Technical Report Nr. 756

Systems Group, Department of Computer Science, ETH Zurich

Efficient data-parallel computing on small heterogeneous clusters

by

Simon Peter, Rebecca Isaacs, Paul Barham, Richard Black, Timothy Roscoe

April 9, 2012

Efficient data-parallel computing on small heterogeneous clusters

Simon Peter[*], Rebecca Isaacs[†], Paul Barham[†], Richard Black[†], Timothy Roscoe[*]

[*] *Systems Group, Department of Computer Science, ETH Zurich*
[†] *Microsoft Research, Cambridge*

Abstract

Cluster-based data-parallel frameworks such as MapReduce, Hadoop, and Dryad are increasingly popular for a large class of compute-intensive tasks. Such systems are designed for large-scale clusters, and employ several techniques to decrease the run time of jobs in the presence of failures, slow machines, and other effects. In this paper, we apply Dryad to smaller-scale, "ad-hoc" clusters such as those formed by aggregating the servers and workstations in a small office. We first show that, while Dryad's greedy scheduling algorithm performs well at scale, it is significantly less optimal in a small (5-10 machine) cluster environment where nodes have widely differing performance characteristics.

We further show that in such cases, performance models of dataflow operators can be constructed which predict runtimes of vertex processes with sufficient accuracy to allow a more intelligent planner to achieve significant performance gains for a variety of jobs, and we show how to efficiently construct such models. Our system enhances the DryadLINQ data-parallel language compiler with a planner/optimizer implemented using constraint programming, and can exploit our operator models to significantly enhance the performance of parallel jobs on ad-hoc clusters.

1 Introduction

This paper explores the challenges in executing data-parallel programs efficiently on hardware platforms which are composed of a small number of machines that vary considerably in performance characteristics.

Coarse-grained data-parallel frameworks such as Google MapReduce [7], Yahoo! Hadoop [1], and Microsoft Dryad [13] are increasingly popular for efficiently processing large datasets on clusters of computers, and process petabytes of data every day as part of large online services.

These systems abstract a computation as a dataflow graph of operators (typically small sequential programs supplied by the programmer) connected by communication links (which might in practice be files), and are appealing because they provide conceptually simple access to parallel computing resources, particularly when combined with powerful languages such as Pig Latin [16] or DryadLINQ [23].

While originally intended for large datacenter environments, there has recently been interest in using these systems for smaller-scale applications in other kinds of hardware environment, for example rented virtual machines [24]. We are particularly interested in the issues in running data-parallel jobs on small, heterogeneous collections of machines which we term *ad-hoc clusters*.

On a scale of parallelism, ad-hoc clusters are situated between today's data-centers and regular shared multiprocessor machines. The limited, but still significant parallelism makes them interesting for a range of tasks, not least because of their widespread presence even today.

An illustrative example is a small business which may not own a cluster of high-end machines, but has fast Internet connectivity and regularly needs to process data-intensive jobs such as payrolls, data mining of customer information, and so on. An ad-hoc cluster in this case might consist of both the company's workstations and some additional datacenter machines rented for a limited time. The Internet connection is fast enough to accommodate the data transfer for the ad-hoc computation.

To give a second example, a freelance design draftsman or computer animation artist might own five or ten machines that have accumulated over the years, including desktops and laptops. All machines have different CPU, memory, disk and network configurations, and throughput for each can vary by an order of magnitude (a close relative of one of the authors, a freelance 3D modeler, uses just such a cluster for rendering).

Ad-hoc clusters have three interesting characteristics from the point of view of data-parallel applications.

Firstly, their small size often implies that resource allocation options are tightly constrained, in contrast to the ample freedom found in large datacenters. In this paper we show the impact of this on scheduling algorithms.

Secondly, the cluster is highly diverse. Some machines may have ample storage but little network bandwidth, or vice versa. This suggests that placement of CPU or storage-intensive data flow operators can have a disproportionate impact on application performance, a conjecture we confirm in Section 5 of this paper.

Thirdly, failures are rare. Though a machine in an ad-hoc cluster may have a slightly shorter mean time to failure (MTTF) than those in a datacenter, there are relatively few machines to fail. In very large clusters machine failure during a job is the norm, whereas with an ad-hoc cluster it is rare, to the extent that it is not unreasonable to fail an entire job if a single machine fails during execution.

Finally, different use cases of an ad-hoc cluster can have different performance metrics, e.g. job latency, power consumption, or monetary cost subject to a deadline (particularly if rented virtual machines are involved).

Contributions

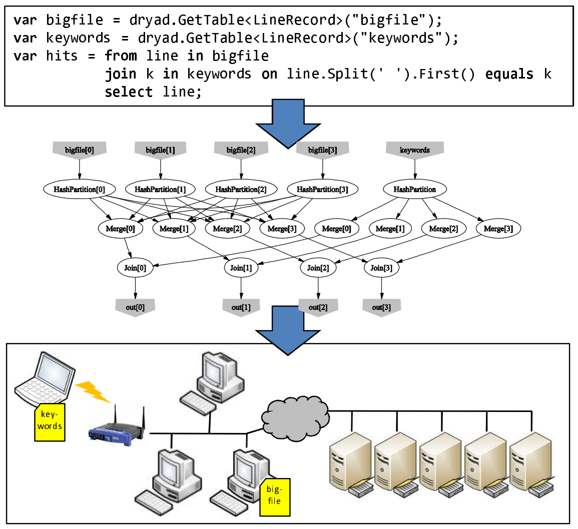
This paper makes the following contributions:

First, we show in the next section that while the default greedy schedulers used in most data-parallel systems exhibit near-optimal performance in medium- and large-scale installations, they perform less well on small clusters with a significant degree of node heterogeneity. Using Dryad as our example of a data-parallel programming framework, we explore the space of such small, ad-hoc clusters ill-served by conventional scheduling and placement algorithms.

Secondly, we introduce in Section 3 the idea of building performance models for Dryad vertex operators, and show that, in contrast to traditional results for centralized relational database systems, such models can indeed generate predictions which can be related to end-to-end application performance on ad-hoc clusters, and may serve as a basis for placing and scheduling Dryad vertices on an ad-hoc cluster.

Thirdly, in Section 4 we show how to integrate such models into a planning system that can take unmodified DryadLINQ programs and generate execution schedules for small, heterogeneous clusters based on hardware information. Our system uses the ECLiPSe logic constraint solver to implement the scheduler.

Finally, we evaluate our scheduler and demonstrate its significant performance improvements relative to the default Dryad scheduler in Section 5.



```
var bigfile = dryad.GetTable<LineRecord>("bigfile");
var keywords = dryad.GetTable<LineRecord>("keywords");
var hits = from line in bigfile
           join k in keywords on line.Split(' ').First() equals k
           select line;
```

Figure 1: DryadLINQ Execution Pipeline

2 Context

We use the Dryad system, and the DryadLINQ programming environment, as a convenient context for our work. Dryad is one of the more flexible MapReduce-like frameworks, and is in daily production use within Microsoft (for example, in Live Search). Furthermore, DryadLINQ provides a convenient programming language model above Dryad. Nevertheless, we believe the ideas we explore in this paper are applicable to most other coarse-grained data-parallel computing systems. In this section we describe Dryad, DryadLINQ, and the Dryad scheduler, and then present simulation results to motivate our work.

