is DryadLINQ **different** ?
 - The computation is not dependent on the nature of underlying resources.
 - Uses virtual execution plans
 - Underlying computational resources can change dynamically (faults, outages, ...)



NPTEL Smrutil R. Sarangi Dryad LINQ 8/33

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Overview of DryadLINQ

Structure of a Dryad job

- It is a directed acyclic graph (DAG)
- Each vertex is a program
- Each edge is a data channel that transmits a finite sequence of records at runtime

NPTEL

Smruti R. Sarangi Dryad LINQ 9/33

So, a brief review of related work. So, of course, we always have SQL as the gold standard to compare with, say in SQL, also, we do not specify how something needs to be done. Rather, we specify what needs to be done. So, let us say we want to read a table, and then we want to order them we do not actually work at the level of sorting the data.

So, we do not say use quick sort or merge sort, for example. So, this is a declarative language, where you pretty much tell SQL what needs to be done. And of course, SQL has automatic parallelization mechanisms where databases automatically parallelized SQL Queries. Or also we have variants of SQL known as parallel variants of SQL.

After SQL, we had a large explosion of these big data programming languages. And the prominent among them, the most prominent among them was Map-Reduce. Fair, of course, we take a large problem, break it into small chunks. And so this is a map phase, which is that take a large problem, break it into small chunks, execute each small chunk, and then again, collate the results (())(09:02) reduce.

So, of course, a lot of operations like sorting or database joins was hard. And so basically, so if you would see, two of the problems we have tried to describe over here from the paper are Terasort and SkyServer, which essentially performs database joints. So, both of them are hard to do in a Map-Reduce kind of setting. Also the support for types and Map-Reduce is kind of weak. The reason being that was never the mainstay of the design.

So, if there is any problem because the wrong usage of types. For example, something is an integer on one machine and it is read as floating point on some other machine. There is actually no way to catch such errors. So, then, of course, we have many domain specific implementations of Map-Reduce, so we have Sawzall which is very popular, we have Pig.

So, that also was Yahoo's system. Again, very popular, and of course, Facebook Hive, which is Facebook's rendition of Map-Reduce. So, all of these are rather popular, so use. So, they use the same simplistic notion of a map and reduce kind of operation. And they are also a combination of declarative and iterative constructs and they are inherently extensions of SQL. So, in that sense, DryadLINQ makes a marked departure and tries to do what Condor does to pretty much this area.

So, of course, this is as we had mentioned, at the end of the Condor lecture, that DryadLINQ is kind of a more modern avatar of Condor. So, the first is that unlike Map-Reduce, which makes rather strong assumptions on the system, in this case, the computation is not dependent on the nature of the underlying resources.

So, all the underlying resources are reasonably independent of the computation. The second is that we make a virtual execution plan. So, here also the planning phase and the scheduling phase are separate. So, we make a virtual execution plan, which says that given a set of jobs we will execute, and then of course, the plan is scheduled on a cluster of machines.

And in the cluster machines a lot of things can happen, lot of dynamic changes can happen. We can faults, we can outages, a machine can use swap jobs out, all of those things can happen. So, that would be the scheduling aspect. But here also planning and scheduling are separate. So, what the scheduler gets is a virtual execution plan.

Structure of a Dryad job well, similar to Condor, it is a directed acyclic graph of jobs, T1, T2, T3 and then of course, we can have a job T4, that is dependent on the completion of both T2 and T3. In this case, each of the vertices here is a program. And each edge is a data channel that transmits a finite sequence of records at runtime. So, T4 would get some data from T2 and some data from T3, once it gets the data of both, T2 and T3, then it starts executing.

(Refer Slide Time: 12:43)

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

NPTEL

Smruti R. Sarangi Dryad LINQ 10/33

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Dryad System Architecture

Dryad System Architecture

It contains a centralized job manager whose role is:

- Instantiating a job's dataflow graph
- Scheduling processes
- Fault tolerance → *crash, reschedule*
- Job monitoring and management
- Transforming the job graph at runtime according to the user's instructions → *new*

NPTEL

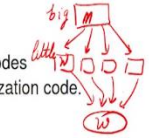
Smruti R. Sarangi Dryad LINQ 11/33

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

DryadLINQ Execution Overview

$f(g(\dots)) \rightarrow f(\dots)$

- 1 The user runs a .NET application. It creates a DryadLINQ expression object that has deferred evaluation. *partial evaluation*
- 2 The application calls the method `ToDryadTable`. This method hands over the expression object to DryadLINQ.
- 3 DryadLINQ compiles the expression, and makes an execution plan
 - 1 Decomposition into sub-expressions
 - 2 Generation of code and data for Dryad nodes *big*
 - 3 Generation of serialization and synchronization code. *little*
- 4 Dryad invokes a custom job manager.

Smrutti R. Sarangi Dryad LINQ 12/33

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

DryadLINQ Execution Overview - II

- 1 The job manager creates a job graph. It schedules and spawns the jobs.
- 2 Each node in the graph executes the program assigned to it.
- 3 When the program is done, it writes the data to the output table.
- 4 After the job manager terminates, DryadLINQ collates all the outputs and creates the DryadTable object. *← results*
- 5 Control returns to the user application.
 - 1 Dryad passes an iterator object to the table object.
 - 2 This can be passed to subsequent statements.

