Monitoring and Debugging DryadLINQ Applications with Daphne

Vilas Jagannath, Zuoning Yin, Mihai Budiu
vbangal2@illinois.edu, zyin2@uiuc.edu, mbudiu@microsoft.com

Abstract

Debugging and optimizing large-scale applications is still more art than engineering discipline. This document describes our experience in building a set of tools to help DryadLINQ application developers understand and debug their programs.

The core infrastructure for our tools is a portable library which provides a DryadLINQ *job object model* (i.e., a local representation of the distributed state of an executed application). Layered on the job object model we have built a variety of interactive and batch tools for: performance data collection and analysis, distributed state visualization, failure diagnostics, debugging, and profiling.

1. Introduction

The emergence of high-level programming frameworks for large-scale distributed systems, such as Map-Reduce [1] Hadoop [2], and DryadLINQ [4] has led to an explosion of interest in the development of very large scale batch-processing applications. The success of these frameworks is due to the fact that they hide the complexity of the underlying distributed systems from the programmer by providing a simple sequential programming language interface and a single-system abstraction. (In this document we will use the term "job" for such distributed computations.)

Unfortunately the abstraction provided by these frameworks is quite fragile and breaks down when bugs are encountered (either performance or correctness bugs). To diagnose application problems programmers have to understand the structure of the distributed job and the mapping between the original program source and the distributed code running on the cluster. The diagnoses commonly involve "combing" through log files spread among the cluster machines where execution was performed. The scale of the systems involved magnifies the difficulties, since a single computation can generate millions of distributed processes and billions of files. When this infrastructure is exposed to the programmers the daunting complexity of the underlying distributed system becomes visible, negating most of the benefits of the simple programming language.

The end goal of this research project is to simplify the development experience. We attempted to address these issues by building tools for monitoring, profiling and debugging distributed jobs. We started by building several batch and interactive tools. During this process we have discovered much commonality in their structure, so we have factored a common API, which we then used to rewrite the tools. We regard this API as the main technical contribution of the current paper.

This API provides a structured view of the distributed information describing a job. We call this view *a job object model or JOM* (similar to the document object model DOM provided by web browsers to JavaScript engines). Unlike the DOM, the JOM provides a read-only API for clients. It aggregates information generated by a large number of sources from the cluster runtime, the job submission system, the job control process and from processes executed on behalf of jobs on the cluster machines. Data sources include: the job plan, cluster runtime logs, application logs, performance counters, job inputs and outputs, and even the results of querying various cluster services. Since we are targeting large-scale computation platforms, the execution of a single job can lead to the generation of a huge amount of state information (e.g., terabytes of logs). For this reason parts of the JOM are computed *lazily*, e.g., in response to user actions in a browser GUI. This enables us to build interactive tools with good response time even when browsing large data sets.

Having built a JOM one can then more easily build a set of job understanding tools; we dedicate most of this paper to describing the tools which we have created as companions to the DryadLINQ system. In particular, we discuss tools for: browsing the job state to monitor job execution, automatic job failure diagnosis, interactive debugging, scripting job data analyses and performance data collection and analysis. Most of these tools are integrated with Daphne, an interactive job browsing application. Artemis, the performance analysis toolkit [8] is now integrated with Daphne. In the Greek mythology Daphne is the tree nymph of the laurel tree. The tree nymphs are also called Dryads. Artemis is their closest friend.

Whereas Daphne is designed to analyze DryadLINQ programs, we believe that our approach is quite general, and could be employed for other popular classes of distributed jobs, such as Map-Reduce and Hadoop, Sawzall [6], Pig [3] or FlumeJava [7].

2. Background

To better understand the flow of control and the distributed job state we start by describing our cluster infrastructure , shown in Figure 1, including our compiler (DryadLINQ) and runtime (Dryad) [5].

2.1 DryadLINQ

DryadLINQ is a compiler and runtime which allows users to execute .Net programs on large computer clusters. DryadLINQ compiles LINQ [9] constructs into distributed execution plans, and uses the Dryad distributed runtime to reliably execute these plans on a distributed computer cluster.