2.1 Dryad

Dryad [13] is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad application is formulated as a dataflow graph. Dryad dataflow graphs are directed acyclic graphs that connect computational *vertices*, usually implemented as sequential programs, with communication *channels* (edges in the graph). Channels are realized through files, TCP pipes and shared-memory FIFOs.

An application is run by dispatching vertices onto a set of available computers. Data flows from one or multiple data sources through channels and vertices to one or more data sinks. Data sources and sinks (referred to as input and output tables) are realized through files. Dryad is intended to scale from single powerful multicore computers, through small clusters, to data centers with thousands of machines.

Dryad dataflow graphs provide opportunities for both

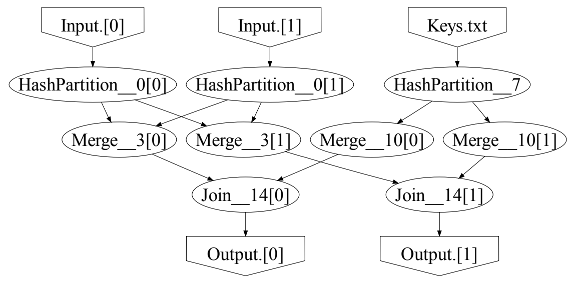


Figure 2: Dataflow graph of the *Join* query. Oval shaped nodes are executable programs, triangles represent files.

horizontal parallelism, whereby independent vertices can be executed on different processing nodes in parallel, and *vertical parallelism*, via pipelining of a series of dependent vertices on the same or different processing nodes. Vertices are *dependent* when they are connected by a channel. Dryad does not permit independent vertices on the same processing node at the same time.

A Dryad cluster is made up of a single *name server*, which knows the names and topology of the computers in the cluster, a *job manager*, which coordinates Dryad jobs, and a lightweight daemon running on each computer to enable processes to be created on demand. This infrastructure is unintrusive, making it feasible for a computer that is primarily used for other tasks to be part of a Dryad system.

To run a Dryad job, a dataflow graph is handed to the job manager, which builds the *schedule* (an assignment of vertices to machines, and an order in which to run them). The job manager then executes that schedule by communicating with the relevant daemons and transmitting the vertex binaries.

2.2 DryadLINQ

DryadLINQ [23] provides a programming model for data-parallel applications by extending C# with expressions for arbitrary side-effect-free transformations on datasets [3]. The data-parallel portions of a DryadLINQ program are translated into a Dryad execution plan, as shown in Figure 1. The program is executed by running a single binary; Dryad transparently handles distributing the computation.

An example DryadLINQ query is shown at the top of Figure 1. The "*Join*" query performs a relational join of two datasets, the first consisting of text lines representing key-value pairs and the second a list of keywords. The output dataset containst all of the lines whose key matches an entry in the keywords file.

Figure 2 shows the Dryad graph of the *Join* query when the first (larger) dataset is partitioned across two nodes. The query hashes the keywords to send records from each input file deterministically to one of the two `Join` vertices. The join is then performed in parallel and the output dataset also partitioned across two disks. The seemingly superfluous Merge vertices with only one input are needed for the correct operation of some vertex types, like Join.

2.3 Dryad scheduling

Dryad's job scheduler performs simple greedy scheduling, by dispatching vertices that have all their input channel requirements satisfied onto the next free machine. Machines are assumed to be homogeneous and well-connected. Temporary heterogeneity in execution speed through node misconfiguration, failures or throughput fluctuation is alleviated by speculative execution: when a vertex executes longer than a set threshold, Dryad dispatches an identical vertex on a different processing node, in the expectation that this will outperform the vertex already running.

Dryad can exploit vertical parallelism by pipelining vertices using TCP pipes and FIFOs. However, Dryad does not choose communication channel types itself and this is left to the programmer. The choice of appropriate channel type depends heavily on vertex, workload and cluster properties.

The scheduler makes several assumptions about job execution and expected workloads. Firstly, cluster nodes are assumed to execute at roughly the same speed, and parallel vertices are assumed to complete processing at roughly the same time (and hence are grouped into computation *stages*).

Secondly, vertices taking significantly longer to execute are classified as *outliers* and handled by speculatively re-scheduling them on a different node, as explained above. Furthermore, it is assumed that launching a speculative task on an otherwise idle node is cheap.

Finally, node failures are expected and performance in the presence of failures is deemed more important than failure-free performance. Thus file-based connections are preferred to TCP/FIFO pipelines as they provide stable checkpoints of vertex computations.

2.4 Discussion

While appropriate to a datacenter environment, we note that none of Dryad's scheduling assumptions hold on an ad-hoc cluster. Nodes have different processing speeds and connectivity, and so they do not execute at equal speeds and vertices within a stage will rarely finish at similar times. Worse, Dryad's default scheduler logic

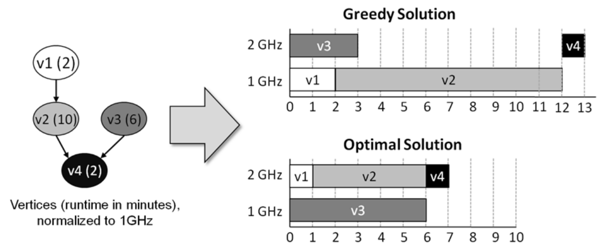


Figure 3: Example of sub-optimal schedule yielded by greedy scheduling policy.

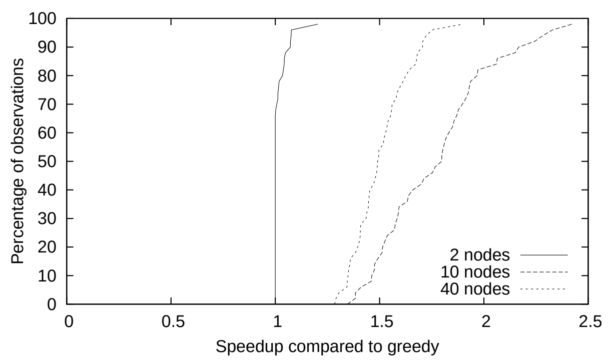


Figure 4: Comparison of intelligent vs. greedy schedules on randomly-generated ad-hoc clusters.

may falsely classify some vertices as outliers and delay the progress of the job.

Furthermore, ad-hoc clusters provide less parallelism. Launching speculative tasks is expensive if the cluster is already fully utilized by non-speculative vertices.

Finally, failure-free performance is more important than dealing with (rare) failures. Thus, pipelines using non-persistent channels which exploit vertical parallelism are more attractive for good performance.

The greedy scheduling used in Dryad assumes that costs for vertices dispatched to inadequate processing nodes will amortize over time through duplicate scheduling and speculative execution. We claim that there is simply no room on an ad-hoc cluster to exercise these techniques, and that we must be smarter in advance about where to dispatch vertices.

Worse, known approximation bounds for greedy schedulers do not hold as Dryad is a non-preemptive system—vertices dispatched cannot be preempted and, unless in a FIFO pipeline, only one vertex can run on a cluster node at a time. In general, we are not aware of a bound for greedy approximation algorithms for this class of scheduling problem. Figure 3 shows a case where a precedence relation of four tasks (on the left) under a greedy scheduling policy yields a result (top right) in a heterogeneous environment which is considerably worse than the optimal schedule (bottom right).