Smrutti R. Sarangi Dryad LINQ 13/33

So, now we will discuss the System Architecture, architecture of the system. So, the Dryad system is architected like this, it contains a centralized job manager. Similar to the manager worker paradigm in Condor, it has a centralized job manager. So, this instantiates a jobs data flow graph.

It also schedules all the processes that are similar to the tracking manager and steering manager in Condor, schedules the processes that are supposed to run, ensure fault tolerance, if there is a crash it essentially restarts a job, monitors the job and transforms the job graph at runtime according to the user's instructions, so this part is new.

So, the job graph can dynamically transform we will see how according to certain directives that the user gives, so this part is new and rather interesting. So, how is the what is the execution plan or the execution overview? So, the overview is like this, the user runs a .Net application and a .Net application creates a DryadLINQ expression object.

So, in functional programming, what we can do is that we can kind of create an expression object. So, it is also possible to partially evaluate it in the sense that the function g will be evaluated on one machine and then what will be sent to the other machine is f with the output of whatever g got and whatever g produced and this can again be evaluated at some other machine.

So, partial evaluation is also tantamount to deferred evaluation, where for a sequence of actions we $(\lambda x.f(x))$ each action is being represented by a function call. Some of it is being evaluated immediately, and some of it is being evaluated later. So, let us say in this case, we evaluate g immediately and evaluate f later. This is called deferred evaluation.

So, of course, people who have studied functional programming would find this very easy to relate to. But for people who have not studied functional programming, I would definitely ask them to do a tutorial over here on Wikipedia or in anywhere else that talks about partial evaluation. So, we evaluate something partially now and then defer the rest for a later point in time.

So, the application calls the method To Dryad Table. So, ToDryadTable, what it does? Is that this method hands over the expression object to DryadLINQ. So, in this case, so, this expression is essentially an expression of a data and what needs to be done with the data. So, this entire thing is the expression object.

So, the DryadLINQ system is given the expression object, DryadLINQ compiles the expression and makes an execution plan. So, the execution plan what it does is first it decomposes it into sub expressions, smaller expressions, it generates the code and data for the different Dryad nodes. So of course, since we are talking of parallelization over here, the same work will be kind of divided and distributed across the nodes.

So, then, for each of them, it will have it may have separate code and data, for separate code and data, whatever it needs to be done over here that is generated. And then similar to Condor, when

data is being sent from a manager to a worker using the same terminology, it is very, very well possible that this uses a big indian representation. And this uses a little indian representation.

So, to ensure that data is transferred correctly, we actually need to make the transfer kind of independent of the exact notation. So, this process is known also a serialization and deserialization. Or it is also known as marshalling and unmarshalling. And once the data reaches there, all the workers need to work.

And maybe let us say you know, after that, they send the data to one more worker. So, he this is like a synchronization point, because it waits for all the other intermediate nodes to finish their work. So, all of this code is automatically generated un belongs to the user. So, all that the user actually gives DryadLINQ is an expression.

And everything else happens automatically, which is what which is decomposition into smaller sub expressions, generation of code and data for the individual Dryad nodes, generation of serialization, deserialization, also known as marshalling and unmarshalling code, and the code to synchronize the data accesses data and code accesses across the nodes.

So, Dryad invokes a custom job manager to do all of this. So, the job manager here creates a job graph. So, the role of the job graph is to schedules and spawn the jobs. Each node in this graph executes the program that is assigned to it. And of course, when the program is done, it writes the data to an output table.

So, almost all the data in Dryad is stored in a tabular format recall percolator and Google big table. So, the inputs also go via a table and outputs also come back via a table. So, after the job manager terminates, which means all the individual tasks terminate, DryadLINQ collates all the outputs and creates the final DryadTable object, which essentially contains all the results. So, control then returns to the user application and what so this can be a very large table. So, this may not be stored in one location. It is actually a distributed table, which is stored across many locations stored across multiple locations, not across one.

So, what Dryad does is that it passes an iterator object. So, the iterator object is essentially a handle to an object to traverse this table. And the iterator object pretty much traverses this table. And

wherever the data is there in the network, it is fetched. And the iterator object is something also that can be passed to subsequent statements.

So, it is like a handle to the result. But the results themselves can be quite large. And they can be distributed. But that does not matter. As far as the invoking application is concerned. To get any row of the table, it just has to use the iterator object and Dryad will automatically fetch it from wherever it is stored.

(Refer Slide Time: 20:20)

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

Smruti R. Sarangi | Dryad LINQ | 1433

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

LINQ

- The base type is an interface `IEnumerable<T>` – An iterator for a set of objects with type `T`
 - The programmer is not aware of the data type associated with an instance of `IEnumerable`
- `IQueryable<T>` is a subtype of `IEnumerable<T>`
 - This is an unevaluated expression
 - It undergoes deferred evaluation
 - DryadLINQ creates a concrete class to implement the `IQueryable` expression at runtime

Handwritten notes:
 $f(f(2,3),5) \rightarrow f(5,5)$ with $2+3=5$ above the inner f .
pre-req class interface
→ OOP programming

Smruti R. Sarangi | Dryad LINQ | 1533

Motivation System Architecture
 Dryad.LINQ Design LINQ
 Evaluation Execution Plan Graph
 Miscellaneous

LINQ SQL Syntax Example

```
// Join two tables: 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);
```

d.docID = r.key

expressions

NPTEL Smruti R. Sarangi Dryad LINQ 16/33

Motivation System Architecture
 Dryad.LINQ Design LINQ
 Evaluation Execution Plan Graph
 Miscellaneous

