Scripting the cloud with Skywriting

Derek G. Murray Steven Hand
University of Cambridge Computer Laboratory

Abstract

Recent distributed computing frameworks—such as MapReduce, Hadoop and Dryad—have made it simple to exploit multiple machines in a compute cloud. However, these frameworks use coordination languages that are insufficiently expressive for many classes of computation, including iterative and recursive algorithms. To address this problem, and generalise previous approaches, we introduce *Skywriting*: a Turing-powerful, purely-functional script language for describing distributed computations. In this paper, we introduce the main features of Skywriting, and outline our novel *cooperative task farming* execution engine.

1 Introduction

Recent frameworks for data-intensive computing have made it much easier to exploit multiple computers in parallel [2, 11, 18]. One reason for this success is that the frameworks provide a high-level interface that frees the user from concerns about sockets, remote procedure calls, data movement and machine failure. Instead, the user specifies the computation in a *coordination language* (such as Sawzall [22], Pig [21], Hive [24] or DryadLINQ [26]) that describes the data flow at a high level. While these languages are useful, there are important classes of algorithm that they cannot express. In this paper, we introduce *Skywriting*: a new coordination language that generalises the existing languages and increases their expressivity.

For example, others have observed that MapReduce and Dryad cannot express iterative computations [13, 17]. Such computations repeatedly apply the same algorithm in order to improve the current result until it meets a convergence criterion: the PageRank algorithm is a well-known example [10]. Since the number of iterations is not known in advance, we need to express *unbounded iteration*, but existing frameworks and languages force the user to specify the entire computation when a job is submitted. Hence an additional driver program running on the client must ship data out of the cluster to evaluate convergence, and, if required, submit multiple jobs: this leads to additional job submission latency. Moreover, dividing a computation into several jobs reduces the opportunity for high-level optimisations [20, 25].

We distinguish between coordination languages and *task languages* for data-intensive computing. The task language is a general-purpose programming language that is used to express an individual step in a larger computation. For example, the normal task language in Hadoop is Java, which is used to implement the `map()` and `reduce()` functions, but other languages may be used [2]. The task language is usually Turing-powerful. By contrast, existing coordination languages are less expressive, as they only describe a static, acyclic graph of tasks and their dependencies.

In order to add unbounded iteration to the coordination language, we must make it Turing-powerful. As a proof-of-concept, we have developed the Skywriting script language (Section 2). Skywriting is a dynamically-typed, purely-functional language, which provides an executable representation of a distributed job. A script can create new tasks asynchronously, evaluate data dependencies and perform unbounded (while-loop) iteration. This enables Skywriting to describe a more general class of distributed computations than previous languages.

We have also developed a new execution engine that can execute Skywriting scripts. Our system must support dynamically growing jobs, and therefore we have developed *cooperative task farming* (Section 3). In a typical task-farming architecture, the master has complete control over the set of tasks that are assigned to workers. Our system allows workers to *spawn* additional tasks in a manner similar to Cilk [9]. Hence the workers cooperate with the master in order to complete a distributed computation. We describe our prototype system, and how it can be used to execute Skywriting scripts.

```
 1 function process_chunk(chunk, prev_result) {
 2   // Execute native code for chunk processing.
 3   // Returns a partial result.
 4 };

 5 function is_converged(curr_result, prev_result) {
 6   // Execute native code for convergence test.
 7   // Returns a boolean.
 8 };

 9 function iterative_alg(data_chunks) {
10   curr = ...; // Initial guess at the result.

11   do {
12     prev = curr;
13     curr = [];
14     for (chunk in data_chunks) {
15       curr += spawn(process_chunk, [chunk, prev]);
16     }
17     converged = *spawn(is_converged, [curr, prev]);
18   } while (!converged);

19   return curr;
20 };

21 input_data = [ref("file://cluster-1-37/input0"),
22               ref("file://cluster-2-23/input1"),
23               ...];

24 return iterative_alg(input_data);
```

Figure 1: Iterative computation implemented in Skywriting. `input_data` is a list of n input chunks (e.g. of a matrix), and `curr` is initialised to a list of n partial results.

2 The Skywriting language

Skywriting is a Turing-powerful language for describing distributed computations. In order to maximise developer familiarity, we have designed our language to resemble JavaScript, with first-class functions, lexical scoping and dynamic typing [12]. However, in order to make the language purely functional, we impose the restriction that captured variables and parameters are read-only. This restriction makes it straightforward to parallelise the execution of a Skywriting script, which we discuss in Section 3.

Figure 1 shows an example of Skywriting syntax for an iterative computation. The `iterative_alg()` function takes a list of input data, makes an initial estimate of the result, and repeatedly spawns tasks to process the data until convergence is reached. The example highlights three novel features of our language, which we discuss in the remainder of this section. The input data are provided as *references* (§2.1). Distributed computation is initiated by `spawn()`-ing tasks (§2.2). Finally, data dependencies are dereferenced using the $*$-operator (§2.3).