2.5 Quantitative motivation

The potential for improvement over greedy scheduling in small clusters is also illustrated in Figure 4. In this experiment we generated 50 random, heterogeneous clusters of a given size, and simulated running a relatively simple Dryad query, an equijoin of two 250MB text files.

We run the simulations using both Dryad's default greedy scheduling algorithm and a more intelligent, performance-aware scheduler based on the techniques in this paper. Note that the comparison in this Monte Carlo simulation is solely comparing scheduling approaches. TCP and FIFO vertex pipelines are disabled.

On 2-node clusters the benefit of intelligent scheduling is low, as there is very little room for resource reallocation. Results for 10 nodes, however, suggest dramatic opportunities for performance improvement. Results for 40 node clusters show the benefits decreasing as opportunities for on-line speculation increase with cluster size. Note also that these results do not include runtime of our optimizer, which is negligible at the 10-node scale but becomes increasingly complex as the cluster size becomes large.

3 Modelling vertex performance

We have shown above that a simple, greedy method for placement of Dryad vertices can result in poor performance on small, heterogeneous clusters. In this section we describe a performance model that takes a reference trace of the execution of the vertex of interest and predicts its running times on different hardware with different sizes of input. Accurate performance prediction of individual vertices supports a more sophisticated approach to scheduling the program's dataflow graph, and we describe our system for automated intelligent scheduling in the next section.

3.1 Vertex behavior

Dryad vertices are implemented so as to maximise system throughput. All I/O operations are asynchronous and typically pipelined at least 4 deep, and each read and write is normally 256KB. In addition, Dryad tends to perform reads and writes to the same channel in large batches (usually 256MB) to avoid disk seeks. In combination, these techniques result in good memory-system and disk performance.

Usually only a single vertex is run at a time on each node in the cluster so it has exclusive access to all the

CPUs on the machine. For fault tolerance reasons, vertices normally communicate via files on disk and producers and consumers are therefore serialized – in the event of a vertex failure, the input channels can simply be reread by a replacement vertex. Each vertex writes its output to the local disk, and child vertices access data on parent nodes via a standard remote file system protocol.

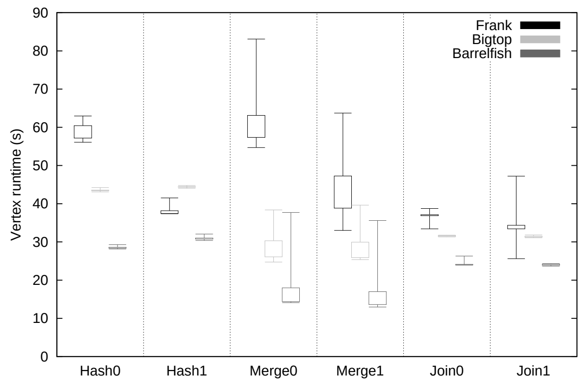


Figure 5: Variance in vertex runtimes for *Join*.

The pattern of I/O and computation exhibited varies depending on the DryadLINQ operator implemented by the vertex. Some of the common operators are as follows:

- `Select` is the DryadLINQ equivalent of “map” in MapReduce. It reads a single input channel in large chunks, applies a function to each record, and writes the results to a single output channel in 256MB chunks.

- `HashPartition` takes a single input channel and distributes its records across a specified number of output channels according to a user-supplied hash function. The input is processed a buffer-at-a-time, and each output channel is written to whenever a buffer fills up. Note that all output channels are written to the *same* local disk.

- `Where` reads a single input channel, one buffer at a time, and outputs a subset of the records matching a user-supplied predicate to a single output channel. Output is written to disk a buffer at a time.

- `Merge` can have an arbitrary number of input channels, but has only a single output, and is used to combine multiple inputs for other operators which expect a single input channel. Inputs are processed sequentially, writing each record to the output unmodified.

- `OrderBy` processes a single input channel, reading buffers of records and quick-sorting each buffer in-memory. The resulting buffers are merge-sorted using a tree of `MergeSort` operators. Each `MergeSort` consumes its inputs in an interleaved fashion, writing the output channel each time a buffer is filled.

- `Join` reads two input channels and performs an SQL-like join operation to produce a single channel of output records. The user supplies functions to extract join fields from each input record and a constructor for output records. There are serveral implementations of `Join` depending on the sizes of the inputs, e.g. if one input is small then it it is read into an in-memory hash-table and the other input is then processed sequentially. Another implementation behaves much like a mergesort.

Other operators include `TakeN`, `RangePartition`, `Count`, `Sum` and numerous similar aggregation operators.

Vertices show inherent performance variability. Figure 5 shows the run-to-run variation in running time of each vertex in the *Join* query of Figure 2 when executed on three different clusters. The error bars show the minimum and maximum of 10 runs and the box shows the 25th and 75th percentiles. For clarity we omit vertices with runtimes of less than 2 seconds. The plot demonstrates that vertex runtimes can vary significantly depending on the cluster hardware. We have found that the inherent variability of a vertex’s runtime due to disk contention, file system layout and background activity is typically less than 10%, but outliers can easily account for a factor of two.

The main factor that determines the running time of a vertex is the *bottleneck* demand. This might be for the local disk, the network, or the CPU (we currently leave vertex memory requirements as future work). It might seem natural to apply queueing theory to this problem, but there are several reasons this is not straightforward [14].

Firstly, vertices often consume multiple resources simultaneously, overlapping computation with I/O.

Secondly, batching is used to increase throughput which results in a coarse-grained I/O profile where the bottleneck resource changes frequently during vertex execution.[1]

Finally, even the bottleneck resource may not be 100% utilized due to poor overlapping of I/O and computation.

3.2 Performance models

We now describe how we extract vertex performance models from a reference trace that characterizes the

[1]256MB batches of reads and writes typically last for several seconds

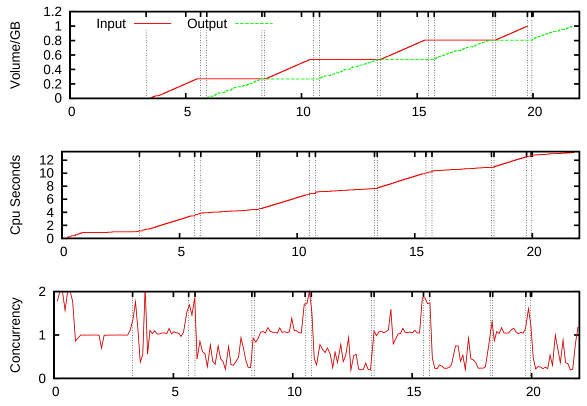


Figure 6: Cumulative resource usage by a `Select` vertex and the inferred phase boundaries.

Type	Run(s)	CPU(s)	CPU%	I/O%	Phases
Init	1.4	1.1	9.4	0.0	1
Read	7.3	7.7	66.0	100.0	4
Write	8.8	1.7	15.2	100.0	4
Comp	1.0	1.1	9.4	0.0	5
Comp	0.2	0.0	0.0	0.0	2

Figure 7: Performance model phase groups extracted from a reference trace of the `Select` vertex. The Read and Write phase groups are each made up of 4 individual phases, as we expect from the vertex behavior shown in Figure 6. Note that CPU time can be greater than runtime due to concurrency.

phases of vertex behavior, and in the following section evaluate its accuracy.