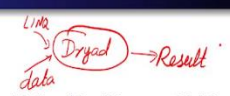
LINK OOP Syntax Example

```
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));
```

NPTEL Smruti R. Sarangi Dryad LINQ 17/33

Motivation System Architecture
 DryadLINQ Design LINQ
 Evaluation Execution Plan Graph
 Miscellaneous

DryadLINQ Constructs



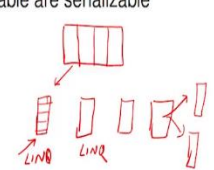
- A DryadLINQ collection (defined by IEnumerable) is a distributed dataset. Partitioning strategies.
 - Hash Partitioning ✓
 - Range Partitioning ✓
 - Round-robin Partitioning ✓
- The results of a DryadLINQ computation are represented by the object – DryadTable<T>
 - Subtypes determine the actual storage interface.
 - Can include additional details such as metadata and schemas.

NPTEL Smruti R. Sarangi Dryad LINQ 18/33

Motivation System Architecture
 DryadLINQ Design LINQ
 Evaluation Execution Plan Graph
 Miscellaneous

DryadLINQ Methods

- All the methods need to be side effect free [no global var]
- Shared objects can be distributed in any way
- The functions to access a DryadTable are serializable
 - GetTable<T>
 - ToDryadTable<T>
- Custom partitioning operators
 - HashPartition<T,K>
 - RangePartition<T,K>
- Functional Operators
 - apply(f,dataset) Applies function f to all the elements in a dataset
 - fork(f,dataset) Similar to Apply, but can produce multiple output datasets.
- Dryad annotations – parallelization, storage policies



NPTEL Smruti R. Sarangi Dryad LINQ 19/33

Now a word or two about LINQ. So, LINQ has a very deep object oriented hierarchy. The base type in LINQ is an interface, recall that an interface primarily defines functions of the type IEnumerable. So, IEnumerable basically is an iterator where we can just go through every entry of the table, that is the base class.

So, it is an iterator for a set of objects with type T. So, the programmer is not aware of the data type that is associated. That is with an instance of IEnumerable, it does not have to know. So, IQueryable is a subtype of IEnumerable. So, this is an unevaluated expression. So, it is kind of those functional expressions that we have spoken to in the past. So, basically, let us say that we want to add 2 + 3.

So, will, say maybe $2 + 3 + 5$. So, this can be represented as $f(f(2, 3), 5)$ where f is the add function, and this and 5. So, it is possible that we can evaluate this, the first f in one machine and then transfer to another machine for deferred evaluation. This is of course, a very simple example.

But in a real life setting, such functional programs do undergo deferred evaluation, because many times the inputs that are required like this 5 over here, this input might not be there, or the resources required to compute this f might not be there, we might need a specialized resource to compute this.

So, that is why the entire function is treated as an expression, which undergoes different phases of evaluation at different points in time. So, to implement the IQueryable expression at runtime, it of course, instantiates this and creates a concrete class. So, what are the pre-reqs to understand what I just said? Well, the ideas of object oriented programming have to be very, very clear, particularly in the context of .Net.

So, it should not be clear, what is a class, what is an interface and how do you instantiate a class. So, these are some of the basic fundamentals that I am assuming before I go forward. So, let me give an example of the LINQ SQL syntax. So, let us say I were to join two tables scoreTriples in staticRank.

So, I were to say that the adjusted score triples is that I take each entry d in scoreTriples, and I join r in staticRank, as long as the document ID of scoreTriples equals the key of the entry r in staticRank. And then I select a new QueryScoreDocIDTriple, which contains d and r . Furthermore, what I do is, I take the adjusted score triples, which are essentially d r , which are essentially triples of d and r , as long as $d.docID = r.key$.

So, if this is a regular database join, there is nothing great with this, but this is exactly the way that we will specify it in LINQ. And so then, we can create a rank queries which means I take each s in adjusted scoreTriples, then I group the s s by you know the s entries by the value of the query s dot query into g . And then for each such group, I take the TopQueryResults. So, as you can see, this is not very different from a regular SQL syntax. So, this is a LINQ SQL syntax example, which is an extension of LINQ to model SQL.

So, the moment that such a query given to the Dryad system, it internally breaks it into expressions and sub expressions. And as we just discussed, there is a complicated compiler chain that instantiates code and data, creates code data Triples that are sent to the individual machines that run parts of this query.

So, again, the same thing can be done with object oriented. So, basically this should be LINQ. And so here, the idea is that instead of using an SQL like syntax, we could use an object oriented programming like syntax. So, in this case, this uses the same thing, where we just call the join function, on the staticRank, and where the doc ID and r, of course, the join is on d and r where d is the doc ID, and r is the key.

And both of these need to be equal. And we then create a QueryScoreDocIDTriple. And we do the same thing we group the queries based on the query field. And finally, for each group query, we select the top Query results. So, this query is exactly the same as the previous one. It is just that LINQ provides different ways of expressing the same thing, just to ensure that it is compatible with all the existing query languages.

So, any DryadLINQ collection IEnumerable is essentially a distributed data set. So, it is so the input is always a table similar to SQL. It is a large table, similar to Google big table as well. So, you can partition that in several ways. So, after the large data set, needs to be partitioned among the nodes such that each node can work on a part of it.