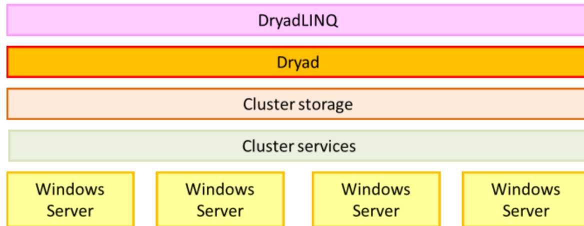

Figure 1: The DryadLINQ cluster software stack

LINQ is essentially a language of operators that compute on *collections* of values. Chains of LINQ operators can be applied to an input collection, forming LINQ *queries*. Each LINQ query is translated by DryadLINQ into a Dryad job (as described in the next section), which is then executed by the Dryad runtime on the cluster.

While conceptually the user writes a single program that operates on a set of collections and runs on a typical workstation, at runtime the program is executed using multiple machines, and the collections are partitioned, stored and manipulated by multiple machines concurrently, providing high throughput computation on very large data.

2.2 Dryad

A Dryad job (Figure 2) is a directed acyclic graph: the nodes of the graph (also called *vertices*) are processes that run independently, often on different machines. The edges of the graph are communication *channels* that move data between the vertices. The vertices are usually organized in *stages:* all vertices in a stage perform the same computation on different partitions of a large dataset.

Dryad assumes that vertices are deterministic, functional and idempotent: i.e., their behavior only depends on the data in the input channels, and re-executing a vertex several times will produce a functionally equivalent output. Dryad takes advantage of these properties to provide fault-tolerant execution through re-execution and speculative execution of vertices. Dryad supports multiple channel types; most frequently the Dryad channels are implemented as persistent files, offering automatic checkpoint and restart at vertex granularity.

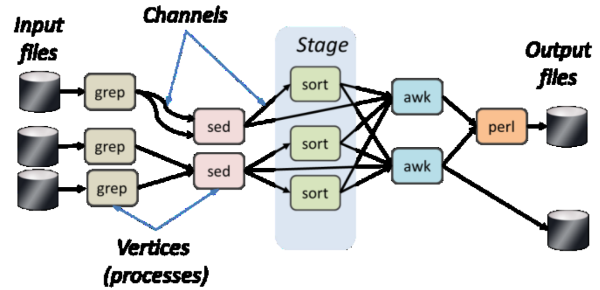

Figure 2: Dryad job structure

2.3 DryadLINQ Operation

To use DryadLINQ the user writes and executes a LINQ program on the *local client* workstation. The DryadLINQ system transparently generates executable code and an execution plan for Dryad and invokes the execution of the Dryad program on a cluster. Figure 3 shows the steps of this process in detail:

1. The application executes a LINQ query on the local client.
2. The DryadLINQ provider compiles the query into a Dryad execution plan.
3. The DryadLINQ system uses a job submission library to contact the cluster runtime and to initiate the job execution.
4. The job submission library sends the job (including binaries, file resources, job plan, etc.) for execution to a cluster-level scheduler, which is usually accessible through a firewall.
5. The cluster scheduler allocates resources for the job and initiates the job execution by contacting a PN service to start the job manager. The PN is a service (part of the “cluster services” box in Figure 1) that runs on all machines in the cluster and provides remote execution and monitoring capabilities.
6. The PN creates a sandbox and starts the execution of the Dryad job manager process. The job manager is the central control point which oversees the execution of the job.
7. The Dryad job manager reads the job plan and starts creating, dispatching and monitoring the vertex processes. The job manager invokes the PN services from the other cluster machines to execute the vertices remotely.

Each vertex is executed by a PN in a sandbox on the local machine. Vertices use the local storage in the sandbox to write logging information. Each vertex writes the temporary and output files in the local sandbox, and may read the input files from a remote sandbox.

On job completion the control signaling goes in the reverse direction (not shown in the figure): the cluster scheduler notices the completion of the job manager process and returns a result to the job submission

library on the client workstation, which returns to DryadLINQ, which returns to the application.