Because Dryad tries to perform channel reads and writes in large sequential batches, the resource demands of a vertex tend to change every few seconds, frequently shifting the bottleneck between CPU, local disk, or remote disk. This means that the runtime of the vertex cannot be computed simply as the demand for a bottleneck resource divided by its capacity, since the vertex as a whole is not bottlenecked on any single resource.

In our models we therefore to divide vertex execution into short “phases” within which resource demands are consistent and operational laws can predict the rate of progress. The predicted total runtime of the vertex is then the the sum of execution times for each phase. In practice, many phases have similar resource demands, and we can merge these into a single, larger phase without affecting the vertex runtime.

To detect behavioral phase changes, we exploit the I/O batching in Dryad to distinguish between periods when I/O is taking place on a channel and when it is not. From a trace of the vertex executing on reference hardware, with fixed input size, we extract cumulative statistics for CPU usage and bytes read and written on each Dryad channel (each of which may be associated with either local disk or network). Changes in gradient in the curves plotted from the cumulative read/write bytes indicate phase boundaries. The resource demands per data record are computed for each phase individually.

An additional subtlety in modelling a DryadLINQ vertex is the potential for concurrency. A vertex that can take advantage of multiple processors may execute significantly faster on a multi-core machine than one that cannot. Unfortunately, this capability can’t be directly measured by tracing vertex execution, and so we approximate it by tracking how many threads are runnable at any time and representing CPU demand as a histogram of the amount of computation which could be performed with each degree of concurrency.

The top graph of Figure 6 shows the cumulative data read and written against time for a `Select` vertex processing 1GB of data. The middle plot shows the cumulative CPU seconds consumed by the process, and the bottom the average number of runnable threads, which gives an indication of the degree of concurrency. Dotted vertical lines show inferred phase boundaries.

We can see that, in this execution, I/O occurred in chunks of 256MB and computation overlapped with reading the input. In general, however, the ordering and interleaving of phases within a vertex can vary from one run to another and so we group similar phases and the vertex model uses the average resource demands and utilizations of each of these phase groups. In the `Select` vertex shown above, all the read phases end up in a single group, while the write phases fall into two separate groups (the concurrency potential is different in the final write phase) and there are three compute phase groups with distinct processor utilizations.

By predicting the running time for each phase separately, we are able to deal with two important challenges. Firstly, the bottleneck resource for a given phase may vary with different hardware. For example, both the read and the write phases in the `Select` vertex are bottlenecked on the disk. With a faster disk and the same processor, we find that the read phases become bottlenecked on CPU, but the write phases remain disk-limited.

Secondly, Dryad’s channel abstraction hides whether the underlying data is stored locally or fetched remotely. Dividing the vertex execution into phases allows us to vary the characteristics of the hardware device behind individual channels, for example by replacing local inputs or outputs with remote files. This functionality is critical for the scheduler we describe in Section 4 to effectively plan execution for the complete dataflow graph

Label	Read	Write	CPU	I/O	Input	Predicted(s)	Avg actual(s)	Avg %error(s)
Reference	140	128	2.66 * 8	1.0	local	20.6	21.1	5.4
Half size input	140	128	2.66 * 8	0.5	local	11.5	11.1	3.6
Slow disk	42	42	2.39 * 2	1.0	local	52.2	48.5	7.6
Remote input	11.5	128	2.66 * 8	1.0	remote	20.5	22.6	9.6
Slow cpu, fast disk	210	180	2.00 * 4	1.0	local	21.2	19.8	6.8
Slow cpu+disk,remote	20	20	1.83	1.0	remote	91.3	91.9	0.7

Figure 8: Prediction accuracy of the `Select` performance model for various combinations of hardware, input size, and local vs remote input data. The "Avg actual" and "Avg %error" columns report the average over 10 runs. The units for Read and Write are MBps, the CPU speed is in GHz and the I/O volume is GB.

Label	Read	Write	CPU	I/O	Output	Predicted(s)	Avg actual(s)	Avg %error(s)
Reference	140	128	2.66 * 8	1.0	local	20.7	18.7	9.9
Half size output	140	128	2.66 * 8	0.5	local	10.3	10.1	1.9
Slow disk	42	42	2.39 * 2	1.0	local	51.6	48.8	5.7
Remote output	11.5	128	2.66 * 8	1.0	remote	18.1	29.6	38.9
Slow cpu, fast disk	210	180	2.00 * 4	1.0	local	18.8	16.7	12.6
Slow cpu+disk,remote	20	20	1.83	1.0	remote	79.9	91.9	13.1

Figure 9: Prediction accuracy of the `Merge` performance model for various combinations of hardware, output size, and local vs remote output data. This `Merge` vertex has just one input and one output stream. The "Avg actual" and "Avg %error" columns report the average over 10 runs. The units for Read and Write are MBps, the CPU speed is in GHz and the I/O volume is GB.

representing the program.

3.3 Building and using the model

The first step in constructing a performance model is to run the application on the reference hardware while tracing CPU, disk and network usage, and the number of runnable threads. We use the Windows tracing facilities; similar tools exist for other operating systems.

We assume that our target applications are run more than once, and that the cost of building models is outweighed over time by overall reduction in runtime. We do expect the behavior of vertices running in a given application will be specific to that application – the model shown in Figure 7 would be quite different for the same vertex in a different DryadLINQ program.

We parse log files to extract CPU and I/O profiles, and derive phases as above. We also record the hardware specification of the reference machine and the potential target machines (number of CPUs, speed, local and remote I/O throughput, and inter-machine network bandwidth).

The vertex models and hardware specification are all that is required for a scheduler (described in the next section) to compute an efficient schedule at runtime. The scheduler can predict the runtime of any vertex on a particular computer with a given I/O channel configuration by simply summing predictions for each phase in the model. The runtime of each vertex's initialization phase is taken unmodified from the trace. For each other phase, the predicted runtime is the larger of the predicted CPU duration and predicted I/O duration. The latter is given by the size of data divided by the disk or network bandwidth available to read or write it.

The predicted CPU duration is scaled from the reference model by the available concurrency, number of processors available, and CPU speed on the target machine. For I/O phases, we scale this by the relative size of the input or output. For non-I/O phases, we add the time spent blocked in the reference trace.

3.4 A worked example

In this section we illustrate the derivation, and explore the prediction accuracy, of a performance model for the DryadLINQ `Select` operator.

We took a reference trace for this vertex with 1GB of input on a Dell Precision T5400 with two Intel Xeon quad-core processors at 2.66GHz, 16GB memory and 2 Hitachi Deskstar 7200RPM disks configured as a striped volume. Figure 6 shows the vertex behavior captured by the trace, and the performance model generated by our tools is shown in Figure 7. As expected, the "Read" and "Write" groups are each made up from 4 distinct phases identified in the raw data, while the 7 "Compute" phases cluster into two groups with different utilization characteristics. In general we have found that phase resource demands cluster well, models rarely contain more than 4-5 clusters.