So, either what we can do is we can hash these elements, each of these elements, so let us assume that each of these elements, we can hash them, and we can store them in the sorted order or their hashes and partition them on the basis of the hashes. This is hash partitioning. Or we can take any single column of the row and based on the value of the column, we can create ranges and partition or we can simply distribute the rows and the Round-robin Partitioning.

The results of any single DryadLINQ computation are represented by the object DryadTable T, where T is the type of the data. And the subtypes will determine the actual kind of storage whether if there is an integer, we store it in a certain way. If it is a floating point, well of course, we then store it in another way. And we can also include additional details to specify the schema, the exact storage structure and the metadata of these DryadTables. So, what is the broad idea now?

Well, the broad idea is that if this was our DryadLINQ system, what essentially we get in is we get LINQ expressions and we get in data. The data is stored as tables. Instantiations of the IEnumerable interface. Dryad internally partitions the tables based on any one of these algorithms hash, range or Round-robin partitioning. Then the individual partition tables are sent to different machines where LINQ expressions or sub expressions run on them.

And finally, we get the result which is a table of course, a table is very large. So, it is a distributed table where parts of it are stored on different machines. But the client that actually issued the request does not care because it gets an iterated in return. And the moment it accesses something Dryad knows where to locate the data.

So, all of the methods as we have discussed in the past even with Condor have to be side effect free. So, being side effect free means that only the part of the data that we are changing actually that changes. So, there are global variables another way of side effect free means that there are no global variables no file is nothing outside the scope.

So, shared objects can be distributed in any way. So, the functions to access a DryadTable are serializable. So, serializable in this case means that the serializable semantics are followed. So, when I say get table, this means that I first you know I get the entire table. So, this is like a transaction. And ToDryadTable as we saw is a gateway function for starting in a Dryad job.

And we can have, we can also define our partitioning functions, such that the tables can be partitioned. So, we can define our custom hash partition function, or a custom range partition function. So, this in a certain sense is conceptually like Map-Reduce. For Map-Reduce partitions the computation and Dryad partitions the data.

So, in this case, if you look at it, a large table is being partitioned into smaller tables, regardless of how it is being done, each of these smaller tables are dispatched to different nodes. And then on the different nodes, LINQ expressions actually work on the table to produce an output they can be the same LINQ expression or different LINQ expressions.

In addition, we have two specialized functions that actually run on them. One is the apply function, which applies a given function f to all the elements in the table, so we take each row of the table, or we take each cell of the table right within each column of each row.

And we apply the apply function to all of them. For example, the apply function can be 2 multiply by 2. So, we can take the table and multiply every single entry by 2 that would do it or we can have a fork function. So, it is similar to apply but output can be not just one data set, but multiple data sets.

The fork function can take this and output can be like two tables, that is a fork. And Dryad is these may be expressive. So, it supports different parallelization and storage policies and they can be provided as directives.

(Refer Slide Time: 31:47)

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

Smruti R. Sarangi | Dryad LINQ | 20/33

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

System Implementation

Execution Plan Graph (EPG)

- DryadLINQ converts the raw link expressions to the nodes of the EPG
- The EPG is a DAG
- A part of the EPG can also be generated at runtime based on the values of iterative and conditional expressions
- DryadLINQ also needs to respect the metadata (node requirements, and parallelization directives) while generating the EPG
- Needs to support the deferred evaluation of functions

LINK


```
graph TD; A((1)) --> B((2)); A --> C((3)); B --> D((4)); C --> D((4)); D --> E((5));
```

Smruti R. Sarangi | Dryad LINQ | 21/33


Motivation
Dryad.LINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Static Optimizations



- **Pipelining** : One process executes multiple operations in a pipelined fashion
- **Redundancy Removal** : Remove dead code, and unnecessary partitioning $100 \quad (1000) \quad | \quad 100$
- **Eager Aggregation** : Intelligently reduce data movement by optimizing aggregation and repartitioning
- **I/O Reduction** : Use TCP pipes, and in-memory channels to reduce persistence to files




Smruti R. Sarangi Dryad LINQ 22/33

Motivation
Dryad.LINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Dynamic Optimizations



Optimally Implementing OrderBy

- Deterministically sample the values.
- Plot a histogram, and compute the appropriate keys for range partitioning
- A set of vertices now perform the range partitioning.
- A node now fetches the inputs, and then sorts them. These two actions can be pipelined.

Sampling → range partitioning → fetch & sort

Smruti R. Sarangi Dryad LINQ 23/33

Motivation
DryadLINQ Design
Evaluation

Terasort
SkyServer

Results: Terasort

- The number of **nodes** was varied from 1 to 240
- Each node stored 3.87 GB of data.
- The execution time was 120s for 1 machine, and quickly jumped to 250 s.
- Then it grew very **slowly** (sub-linearly) to 320s.

Source [1]

NPTEL

Smrutti R. Sarangi Dryad LINQ 31/33

So, let us now discuss the execution plan graph or the EPG. So, EPG is something like this. So the DryadLINQ system converts this also should be LINQ expressions to the nodes of an EPG. So, the EPG is DAG. So, we essentially create tasks and then sub tasks and then there are relationships between the tasks in the same way a DAG functions.