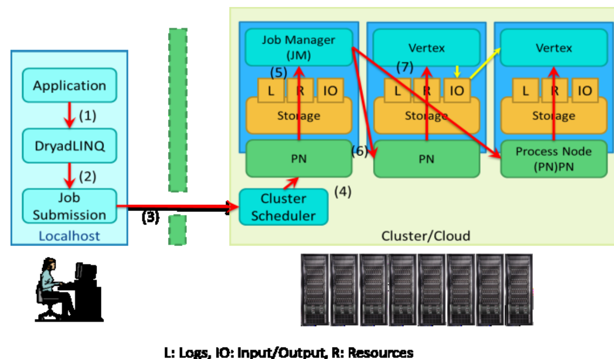

Figure 3: DryadLINQ detailed operation: red arrows denote control, yellow arrows denote data movement.

Figure 3 is quite complex. Normally the DryadLINQ developers are shielded from this complexity; however, while debugging performance or correctness problems the developers are exposed to most of the details of this architecture. To make matters worse, the tools available to inspect the distributed state depend on the cluster platform that is being used (Cosmos, Windows Azure or Windows HPC).

2. System Architecture

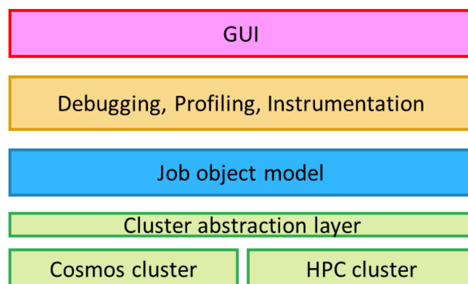

Figure 4: Software stack of the DryadLINQ job manipulation tools.

Figure 4 shows the architecture of the software stack for the job manipulation tools. Through the use of abstraction layers (both at the level of cluster and JOM) this stack is portable across several cluster system architectures (we have not yet ported our tools to the Azure cluster services layer, but we support several versions of HPC, Dryad, and DryadLINQ). Most of these tools run on the client machine (but they occasionally spawn DryadLINQ computations when they need to collect large amounts of data from the cluster).

At the bottom is a cluster-abstraction layer which hides the specifics of each cluster platform. It provides a narrow interface for enumerating jobs, cancelling them and locating job placement on the cluster.

2.1. The Job Object Model

The JOM is a set of .Net classes which provides a view of a running job: it contains representations for the job and its vertices, and it provides APIs to discover the state of the job vertices, their location and their associated logs, inputs, and outputs. It also represents other important job-related entities, such as the input and output files, the static job plan, the job schedule. The main parts of the JOM are a job object and a list of vertex objects. The top-left pane in Figure 6 shows a view of the job object, while the top-right pane shows a view of one vertex object.

The JOM is built from a variety of heterogeneous sources of information; by hiding the heterogeneity and the platform-specific details under a generic API the JOM insulates the tool developers from details of the actual system which change on different cluster runtimes and with new software versions. For example, new versions of DryadLINQ and Dryad have changed the format of the log messages and plan representation, however, most of the tools continued to operate unchanged.

The bulk of the state used to build the JOM is obtained from the job manager logs and the static job plan. The Dryad job manager process emits one log line for each important state-machine transition of a job vertex (e.g., vertex is ready, started, running, cancelled, failed, or completed).

In building the JOM we had to trade-off between comprehensiveness and portability. We have erred towards portability: to provide cross-platform functionality, the JOM uses a minimal amount of platform-specific data. For instance, it does not use information from the PN service or the cluster scheduler. The nature of such information differs substantially between the various cluster software platforms. For example, only on some platforms the PN service can provide information about the resource consumption of the vertex (memory, CPU, etc.).

3. Job Inspection Tools

Here we describe a set of tools that can be used for inspecting the distributed job state.