The "Init" phase characterizes the CPU consumption prior to any input being read, since it is important to distinguish between CPU consumption that scales with the number of data records and that which does not. We found that the duration of this initialization phase varies depending on a number of factors, including the speed of the network link to the Dryad job manager and whether the node has already cached some of the Dryad libraries. However, the duration of the initialization phase is typically small compared to the total runtime of a vertex. In the experiments we present here we take the mean of this duration over several runs on the same hardware. A better model for this portion of the vertex execution is left as future work.

Figure 8 shows results from an evaluation of the prediction accuracy of this model for various permutations of disk speed, CPU speed, size of input, and whether the input file is on the local disk or fetched over the network using SMB. Each experiment was repeated 10 times.

The top line shows the results of self-tests against the 10 reference traces—in other words, how accurately does the model describe the behaviour from which it was constructed. This gives an error of less than 1 second (5.4%). The remaining rows in the table show the prediction together with the average runtime from 10 executions on the various hardware combinations and the corresponding average error. These results are encouraging, with the largest error of 9.6% being a remote read.

Not all vertex types are as amenable to accurate prediction, with the `Merge` vertex being one of the hardest to predict. Results in Figure 9, show the challenges of reliably predicting remote file access performance, with almost 40% error for this case.

Although we cannot show figures for other vertex models for space reasons, Figure 10 shows model accuracy for the several other vertices both for the self-prediction against the reference trace, and for execution on the low-spec laptop. The reference traces were taken on the same computer used for the `Select` reference trace above.

Vertex	Machine	Predicted	Actual	% error
`OrderBy`	Reference	26.6	26.5	0.5
	Laptop	136.4	121.9	11.9
`Merge`	Reference	18.7	20.7	9.9
	Laptop	112.4	111.9	0.4
`Hash Partition`	Reference	18.5	20.2	8.1
	Laptop	116.8	113.9	2.55
`Join`	Reference	25.4	24.3	4.39
	Laptop			

Figure 10: Prediction accuracy of a selection of vertex types on the reference machine and on a low-spec laptop. Predicted and actual times in seconds. The predictions are obtained using the best of 10 reference models (according to self-test accuracy). The "Actual" and "%error" columns show the average over 5 runs on the laptop and over 10 on the reference machine.

3.5 Discussion

The performance of some DryadLINQ operators are highly data dependent, for example `Where` and `Join`. The output of `Where` consists of a subset of the input records which match a user-supplied predicate, which could clearly range from 0 to 100% of the input size (often termed the *selectivity*). Our present system makes the simplifying assumption that the selectivity of `Join` and `Where` vertices is not significantly different from the reference workload used when building the performance model.

Similar data-dependent issues can arise with skew in the `Join` and `HashPartition` operators. For a `Join`, the output size depends on the number of matching rows, and indeed DryadLINQ is sometimes able to choose among different implementations of `Join` based on the relative sizes of the two input channels. For a `HashPartition`, it is possible that records are not distributed evenly across the output partitions. This is a common cause of inefficiencies in a DryadLINQ program since one partition of the parallel computation can end up taking much longer than the others, blocking downstream vertices.

In a relational database, these issues are addressed either by keeping histograms of the values in each table and using them to estimate the query selectivity, or using sampling. In our system the selectivity of such vertices is a "hidden variable" which can be estimated at runtime and an updated plan used to determine whether it is worth reconfiguring the Dryad graph. Additionally, we intend to investigate the use of historical data from previous runs of each binary to inform the planner. Unlike a traditional database, the semantics of Dryad channels make it straightforward to abandon parts of a query execution and restart them elsewhere.

Note that the accuracy of these results is in part due to the underlying hardware having consistent performance characteristics, which is unlikely to be the case in a live system. We looked at how disk reads and writes can vary using the `diskspd.exe` tool[2] and observed that background activity can perturb disk throughput by as much as 19% for reads and 28% for writes. In an ad-hoc clus-

[2] Available from http://research.microsoft.com/barc/sequential_io/.

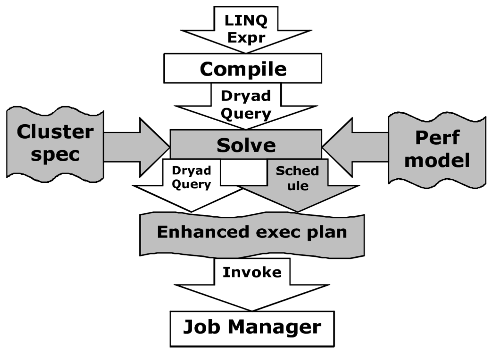


Figure 11: Modifications to Dryad and DryadLINQ (grey background). Square boxes represent computation, swirly boxes are files.

ter running Microsoft Windows, processes such as search indexers or anti-virus software will be accessing the disk even on "idle" computers. We therefore expect a prediction error of up to 30% for real DryadLINQ programs.

4 Scheduler

We now describe how our system uses the models we describe in Section 3 to create a *schedule* for a DryadLINQ program: an assignment of Dryad vertices to machines in an ad-hoc cluster, and an order in which to execute the vertices.

Figure 11 shows the changes we made to both Dryad and DryadLINQ to exploit vertex performance models on ad-hoc clusters. Aside from the model-building process, most of the changes are confined to DryadLINQ. The scheduler is embedded into the DryadLINQ compiler at its final stage to generate an optimized execution schedule that incorporates the vertex models together with a specification of the destination cluster configuration and baseline numbers for our vertex performance models.

We describe in Section 4.1 how the scheduler optimizes the execution plan; the optimized plan is then passed to Dryad's job manager. We also modified the job manager to allow the optimizer to strictly specify the location of each vertex, overriding the job manager's default scheduling behavior.

The scheduler is prototyped entirely as a constraint logic program, using the ECLiPSE CLP system. ECLiPSe [2] is an open-source Prolog-based constraint logic programming system, including several solver libraries, often used for rapid prototyping of solvers. While considerably more sophisticated constraint solvers exist today, we chose ECLiPSe because it allowed us to experiment with different solving techniques before settling with the branch-and-bound method we describe below. In a production system, a working solver and performance model in ECLiPSe code could be re-implemented in a straightforward manner directly in the DryadLINQ compiler, removing the need for ECLiPSe.

We embed ECLiPSe into the DryadLINQ query compiler as a dynamically loaded library. A user-provided cluster specification is given in Prolog, which contains the cluster topology and baseline performance figures for nodes. The DryadLINQ compiler's in-memory representation of the query is translated into Prolog for input to the solver, and the optimized schedule (dispatch times and processing node placement information for the query vertices) is translated back into DryadLINQ's internal representation.

4.1 Planning algorithm

The solver searches for a schedule by enumerating schedules and their cost. The problem of optimally scheduling a set of tasks onto a set of heterogeneous processors has been shown to be NP-complete [4, 6] and we are not aware of competitive bounds shown for greedy schedulers in heterogeneous scenarios, like the ad-hoc cluster, as Dryad is a non-preemptive system. Worse, the variety of channel types in Dryad significantly grows the search space.