Furthermore, it is possible that a part of the EPG can be generated at runtime based on the values of iterated or conditional expressions. So, this was mind you not possible in Condor. So, here, let us say we can have a task, if statement, we can add an extra task. Or if statements, so the statement is successful, we add an extra task, otherwise, we do not.

So, it is possible and dynamically based on the data, the EPG can change. So, this is a rather powerful feature of DryadLINQ. Furthermore, similar to the class add system, in Condor, we specify the requirements of a job in its metadata. So, this specifies what are the requirements of the physical nodes of the machines that are actually running the job?

And what are the parallelization directives? What are the directives that we use to actually paralyze the tasks while generating the EPG. Second, this is new in a functional paradigm, we need to support the deferred evaluation of functions. This is something that we have to do to support the LINQ semantics.

So, there are several kinds of static optimizations that we can do. So, these are not dynamic optimizations, these are essentially compile time optimizations that we can do. So, the first is

Pipelining. So, which means that a single process can execute multiple operations, so let us say it can take one query.

So, it can just kind of pipeline them in a pipeline fashion, which essentially means that let us say I can take some data, process it. So, I can you know, within a node, I can have multiple threads, associated with a process where one thread produces some output with a second thread consumes that produces and output the third thread consumes so on and so forth.

So, it is possible to do pipelining within a node, we can remove redundancy, which means remove dead code, remove LINQ expressions that will not be evaluated. We can remove unnecessary partitioning.

Let us say that prior to compiling; we realize that the user has specified we need to create too many partitions. But in practice, maybe the maximum number of partitions you can tolerate is 100. And the user is saying look, you create 1000 partitions. So, we can remove this, so the user can only give hints but not orders in the system.

So, we can remove this and we can only create 100 partitions based on the system you are looking at. So, what are the two optimizations we have seen till now pipelining, which is again, intra node and redundancy removal? The third is Eager Aggregation. So, this intelligently reduces data movement by optimizing the aggregation and re partitioning steps.

So, which essentially means that whenever we take any data, essentially we break it down, we partition it into several pieces of smaller data, they are mapped, and again, they are aggregate. But let us say there are too many partitions, then the process of aggregation will take a lot of time.

And also these data items can be mapped to different machines, and there will be a lot of communication as well. So, we can instead do smart aggregation. So, maybe a single machine can process these two partitions. And a single machine can process these two partitions. And maybe one of the machines can do the aggregation as well.

So, in this case, what will happen is that we are reducing the data movement and this will further cut down on the latency. Finally, we can reduce the amount of IO. So, to reduce the amount of IO instead of creating TCP connections for every single transfer, you can have a persistent TCP connection.

And if there are different processes on the same physical node, we need not write to file switches the default actually across nodes. So, we need not use files, we can use in memory shared memory channels to transfer data between processes. So, the main aim of the fourth point is that wherever possible, we try to make use of whatever the underlying system is providing in terms of either TCP pipes like persistent TCP connections, or in memory channels.

To increase performance as much as possible, and let us say for example files are slow and shared memory is fast. So, if shared memory is available will always opt for shared memory and not go for files.

A few dynamic optimizations does a dynamic optimizations happen when you actually see the data and the job runs. So, consider OrderBy. So, OrderBy is one of the SQL queries, this is one of the SQL directives, which pretty much talks about sorting the rows based on a key value of a certain column. So, what we do is we deterministically sample the values.

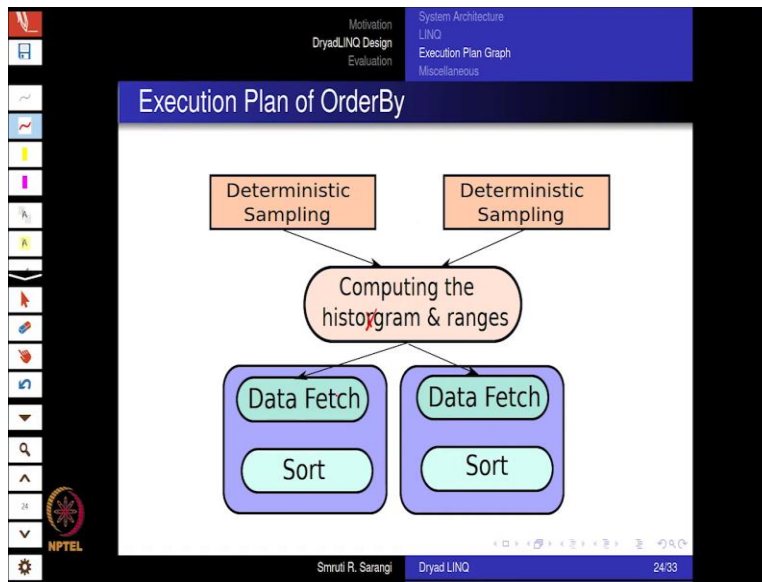
So, we just randomly sample the values first, we create a histogram and then we do a range partitioning. So, let us say the sample the values only find that look the way that the value is actually vary a histogram of this will be something like this, then maybe we create one range partition to be like this, we try to keep the area under the curve the same.

So, we can create you know some sort of intelligent range partitioning based on our sample. So, this will ensure that each physical node is given the same number of entries to sort. Once we have partitioned the ranges, then we can kind of divide the input based on the ranges among the nodes, each node fetches the inputs assigned to it, and sorts it.