3.1 Powershell API

Given the JOM it was straightforward to implement an extension for Microsoft's PowerShell [10] that operates on job objects. One can then use the full Powershell syntax to query for failed jobs, long-running vertices, logs, etc. This feature is very useful for power users and cluster administrators. Here are some sample queries written in PowerShell:

Find the failed jobs on cluster X:

```
get-cluster X | select-allJobs |
where-object { $_.Status –eq “Failed” }
```

Diagnose 3 failed jobs run by user Y:

```
get-cluster X | select-allJobs |
where-object { $_.User -match “Y” } |
select-object -last 3 | select-DryadJob |
Diagnose-Job
```

Find all the failed vertices in the last job executed:

```
(get-cluster X | select-AllJobs  |
 sort-object Date | select-object -last 1 |
 select-DryadJob).Vertices |
     where-object { $_.State -eq "Failed" }
```

3.2 Interactive Job Visualization

On top of the JOM we have also built a set of interactive enable users to explore the distributed job state.

3.2.1 Cluster browser

The cluster browser is shown in Figure 5. Besides filtering and sorting, the cluster browser provides a few other elementary operations: job cancellation, starting the job browser (described below), collecting profile information (Section 3.2.3), and diagnosis of failures (described in Section 4). The cluster browser also offers a set of “administrative” operations, including the starting and stopping of performance counters on the cluster.

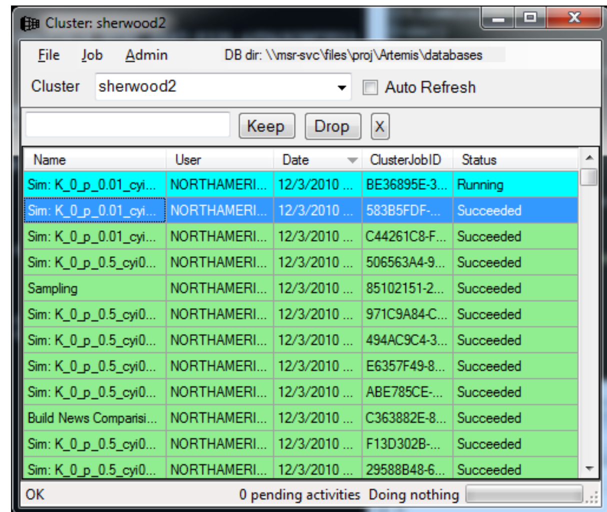

Figure 5: screenshot of the cluster browser

3.2.3. Daphne, the Job Browser

Figure 6 shows a screen-shot of the Daphne job browser, the most complex and useful tool in the set. The job browser GUI is divided into three vertical panes, representing the natural job structure hierarchy, from left to right.