Wherever possible, we reduce the size of the search space by exploiting symmetries introduced when there are multiple machines with the same performance, or multiple identical vertices – each of which would result in a exponentially growing number of equivalent schedules with identical runtimes.

Instead of trying to find an optimal schedule or deriving a heuristic, we utilize a constrained, depth-first, limited backtrack branch-and-bound algorithm to approximate the optimal schedule [6]. We make use of the constraint-solving features of ECLiPSe to prune the majority of the search space. Intuitively, the solver works by maintaining a set of inequalities, such as "the start time of Vertex2 must be greater then the start time of Vertex1 plus the duration of Vertex1".

As the schedule is incrementally constructed, each assignment of a vertex to a node, and each choice of connection strategy results in additional constraints being added to the solver's database – for example, two vertices that communicate via the file system must be run sequentially.

These constraints progressively refine a lower bound on the execution time of the complete schedule and allow ECLiPSe to prune regions of the search space as soon as it can "prove" that they do not contain a better solution than the current best candidate.

Before running the constraint solver, we first construct a schedule using Dryad's greedy algorithm and give the

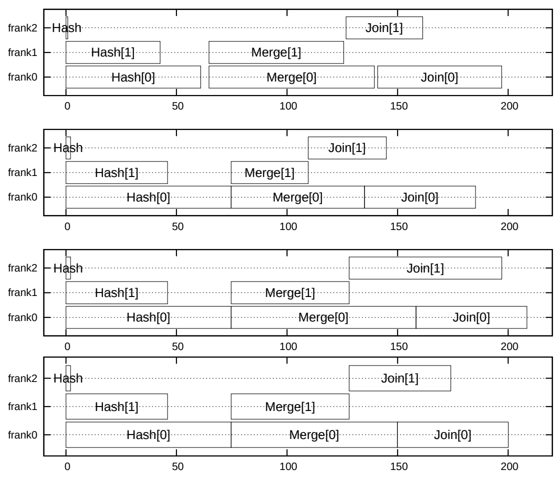


Figure 12: Modelling contention between vertices.

solver an additional constraint that it must find a schedule with a lower cost. This upper bound on the solution cost dramatically reduces the search space, and for most DryadLINQ programs results in the optimizer finding a better solution within milliseconds. In conjunction with a time limit on the optimizer (currently 10 seconds), this allow us to trade-off the time spent searching for a schedule against the potential speedup of the DryadLINQ program.

We do not address failures, instead assuming they are rare on an ad-hoc cluster and probably fatal. An alternative would be to re-enable Dryad's scheduling policy to handle the remaining schedule greedily.

4.2 Modelling inter-vertex contention

Figure 12 demonstrates the need to model resource contention between vertices. The top plot shows an actual execution[3] of *Join* on the *Frank* cluster using the standard Dryad greedy scheduler. The second plot shows the execution timings predicted by our vertex performance models when each vertex is run in isolation using the node assignment chosen by Dryad. The runtimes of the two `Merge` vertices are under-predicted due to contention reading the outputs of the upstream `HashPartition` nodes (each `Merge` vertex reads an input channel from *both* `HashPartition` vertices).

The third plot shows the predicted execution times using a simple contention model where the runtime of a vertex is computed once when it is started, but taking into account the other vertices which are already executing. Note that the runtime of `Join[1]` vertex is massively overestimated since the `Merge[0]` vertex is still running

[3]For clarity these plots omit vertices with execution times of less than one second.

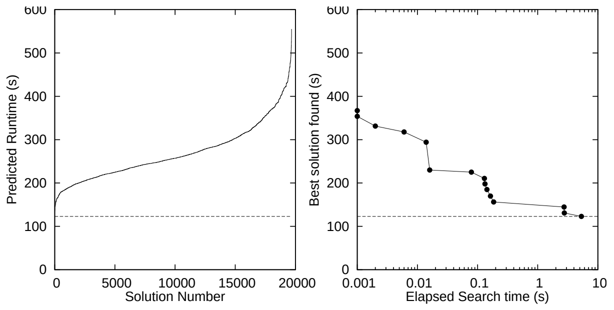


Figure 13: Search space for the *Join* workload.

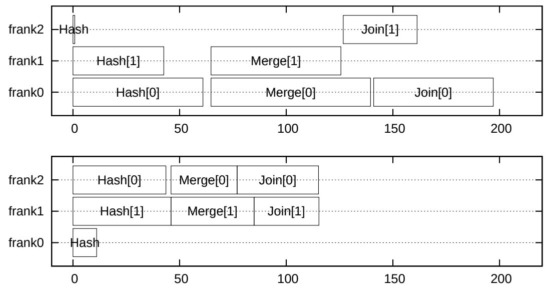


Figure 14: Greedy vs. optimal schedules for *Join*.

when it begins execution. To obtain a good estimate of the runtime of vertices it is necessary to re-evaluate the effects of contention (and hence the per-resource demands) each time a vertex is created or destroyed. The bottom plot shows the predicted vertex timings when this technique is used.

4.3 Search Space

Figure 13 shows the results of an exhaustive search over the space of all possible work-conserving schedules for the *Join* application – with nine vertices and a three node cluster this search space is small enough ($3^9 = 19683$) to be enumerated in a reasonable time. The left hand plot shows the distribution of solution costs, ranging from 115 seconds to 555 seconds with a median of 256.

The right hand plot shows the cost of solutions found by the constraint optimiser as a search progresses. After 200ms the optimiser has found a solution within 27% of the optimum, and finds the optimal solution after 5.3 seconds, taking a further 3 seconds to prove there are no better solutions.

With the ability to accurately predict the execution time of a query with an arbitrary node assigment, it is now possible to search over the space of all possible node assigments to find the assigment which results in the shortest query execution time. For the *Join* query, Figure 14 shows the best execution plan (115s) alongside the default greedy schedule (197s), an improvement

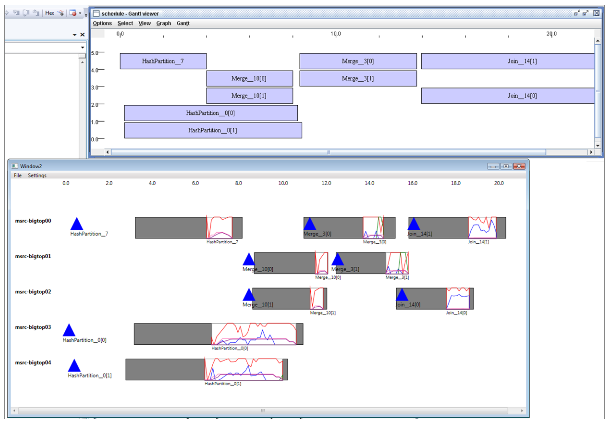


Figure 15: Vertex Execution Analysis: Solver-generated schedule at the top, *perfview* live execution analysis at the bottom.

of over 40%.

4.4 Visualization

In order to analyze and compare the performance model predicted schedule with a live execution on a real cluster, we developed *perfview*, a tool to gather the numerous scattered DryadLINQ execution logs from the execution cluster and present it in a graphical way, directly comparable to the graphical output of the ECLiPSe solver visualization. Both tools output schedules as a Gantt chart with time on the x-axis and processing nodes on the y-axis, as well as additional overlaid information.