So, clearly, we can also pipeline these two actions of fetching the inputs and sorting them. So, this can be done. And so, this is one example where we need to see the values at runtime and then first, so, this is exactly how Terasort was implemented first to sampling to give an idea of what are the range of values from the histogram, we figured out a range partition, you perform a range partition on the table of data, we assign a range to each node.

So, the node basically fetches data and simultaneously sorts whatever it has fetched this can be pipeline. So, this is an example of a dynamic optimization that is done performed in the DryadLINQ system.

(Refer Slide Time: 39:38)



The slide discusses code generation for the EPG. It includes the following points:

- The EPG is a **virtual execution plan** (handwritten: *shadow*).
- DryadLINQ dynamically generates code for each EPG node (handwritten: *Shadow*).
- DryadLINQ generates a **.NET assembly snippet** that corresponds to each LINQ subexpression.
- It contains the **serialization and I/O code** for ferrying data.
- The EPG node code is generated at the computer of the client (**job submitter**), because it may depend on the local context. Values in the local context are embedded in the function/expression. The expression undergoes **partial evaluation** later. (handwritten: *local files*, *PE*, *As*)
- Uses **.NET reflection** to find the **transitive closure** of all .NET libraries. The EPG code, and all the associated libraries are **shipped** to the cluster computer for remote execution.

So, this is kind of a pictorial representation of what was just described, where we have deterministic sampling first. And deterministic sampling provides us the histogram and the ranges based on that we allocate a range to each node and it pipelines the fetch and sort.

So, the EPG over here is a virtual execution plan and DryadLINQ has to dynamically generate code for each EPG node. So, this we have discussed in the past as well. So, what Dryad LINQ does is it generates a .Net assembly snippet.

So, .Net has its own assembly language syntax that corresponds to each LINQ sub expression, it contains all the code for serializing data and IO and for ferrying data. So, all that is built in the user need not be bothered about any of these steps. The code of the EPG node is generated at the computer or the client which is the job submitted.

So, the nice thing about .Net is that it is actually a virtual instruction set similar to Java byte code. So, we can always generate the .Net code on any machine and when it physically runs on the actual machine. There is a translation between the virtual code that .Net has generated and the real machine instructions.

So, this is very similar to the way that Java and Java byte code work. So, the EPG code node code is generated on the computer or the client the job submitter, because it may depend on certain things of the local context. So, similar to a condor shadow, there might be local files that are in the local file system, there might be environment variables.

So, all the values in a local context get embedded in the function expression. So, which means that if let us say this is the entire LINQ expression, whatever can be brought from a local context, all of that is kind of embedded in the expression. So, then we have a partially evaluated, so let us call it PE partially evaluated expression.

This partially evaluated expression is now dispatched to each of the individual nodes for further evaluation for deferred evaluation. So, we use .Net reflection. So, .Net reflection is basically so reflection is a general concept. And it had come in Java a long time back. So, in this context, actually, the .Net virtual machine takes a look at each class or takes a look at all the libraries, and finds all the libraries that it is dependent upon.

So, those dependencies will have dependencies. So, the transitive closure is pretty much the set of all the libraries that you know, let us say there is library A. All of the libraries that it is dependent upon all of the libraries that they are dependent upon, so on and so forth, till we have captured the entire set.

So, that the EPG code and all the associated libraries are shipped to the cluster computer, or remote execution. So, this is a marked departure from what Condor does in this situation. So, Condor

actually sets up a channel between the sandbox and the shadow such that a sandbox need something which the remote machine does not have.

So, then the sandbox in Condor would actually send a message to the shadow, saying that look, I need this, the primary reason being that these are actually separate setups. And the notion are deferred evaluation was not there. So, and also the network speed those days was slow.

And also, the other thing is that maybe they use different versions of an OS, any code over here would actually not run over there and vice versa. But because of the fact that .Net, like Java is a virtual ISA. So, it is very portable across all kinds of machines, the network is fast. Hence, in this case, we do not do what condor does. But instead, we ship the EPG code and all the associated libraries to the remote machine; everything is sent.

(Refer Slide Time: 44:14)

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

Smruti R. Sarangi Dryad LINQ 28/33

Motivation
DryadLINQ Design
Evaluation

System Architecture
LINQ
Execution Plan Graph
Miscellaneous

Interacting with other Frameworks

PLINQ

- Runs a subexpression in a cluster node in parallel using multicore processors.
- Uses user supplied annotations (mostly transparent to the user).
- Uses parallel iterators (similar to OpenMP)

SQL

- DryadLINQ nodes can directly access SQL databases.
- They can save internal datasets in SQL tables.
- Can ship some subexpressions to run directly as SQL procedures.