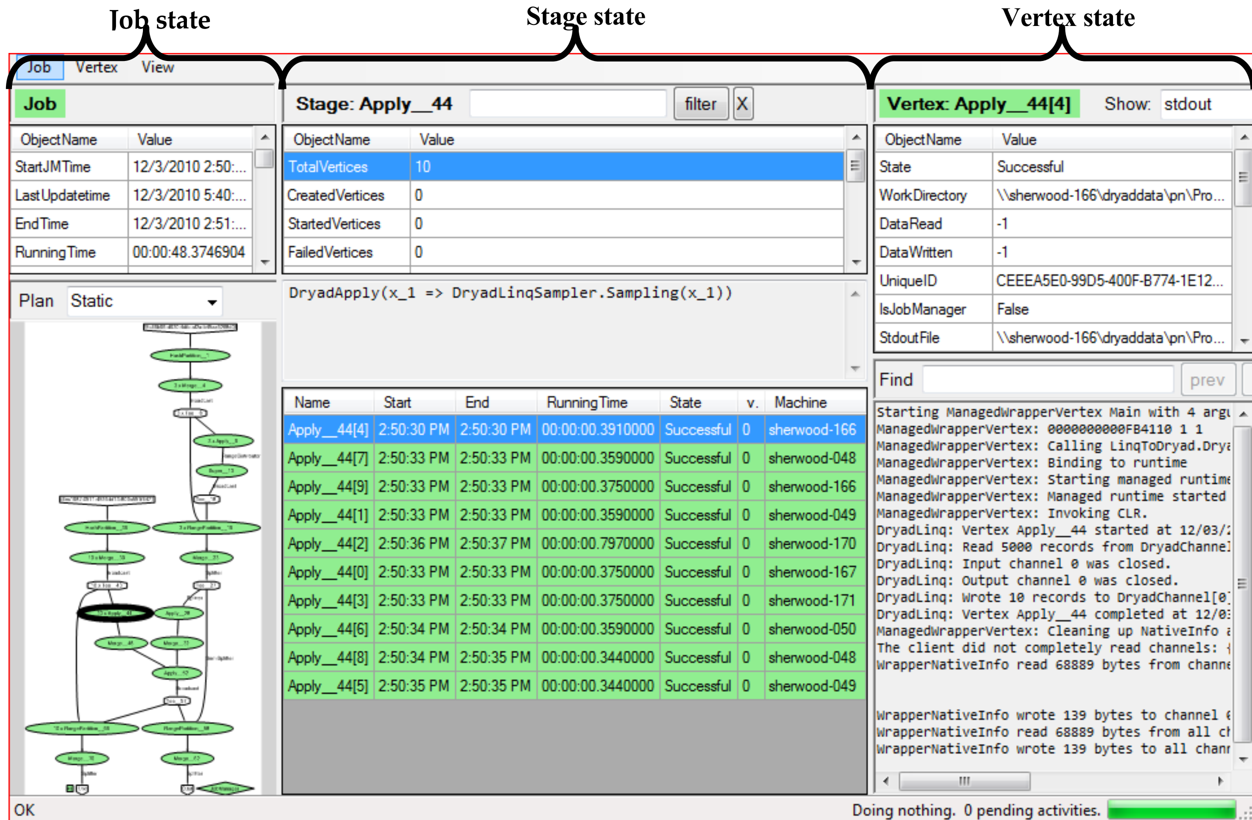

Figure 6: Screenshot of Daphne, the job browser.

The job state pane (left) shows global job information. At the bottom-left is a depiction the job plan, each oval representing a stage. It is worth pointing that there is no simple one-to-one correspondence between the static job plan and the vertices of the job executed, since Dryad can alter the job plan dynamically. Dryad can execute each vertex many times, due either to failures or speculative execution. Also, Dryad can dynamically remove and insert vertices or stages while a job is executing. For this reason the browser can display either static or dynamic views of the job. The user can select one of the job stages by clicking on the job plan. The oval circled with a bold line is the selected stage.

The selected stage (Apply_44 in this figure) is displayed in the central pane. The middle of the central pane displays the LINQ code executed by the selected stage (a DryadApply statement). The stage information includes the list of all vertices in the stage (bottom middle); 10 vertices are shown in the figure. The user can click on a vertex to select it (in the figure vertex Apply_44[4] is selected). Information specific to the selected vertex is shown in the right (vertex) pane.

The rightmost pane allows the user to further drill-down into the selected vertex (Apply_44[4]) state. The user can visualize vertex inputs, outputs, work directory, logs or standard output. Our current implementation discovers this information by browsing remotely the files in the vertex sandbox.

3.2.3 Performance data collection

To diagnose job-level performance problems we have built support to manipulate of performance counters collection on the cluster during job execution. Our tools include a GUI to visualize the performance data and a set of analyses for discovering performance anomalies using statistical methods. These tools have been described in a previous publication [8], but have now been refactored to use the JOM.

Each machine in the cluster collects measurements for the same set of counters and writes the measurements in a segmented log file persisted on the local disk. The user can initiate the collection of performance counters corresponding to a completed job from the cluster browser. Due to the large volume of data that needs to be collected (requiring scanning of potentially thousands of logs from each machine), the data collection is implemented as a DryadLINQ job. Normally the counter collection is a post-mortem activity. The collection job performs a large distributed join by selecting from the perfmon logs on each machine only the measurements corresponding to vertices that ran on that machine.

4. Failure Diagnosis

One of the hardest tasks for a novice DryadLINQ application programmer is to understand the reasons for failing jobs. Diagnosing is difficult in part due to the complex chain of invocations shown in Figure 3. The various software layers involved were not designed to relay structured job failure information from the cluster to the client application (i.e., there is no direct communication channel from the Dryad job manager process back to the client application, which crosses the firewall in the reverse direction and tunnels through the cluster scheduler and job submission libraries).