This allows us the compare the predicted schedule with a live execution very easily and spot errors in the performance model. *perfview*'s way of overlaying Windows performance counters on-top of vertex execution time rectangles in the Gantt chart allows for precise analysis of I/O and CPU performance bottlenecks.

Figure 15 shows a screen-shot of a solver-generated schedule, along with live execution analysis of that same schedule on a real cluster using *perfview*.

5 Evaluation

We evaluate the performance of our approach by comparing our intelligent scheduler with the Dryad greedy scheduler using three workloads on an ad-hoc cluster.

5.1 Workloads and clusters

Our workload queries were chosen using several criteria. Firstly, they have exhibit varying I/O and processing characteristics – some are strongly I/O bound, whereas others are more processor intensive. Secondly, between them they exercise a representative fraction of DryadLINQ vertex types. Finally, they are representative of the kinds of applications users of ad-hoc clusters might wish to deploy.

Our first workload is the "*Join*" query from Figure 1, which performs a join of two files of total size 1GB consisting of text lines representing key-value pairs. The Join query also reads an additional `keys.txt` file, containing 8 keywords to search for in the input file.

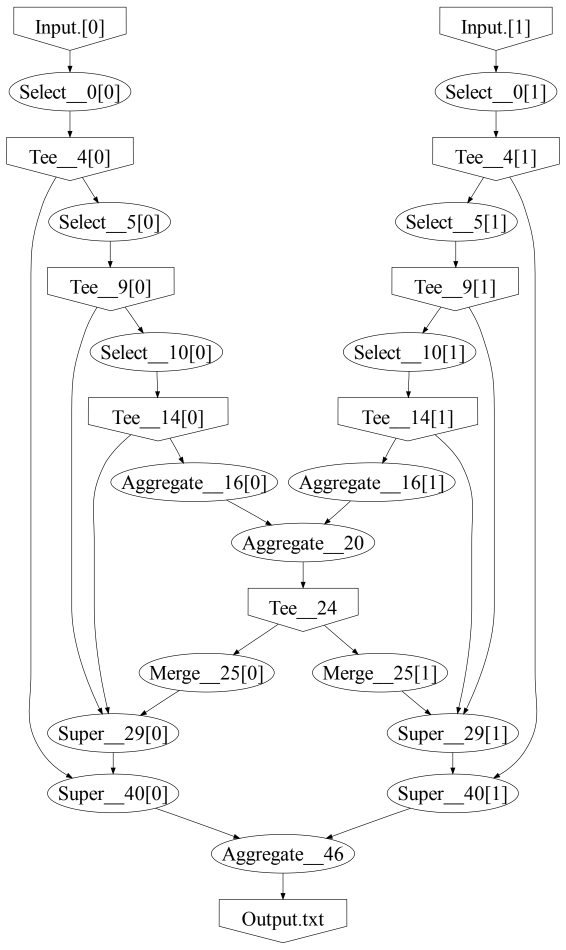


Figure 16: Dataflow graph of the *Algebra* query.

Figure 2 showed the Dryad graph of the *Join* query. Join has 14 vertices and 15 arcs. 5 of the vertices are passive (implemented as files of static tables instead of sequential programs and shown as triangle-shaped vertices in the figure). The maximum breadth over each level of the DAG is 4, which gives an indication of the degree of potential horizontal parallelism. The depth of the portion of active vertices (oval-shaped in the Figure) in the DAG is 3, giving an indication of the maximum pipeline length. Aside from the input and output tables, which we will omit in the following, the implemented vertex types are `HashPartition`, `Merge` and `Join`.

The second workload, "*Terasort*", sorts and merges two specially generated random input textfiles, consisting of key-value pairs, by their key. This is the same query as presented in the original DryadLINQ paper [23] and the workload has been generated accordingly. Terasort has just 3 vertices and the maximum breadth is 2 and the maximum pipeline length is 2. Terasort uses vertex types `OrderBy` and `Merge`. A graph of the *Terasort* query can be found in [23].

The third workload, "*Algebra*" (Figure 16), reads a 40MB text file containing a vector of values and conducts statistical computations on this vector, such as calculating the average and the sum of the vector. The query graph has 26 vertices and 32 arcs; 10 of the vertices are passive. The maximum breadth is 2. The first half of the query is not vertically parallelizable, but the second half has two sections of maximum pipeline length of 2 and 4 vertices, respectively. Implemented vertex types are several versions of `Select Tee`, `Aggregate`, `Merge`, and some composite ("`Super`") vertices performing the CPU-intensive analysis.

Figure 17 describes the configuration of the ad-hoc clusters used in this paper. "Barrelfish" and "Bigtop" are homogeneous, while "Frank" and "Hettie" are heterogeneous.

5.2 Performance Evaluation

In this section we present results for the Frank cluster comparing the runtimes of the three workloads executed 10 times with the original Dryad scheduler [4] and 10 times with the schedule derived by the solver. We measure the individual vertex runtimes using instrumentation software at each cluster node and calculate the runtime of the final schedule using the maximum over all finishing times.

The runtimes of the greedy vs intelligent schedules are shown in Figure 18. The evaluation reflects what we have seen in our simulations, presented in Section 2: Our intelligent scheduling approach is able to improve the runtime by up to 39% for the "Algebra" workload on average. In the best case (max for Dryad, median for our solution), we are able to yield an improvement of 53%, again reflecting the simulations. The file is two-way partitioned and we have 2 fast processing nodes in the cluster, which the solver selected. This is a good example of a case where Dryad's greedy scheduler took the wrong decision and was not able to recover as the query is dominated by the compute intensive "Super" vertices.

[4]By the next revision of this paper the single number for Terasort in this column will be replaced with values from the full 10 runs, and more results for the other clusters will be presented.

The "Terasort" workload is heavily parallelizable, exercising the complete set of machines on the "Frank" cluster. As the workload is mostly CPU-bound and no machine is extraordinarily slow, it is realistic to exercise the complete set of machines. In this case, by both schedulers picking all the machines, we only gain a speedup of 8%.

For the I/O bound "Join" workload, we obtain a speedup of 28%. While the query provides room for both horizontal and vertical parallelism, it is highly sensitive to node placement, since one branch of the graph processes 1GB of data while the other processes only 112 bytes of input query. Dryad's greedy scheduler has yielded a semi-optimal result on average by placing some vertices of the data intensive branch onto nodes with slower disks on the "Frank" cluster. In Dryad's worst case, when most vertices are wrongly placed, we are able to yield a speedup of over 40%.

Overall, it is clear that significant speedups can be obtained over the default scheduler, even when the model accuracy is not perfect as is the case with the Join and Terasort workloads. Dryad's naive greedy scheduler will randomly pick good or bad nodes without chance to refute its choices in most cases.

We can also see that for the "Algebra" and "Join" workloads, even our worst-case figures are never worse than Dryad's best-case figures over 10 runs of the queries.

6 Related Work

Our approach combines coarse-grained data-parallel languages like DryadLINQ and Pig Latin [16] with ideas from the fields of performance modeling and heterogeneous scheduling. Both the latter are large fields; we give a brief survey here.

6.1 Scheduling on Heterogeneous Resources

While there is abundant theory literature on scheduling homogeneous and heterogeneous resources, there are few theoretical results for tasks with precedence constraints on non-identical parallel machines [12].