Smruti R. Sarangi Dryad LINQ 27/33

Motivation System Architecture
 DryadLINQ Design LINQ
 Evaluation Execution Plan Graph Miscellaneous

Debugging

Debugging massively parallel applications is very difficult

Rescue DAG

NPTEL

Smrutil R. Sarangi Dryad LINQ 28/33

Motivation System Architecture
 DryadLINQ Design LINQ
 Evaluation Execution Plan Graph Miscellaneous

Debugging

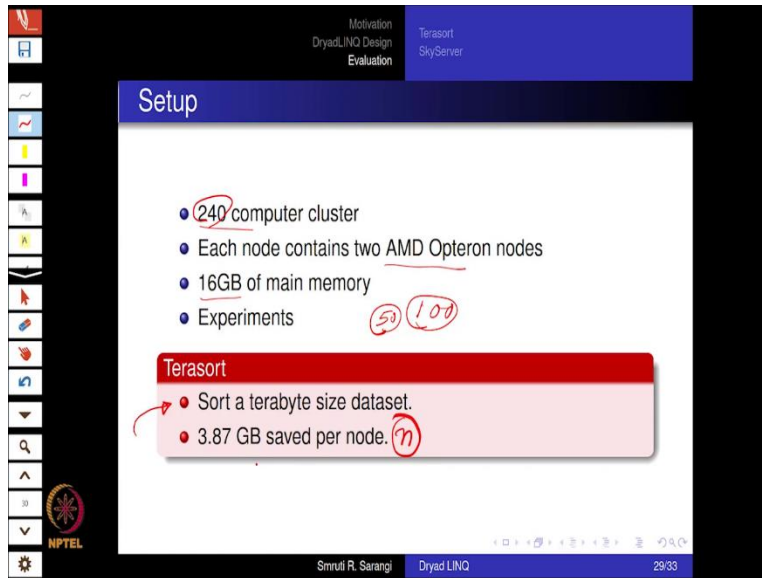
Debugging massively parallel applications is very difficult

Debugging

- Visual Studio .NET interface to debug the DryadLINQ program on a single computer
- DryadLINQ has a deterministic replay model
 - It is possible to replay the entire execution – event by event
 - Secondly, it is possible to replay any subexpression on a local machine and view the outputs
 - Performance Debugging
 - Collect detailed profiling information.

NPTEL

Smrutil R. Sarangi Dryad LINQ 28/33



A few miscellaneous items. So, there are extensions of the LINQ system. So, we have PLINQ where it runs a sub expression in a cluster node in parallel using multi core processors. So, that also, that extension is there, where the users can supply annotations. And these are similar annotations similar to parallel programming variants like OpenMP, where it specifies how these iterators are supposed to run.

And also LINQ interacts with SQL we have already seen an example how, where it can directly access SQL databases, same internal data sets in SQL tables, and also translate LINQ sub expressions to run directly as SQL commands. So, certain interoperability with SQL was added.

So, if you would actually see the design, which is based on tables, at least at this level, it is similar to SQL. Even differences arise later, in terms of deferred evaluation and usage of the basic .Net system. But at least there is a certain level of high level compliance, which allows a degree of interoperability even with traditional database systems.

How do we debug a massively parallel application? Well, how did we deal with this problem in Condor? In Condor, what we did is we had the notion of a rescue DAG. So, here, what happens is that if the post processing script of a Job said that a given job could not execute correctly, then we take the entire DAG, we cut out the part that executed correctly, and then again, cut out the part which did not execute correctly. This can then be analyzed by the user.

And the user can figure out, out of this, which job did not execute correctly, that part again can be like Repatched. And then the entire Dryad can run months ago. So, .Net, of course, had the advantage of the Visual Studio interface to debug many of these things. And the great and the really praiseworthy thing that DryadLINQ added is a deterministic replay model.

Which means that given the fact that these are actually kind of functional programs, which are side effect free and which do not rely on anything which is outside the environment. So, essentially, if you look at any LINQ sub expression, it is like a program where all of the inputs are specified.

So, every time that you run this program, regardless of wherever you run it, because .Net like Java, is provides a degree of platform independence if input remains the same output has to remain the same. This means that we can, let us say that a given job a given task failed on a remote node, we can then bring it back to the client machine, we can provide it the same inputs, which it got on the remote node.

And we can essentially debug it line by line and see exactly what happened. So, we can pretty much replay the entire execution event by event and view the outputs on the local machine figure out what went wrong, fix it and send it back. So, this is the advantage that there is no non determinism in the execution, which is actually there in other frameworks like MPI and so on.

And non determinism would only crop up if this would depend on something which is not really an input, but produced by some other task at an intermediate halfway state. And then depending on the latency of the network, the execution can be non deterministic, it can either get a certain event or not, get a certain event, the time of getting it also matters.

So, all of this has been avoided. So, this is simply a LINQ sub expression. It is given a set of inputs fully or partially evaluated. And then for the same set of inputs, the output will be the same, the behavior will be the same on all machines that support .Net. And so this has taken care of the performance sorry of the debugging aspect very well. So, along with collectors debugging, we can also have performance debugging, where we collect detailed execution profile information.

For example, it is possible that a given job runs very slowly. So, to actually fix that, we can collect detailed runtime statistics to figure out why a job runs slowly. And then from the information maybe we can find out why. So, the execution setup it was a 240 computer cluster. Each node

contains two AMD Opteron nodes, which are server processors 16 gigabytes of main memory, which was, which was considered good 10 years ago.

The first was Terasort where we sort at a terabyte sized dataset. And so essentially, the idea here was that so of course, it is a terabyte when we consider 240 nodes. Otherwise, if we consider n nodes it is 3.87 gigabytes per node. So, the way that they did the experiment is that they consider a terabyte of data for 214 nodes, but they also considered you know 100 nodes 50 nodes and so on. See the number of nodes is n the data was scaled accordingly. So, it was 3.87 n.