Additional complexity stems from the fact that jobs can fail in many different ways, so developers have to know where to look for the failures. For example, application-level errors may result in vertices crashing or deadlocking. The vertex runtime attempts to trap all unhandled exceptions and to save the managed stack trace at the exception point, but some vertex errors do not generate stack traces (in particular, stack overflow exceptions). Obscure errors can be caused by usage of improper library versions (such as using mismatched versions of .Net or Dryad libraries). Data corruption can affect many seemingly unrelated vertices (e.g., corruption on a disk storing inputs for many different vertices). Moreover, some errors are benign, and can be overcome by the automatic fault-tolerance capabilities of Dryad. When multiple distinct errors occur in a single job execution they are difficult to disentangle.

To help users navigate the complex error conditions we have built an automatic diagnosis tool. The diagnostic uses a decision tree to pinpoint the root cause of a job failure. The decision tree is quite complex, and aggregates many sources of information starting from the JOM: the job manager standard output, the job manager error logs, vertex error logs, the vertex stack traces, and statistical analyses for correlated failures across machines. The job failure decision tree is composed of two sub-trees: the root tree performs job-level diagnosis (e.g., incorrect job submissions, cluster-related problems, etc.). The top-level tree invokes a vertex-level decision tree if it decides that job failure is caused by a specific vertex which fails deterministically.

The decision tree is the least portable of our tools. While some of code is generic, many parts had to be specialized for failures specific to the cluster runtime used or a specific software version employed (different software versions fail in very different ways).

The decision tree approach is naturally incomplete and can fail provide a diagnostic for conditions which are not encoded in the tree. The decision tree attempts to locate the root cause of a failure, which is often quite different from the low-level error code generated the runtime which relays the error.

The job and cluster browsers allow the developers to invoke the decision trees (both for jobs and vertices) with a single mouse click. The decision tree can be used also on running jobs (which haven't yet failed) if the jobs have a significant number of vertex failures (which may

indicate impending job failure or infrastructure/hardware malfunctions).

The job diagnosis should be integrated into the job submission software layer from Figure 2, to automatically provide the developers with high-level error messages about jobs that fail. We have not yet done this integration.

We have also built a cluster monitoring service that builds a catalog of all job failures encountered. This tool has helped us improve the coverage of the decision tree, by pointing out incomplete diagnoses. The service automatically emails to the users a diagnosis of their failed jobs; it also emails the DryadLINQ developers every time a failure seems to be due to a bug in DryadLINQ, and it emails the cluster operators when failures seem to be due to hardware-level malfunctions. This service has helped DryadLINQ developers prioritize fixing bugs which happen frequently.

5. Debugging

When a job crashes the application developer needs to debug it. The simplest case is when a vertex fails while executing the managed code written by the user; then the vertex generates a stack trace relayed back to the job manager. However, even in this simple case the stack trace contains references to the code *generated* by DryadLINQ, which can be quite different from the original user LINQ query.

Often the vertex stack trace is not enough to pinpoint the root cause of a failure. For this reason we offer several debugging scenarios.

5.1 Vertex debugging on the client machine

This scenario takes advantage of the fact that Dryad job vertices are meant to be re-executed in case of failure. The Dryad runtime generates several scripts to support manual re-execution of vertices in the correct environment.

We have integrated this debugging scenario within the Daphne job browser. The user can select a vertex by browsing the job and then initiate local debugging using a mouse click. The result of this action is a process running under the control of a local debugger running the vertex code on the developer machine.

Debugging is implemented by creating a temporary “sandbox” on the local machine and copying the content of the PN’s vertex sandbox from the cluster. The debugging tool also locates relevant pdb files (containing debugging information and symbol mappings) and the cluster configuration files that are needed by the vertex runtime for proper execution. These files are obtained from the client machine.

5.2 Vertex profiling on the local machine

To diagnose vertex-level performance problems we offer a profiling scenario. Profiling is also integrated in the job browser. Profiling proceeds similarly to debugging, but instead of attaching a debugger to the local vertex, it uses the Visual Studio managed code profiling facilities to instrument and execute a vertex and generate a traditional Visual Studio managed code profiling report.