In the field of parallel computing, Feitelson et. al. [9] provide an overview of scheduling results for the homogeneous case. Sabin et. al. [20] is a recent example for the heterogeneous case using trace-driven simulations to adapt greedy scheduling to the heterogeneous case, and using duplicate scheduling to amortize the cost of bad assignments. A dynamic, self-optimizing scheduling algorithm for heterogeneous server clusters based on reinforcement learning is presented by Yom-Tov and Aridor

Cluster Name	Barrelfish	Bigtop	Frank	Hettie
Machines	2	15	3	6
CPU speeds (Ghz)	2.66	3.0	1.8, 2.4, 2.0	3.6, 3.6, 3.6, 2.0, 2.0, 2.8
# of cores	8	2	1, 2, 4	2, 2, 2, 2, 2, 4
Memory (GB)	16	2	2, 2, 4	2, 2, 2, 2, 2, 2
Disk read (MB/s)	140	60	20, 42, 210	56, 56, 73, 58, 58, 52
Disk write (MB/s)	128	53	20, 42, 180	57, 57, 70, 58, 58, 52
Network (Mbps)	1000	1000	1000	100, 100, 100, 1000, 1000, 1000

Figure 17: Configuration of the ad-hoc clusters used in this paper.

Workload	Greedy(s) min/med/max	Optimal(s)	Achieved(s) min/med/max	Speedup(%)
Algebra	114/120/155	73	71/73/87	39
Join	165/171/206	115	114/123/144	28
Terasort	155	127	123/143/204	8

Figure 18: Minimum, median and maximum durations for the default greedy schedule compared with the optimal schedule runtime predicted by the solver and that actually achieved when executing with that solver. Each number reports the median over 10 runs of the 3 workloads on the Frank cluster. The speedup is computed from the medians of the observed performance.

[22], using run-time estimates of resource requirements for parallel compute jobs to optimize resource allocation.

In a scenario quite similar to ours, Peng et. al. [17] present a branch-and-bound scheduling algorithm for optimal assignment and scheduling of communicating periodic tasks with precedence constraints in distributed real-time systems. Greedy solutions are shown to be unsuitable, and branch-and-bound scheduling algorithms provide a powerful solution. Sinclair [21] also presents strong efficiency results for branch-and-bound scheduling in distributed systems.

Constraint solvers have been used for offline scheduling and assignment problems common to high-level synthesis and system-level synthesis [15].

6.2 Performance Modeling

Modellus [8] is a good example of automated modeling of complex data center applications via queuing theory and machine learning techniques. Modellus models predict the resource usage of an application and the workload it generates, and can be composed to capture dependencies between interacting applications. Our performance model builds upon ideas presented in Modellus, while specific to the needs of ad-hoc cluster-based DryadLINQ programs [14].

6.3 Database Optimization

There are some similarities between our work and database query optimization, in particular issues of load-aware operator and data placement between clients and servers [10]. The field of parallel/distributed database query optimization has also developed a variety of techniques largely orthogonal to our work, but still applicable to DryadLINQ queries [5].

In some ways Dryad resembles a distributed data-stream processor, however, in the latter all operators are assumed to be running at the same time, and so the scheduling problems reduce to pure placement issues. Pietzuch et. al. [18] use a string relaxation algorithm to optimize placement of operators on network-heterogeneous nodes under a network usage cost metric.

6.4 Datacenters and Grids

Heterogeneity in data-centers is mainly handled through duplicate scheduling of identical job-fragments on the expectation that execution costs will amortize over these duplicate schedules [7]. Recent work on Hadoop using EC2 virtual machines [24] has used techniques that differ substantially from ours, instead using a duplicate scheduling strategy which pays closer attention to heterogeneity.

Heterogeneous scheduling is also of concern in Grid systems, such as in Condor's Matchmaking [19], although here the scale and goals are rather different. Grid schedulers focus on reliability and load balancing rather than runtime optimization. Scheduling is typically seperated from node assignment to fulfill non-functional requirements such as decentralized ownership, which are an important design factor in Grids. Hamscher et. al. [11]

give a good overview.

7 Conclusion

We have investigated executing data-parallel programs efficiently on the ad-hoc cluster, an environment far more constrained and heterogeneous than regular datacenter clusters and which seems intuitively amenable to performance improvement through intelligent scheduling, utilizing automatically generated performance models. In the context of Dryad, we have shown that is is possible to build performance models of vertices which are remarkably good predictors of node performance across a range of hardware types, and which can be used as a basis for placing and scheduling a Dryad graph on a small, heterogeneous cluster.

We have further demonstrated a complete system which performs such placement and scheduling, prototyped using a constraint logic programming system embedded in the DryadLINQ compiler. Even with relatively simple performance models, the benefits are clear, yielding deployments up to twice as fast as those using greedy schedules. We feel that such savings in resource usage are significant for the kinds in scenarios in which ad-hoc clusters are used today, and in the foreseeable future.

References

[1] Apache Foundation. The Hadoop project. http://hadoop.apache.org/, November 2008.

[2] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2007.

[3] Don Box and Anders Hejlsberg. The LINQ project, September 2005. http://msdn.microsoft.com/en-us/library/aa479865.aspx.

[4] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6), 2001.

[5] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS*, 1998.

[6] E. G. Coffman, editor. *Computer and Job-Shop Scheduling Theory*. Wiley, 1976.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[8] Peter Desnoyers, Timothy Wood, Prashant Shenoy, Sangameshwar Patil, and Harrick Vin. Modellus: Automated modeling of complex data center applications. Technical Report TR31-07, University of Massachusetts, 2007.

[9] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling — A status report. In *Job Scheduling Strategies for Parallel Processing*. 2004. LNCS vol. 3277.

[10] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 25(2), 1996.

[11] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Proc. GRID*, LNCS vol. 1971, 2000.

[12] Jeffrey Herrmann, Jean-Marie Proth, and Nathalie Sauer. Heuristics for unrelated machine scheduling with precedence constraints. *European Journal of Operational Research*, (102): 528–537, 1997.

[13] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, 2007.

[14] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., New York, NY, 1991.

[15] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3), 2003.

[16] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proc. SIGMOD Conference*, 2008.

[17] Dar-Tzen Peng, Kang G. Shin, and Tarek F. Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Trans. Software Eng*, 23(12), 1997.

[18] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. *Proc. Intl. Conf. Data Engineering*, 2006.

[19] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC7*, July 1998.

[20] Gerald Sabin, Rajkumar Kettimuthu, Arun Rajan, and Ponnuswamy Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In *Job Scheduling Strategies for Parallel Processing*. 2003. LNCS vol. 2862.

[21] J. B. Sinclair. Efficient computation of optimal assignments for distributed tasks. *J. Parallel Distrib. Comput.*, 4(4):342–362, 1987.

[22] Elad Yom-Tov and Yariv Aridor. A self-optimized job scheduler for heterogeneous server clusters. In *Job Scheduling Strategies for Parallel Processing*. 2007. LNCS vol. 4942.

[23] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. OSDI*, Dec. 2008.

[24] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous envrionments. In *Proc. OSDI*, Dec. 2008.