(Refer Slide Time: 50:08)

Motivation
DryadLINQ Design
Evaluation

Terasort
SkyServer

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

Smruti R. Sarangi | Dryad LINQ | 30/33

Motivation
DryadLINQ Design
Evaluation

Terasort
SkyServer

Results: Terasort

- The number of nodes was varied from 1 to 240
- Each node stored 3.87 GB of data.
- The execution time was 120s for 1 machine, and quickly jumped to 250 s.
- Then it grew very slowly (sub-linearly) to 320s.

Source [1]

Smruti R. Sarangi | Dryad LINQ | 31/33

So, the first as we discussed Terasort. So, Terasort to the execution time. So, again, the data was kept constant per node. So, when you had a single machine, the execution time was 120 seconds. And it quickly jumped to 250 seconds and kind of the execution time was like an asymptotic behavior with it kind of saturating at 320 seconds.

And of course, at one point with 240 nodes, the data was 1 terabyte. So, the reason for this, you know asymptotic behavior. So, of course the starting point is not 0, it is 120 is that as we add more nodes, there are more overheads in terms of EPG compilation, dispatching the inputs and the code to different nodes, network delays, switch delays, and so on.

But still, it is not that bad given that from 1 to 240 machines, the execution time, just increased by around 2.6, 2.7 times in triple. And sorting a terabyte size data within 5 minutes with rather simple code with the partitioning mechanisms that we have discussed, is genuinely praiseworthy.

(Refer Slide Time: 51:26)

Motivation
DryadLINQ Design
Evaluation

Terasort
SkyServer

Outline

- 1 Motivation
 - Basic Idea
 - Related Work
- 2 DryadLINQ Design
 - System Architecture
 - LINQ
 - Execution Plan Graph
 - Miscellaneous
- 3 Evaluation
 - Terasort
 - SkyServer

Smruti R. Sarangi Dryad LINQ 32/33

Motivation
DryadLINQ Design
Evaluation

Terasort
SkyServer

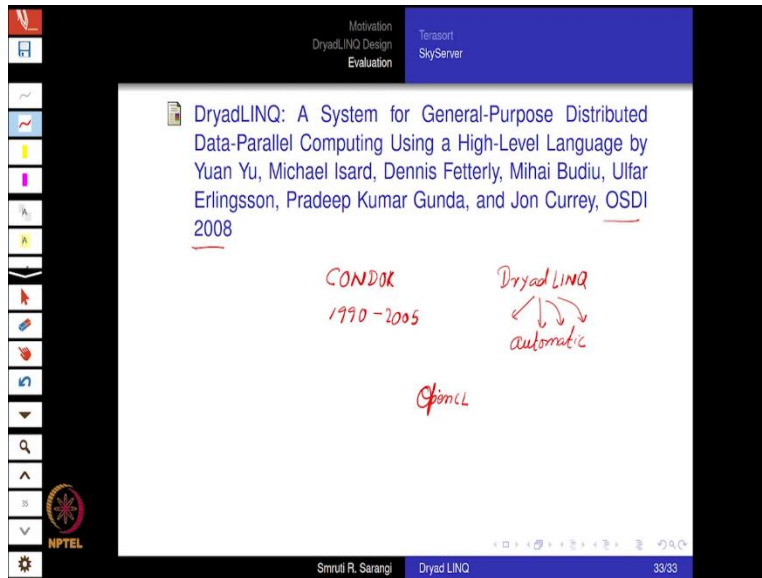
SkyServer benchmark

- Three-way join for two tables containing astronomical data.
Size: 11.8 GB and 41.8 GB
- The number of machines was varied from 1 to 40
- The speedup increased from 1 to 19 sub-linearly for DryadLINQ
- The speedup increased from 1 to 24 sub-linearly for Dryad Two-pass

Source [1]

automatic.

Smruti R. Sarangi Dryad LINQ 33/33



The other was a skyserver benchmark, which contains a lot of astronomical data. There were so it was a 3 way join for 2 tables. One was 11.8 gigabytes others size was 41.8 gigabytes. And the number of machines was varied from 1 to 40. So, the speedup increased. So, they compare this with the baseline Dryad system that does not use the LINQ expression does not use this mechanism, but just uses the basic distributed framework.

So, of course, the speedup was much higher with Dryad increase from 1 to 24 sub linearly of course, and this went down from 24 to 19 in the case of DryadLINQ, but well if you think about it, it is not that bad given all the automatic things that DryadLINQ actually does, and how much it makes programming easy for us, easier for us.

So, the paper that describes the DryadLINQ was published in OSDI 2008, around 12 years ago. So, for those days, these were fantastic results. So, now of course, because of an improvement in hardware, the results have become better, no doubt. But the important point is that for leveraging a cluster of heterogeneous machines, we need systems like Condor of course, which was like, which saw its 3 days from 1990 to let us say, 2005.

And then, of course, the DryadLINQ kind of system. And now, of course, we have many more systems. So, Condor did small a large number of other projects like LSF and load leveler and so on, which are still heavily used in companies. And DryadLINQ has also spawn a large number of similar projects, where we are talking of really expressive mechanisms, of describing a parallel execution.

And the underlying framework actually does a lot for us and that too automatically. So, this is a continuing area of research, because along with regular processes, we have GPUs and accelerators and so on. And so then, for GPUs, of course, we are OpenCL, which is not exactly distributed, but this is kind of in that direction.

So, we have many such frameworks and will continue to have, so this area is clearly not going to die over here. As the diversity of computing devices increases, will have more and more of such frameworks, which actually do more of the planning and scheduling automatically.