5.3 Interactive debugging on the cluster

DryadLINQ attempts to provide a single-machine-like environment for distributed applications. Ideally, debugging DryadLINQ applications should also provide the illusion of executing the debugger on a single machine. The user should be able to set conditional breakpoints, inspect the stack and heap, and start and stop execution.

The scenario we present in this section attempts to emulate an interactive debugging session while the application is executing at scale, on the cluster.

Note that there are some fundamental obstacles to debugging interactively large-scale applications using a traditional debugging metaphor. For example, a Dryad job stage can encompass thousands of processes, which could all hit the same breakpoint. The user interface of the debugger does not scale to handle thousands of simultaneous processes. Also, each vertex has a separate address space, with different values for the variables. As such, the value of a conditional breakpoint expression can be different in each vertex.

In the interactive debugging scenario the users run the client DryadLINQ application in a Visual Studio debugging session on the local client machine. The user can create and insert conditional breakpoints. The application runs and spawns the Dryad computation on the cluster. When a Dryad vertex reaches code guarded by a conditional breakpoint a new “process” window appears in the debugger, as shown in Figure 8.

The experience of debugging the application in this way is similar to debugging a traditional multi-threaded application, except that each vertex runs in a separate address space on a separate machine. The debugging is performed by using the local visual studio debugger to perform remote debugging on vertex processes on the cluster.

We had to make several changes to the DryadLINQ compiler and runtime to enable this functionality:

1. The Visual studio remote debugging stub executable (msvmon.exe) is shipped as a resource of the Dryad job (which makes it available to all vertices in their sandbox).
2. At start-up each vertex launches a remote debugger stub process.

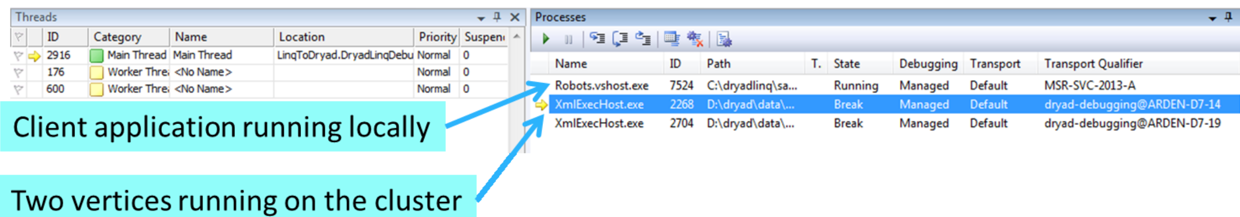

Figure 7: Screenshot of Visual Studio showing interactive debugging on the cluster. Each vertex is shown as a separate process that is being debugged.

3. The vertex next sends a "notification" to the application running on the client machine, signaling its readiness to be debugged. Since the software stack (shown in Figure 2) includes no such notification channel between the vertices and the client application, we have emulated the "notification" mechanism entirely on the client-side machine, by using the DryadLINQ JOM. We have integrated the JOM with the job submission library. The JOM polls the cluster to update the job state periodically and sends notifications to client application when interesting vertex state events have been discovered (e.g., the debugged vertex has been started).

4. To give fine-grained control to the user on which vertices are to be debugged, we augmented the DryadLINQ API to expose to the user application a static "vertex_to_debug" delegate. The delegate is evaluated by each vertex on its local state; when the delegate returns 'true' the vertex is debugged remotely. With the help of this delegate users can restrict the range of vertices to be debugged based on the vertex runtime environment (e.g., debug only the first 3 vertices of a stage).

5. Each debugged vertex enters a loop waiting for a remote debugger to be attached.

6. When it receives a notification that a vertex is waiting for a debugger, the client-side DryadLINQ application attaches a remote debugger to the vertex. Note that attaching a debugger requires suitable permissions to cross the cluster firewall and to debug processes on the cluster.

7. The DryadLINQ application running on the client uses COM automation to instruct the client-side debugger to attach to the remote process.

5. Related work

In recent years, there has been a large body of research on high-level programming frameworks for large-scale distributed systems like DryadLINQ [4], Hadoop [2], Pig [3] and Map Reduce [1]. Debugging remains a big challenge for systems built using these frameworks [12], [13]. Many systems, including Pig and DryadLINQ offer a local debugging mode which can be used to execute the application on the client workstation on a small dataset. This mode is useful for finding some types of application bugs, but it obviously does not help with errors caused by large scale data (e.g., an integer overflow when counting the elements in a large set), or errors due to the cluster environment.

To debug failures on the cluster cases users resort to inspecting logs on the cluster machines manually. Cloudera tools [14] provide a rich set of tools, some of which overlap in functionality with our cluster and job browser. Chukwa [15] is a log collection framework that can be used to monitor Hadoop clusters, which includes tools for resource visualization; it is related to the Artemis component of our toolkit. Jiaqi Tan et al have done extensive work on diagnosing problems for Hadoop computations [17], [18], [16] using a variety of tools, including visualization, and statistical methods; these are also related to the Artemis performance analysis component of Daphne.

Windows Azure [19] offers a debugging scenario using Visual Studio which is related to our client vertex debugging scenario.

The main contribution of our work is the system architecture for building a rich suite of job monitoring tools, based on a job object model. We also offer several powerful interactive and debugging scenarios, including live debugging of vertices on a cluster initiated from a debugger running on the client workstation. Our solution for this problem is inspired by the Visual Studio debugging support for MPI applications [11].

6. Summary

For the foreseeable future programmers will need to be able to understand details of the execution of their large-scale applications. This document describes our attempt to address this problem by building a set of abstractions and tools for understanding distributed computation state: the key abstraction is a DryadLINQ job object model; using the model it is relatively easy to build a rich set of portable tools for distributed job state visualization, performance collection and profiling, failure diagnosis, and debugging. These tools have been very well received by the community of

DryadLINQ system and application developers; in particular, these tools are greatly helping new users to overcome the steep learning curve of programming large clusters. Even power users and system developers are finding these tools to be very convenient when they deal with a port of DryadLINQ to a new unfamiliar cluster platform,

7. References

[1] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[2] Apache Software Foundation. (2007) Hadoop. [Online]. http://hadoop.apache.org/

[3] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008.

[4] Yuan Yu et al., "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *OSDI*, 2008.

[5] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, "Dryad: Distributed data-parallel programs from sequentialbuilding blocks," in *EuroSys*, 2007.

[6] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277-298, 2005.

[7] Craig Chambers et al., "FlumeJava: Easy, Efficient Data-Parallel Pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[8] Gabriela Cretu-Ciocirlie, Mihai Budiu, and Moises Goldszmidt, "Hunting for problems with Artemis," in *USENIX Workshop on the Analysis of System Logs (WASL)*, San Diego, CA, 2008.

[9] Erik Meijer, Brian Beckman, and Gavin M. Bierman, "LINQ: Reconciling object, relations and XML in the.NET framework," in *SIGMOD*, 2006.

[10] (2009, May) Windows PowerShell Getting Started Guide. [Online]. http://msdn.microsoft.com/en-us/library/aa973757(VS.85).aspx

[11] Daniel Moth. (2009, October) VS2010 MPI Cluster Debugger launch integration. [Online]. http://channel9.msdn.com/posts/DanielMoth/VS2010-MPI-Cluster-Debugger-launch-integration/.

[12] Michael Armbrust et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50--58, April 2010.

[13] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh, "A First Look at Problems in the Cloud," in *HotCloud*, 2010.

[14] Cloudera Inc. (2010) Cloudera Management Tools. [Online]. http://www.cloudera.com/products-services/tools/

[15] The Apache Software Foundation. (2008) Chukwa. [Online]. http://incubator.apache.org/chukwa/

[16] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan, "Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop," in *HotCloud*, 2009.

[17] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan, "Visual, Log-based Causal Tracing for Performance Debugging of MapReduce Systems," in *ICDCS*, 2010.

[18] Jiaqi Tan et al., "Kahuna: Problem diagnosis for Mapreduce-based cloud computing environments," in *NOMS*, 2010.