2.1 References

Skywriting uses *references* to add a layer of indirection between the scripting language and the data-intensive processing. In the simplest case, a reference is a name for data that is stored in the cluster. It serves the same role as a file name in GFS [14], or a `DryadTable` object in DryadLINQ [26]. This indirection enables the language to describe manipulations of a large volume of data without loading it all into memory.

A reference is created by calling the built-in `ref()` function, which takes one or more URIs that refer to the data (Figure 1, lines 21–23). By providing multiple URIs, a reference can point to several instances of the same data, which enables data replication. Although the `ref()` function allows a script to refer to arbitrary data, we expect that most users will use library functions to manage collections of references, in a similar manner to how a `DryadTable` represents a list of data partitions in DryadLINQ [26].

2.2 Task spawning

The built-in `spawn()` function asynchronously spawns a new task with the given arguments. Its arguments include the function to be invoked, and optional arguments. It returns one or more *future references*, which behave like references (§2.1), but may refer to data that have not yet been computed. This feature is based on explicit futures [15], which become determined only when dereferenced (§2.3).

A task is executed using our cooperative task farming system, described in Section 3. Typically, one or more of the arguments to `spawn()` will be a reference. If any of the arguments is a future reference, the spawned task will not be scheduled until all of the referenced data has been computed by earlier tasks: thus the script implicitly builds an acyclic graph of task dependencies (§3.1). Once the task is runnable, it is allocated to a worker, and the framework ensures that all referenced data are available locally at the worker. We use a multiple-queue scheduler to ensure that a task usually runs "close" to its input, which reduces the volume of data that must be transferred across the network. In the future, we will investigate more advanced scheduling algorithms, including flow network approaches [19], and decentralised work stealing [9].

Task spawning does not compromise the functional purity of our language. As in other systems, we assume that individual task execution—which may involve code written in other languages—is deterministic and idempotent [11, 18]. Furthermore, the execution of a task has no effect on the script's execution context. However, repeated execution of a task with the same arguments may lead to generating two different references to identical data. This will not affect correctness, but reduces the efficiency of the system, by wasting storage and ignoring opportunities for local execution. Therefore, we are investigating ways of memoising task execution, in a manner similar to cached function calls in Vesta [16].

2.3 Dereferencing

Skywriting's most powerful feature is its ability to include *data dependencies* in the computation. As Figure 1 (line 17) shows, the language includes a $\star$-operator, which dereferences the result of a previous task (in this case, a convergence test). When the operator is applied to a reference, the referenced value can be used like any other Skywriting value. In particular, it can be used in loops and conditional statements, which enables data-dependent control-flow. We borrow the syntax from C, because a reference in Skywriting is conceptually similar to a pointer in C and its related languages.

When a variable is dereferenced, the corresponding data must be fetched and brought into the script environment. Since Skywriting was not designed to be a high-performance language, it is impractical to dereference large volumes of data. Therefore, the example in Figure 1 spawns an additional task that combines the partial results from the distributed computation, and returns a single boolean value. In order to dereference the result, the referenced data must have been written in a compatible serialisation format: the present implementation uses JavaScript Object Notation (JSON), due to its similarity with the Skywriting syntax, and the availability of parsers and serializers for many programming languages [4].

Figure 1 shows the $\star$-operator being applied to a future reference (§2.2). Since `spawn()`-ed tasks execute asynchronously, the result may not yet have been produced when the dereference attempt is made. We support this case by *blocking* the current task and scheduling a continuation (§3.4). The $\star$-operator thereby provides implicit synchronisation with other tasks.

2.4 Other features

As we have developed Skywriting, we have discovered a need for additional language features, and we have implemented these features as a set of built-in functions that comprise our "standard library". Here, we briefly summarise the most important additional features.

Skywriting is an expressive coordination language, but it is unlikely to be the most efficient implementation language for data- or CPU-intensive computations. Therefore we allow scripts to execute external code using the built-in `exec()` function. We currently have bindings for C, Java, .NET and UNIX pipe-based programs, and other languages are supported through a plugin interface.

For advanced use, we have added features to Skywriting that allow *introspection* on both the cluster and the currently-running job. For example, a script can call the `task()` function on a future reference to obtain information about the status of a spawned task. We support task cancellation using the `abort()` function. The script can also obtain information about the cluster nodes using the `workers()` function. Hence it is possible to tailor a job's execution to the available facilities in the cluster, by—for example—altering the degree of parallelism to match the number of idle workers.

We also support custom scheduling policies using the `waituntil()` function. The function takes a predicate and a list of future references, and returns when the predicate becomes true. For example, we provide predicates that fire when any, all or m-out-of-n tasks complete, and the user may supply custom predicates as a Skywriting function. The combination of introspection and custom scheduling enables us to implement straggler detection and speculative task execution [11] in pure Skywriting, and we are continuing to investigate more advanced policies.

3 Cooperative task farming

In order to run Skywriting programs, we need an execution framework that can execute jobs with a dynamic number of tasks. To this end, we have developed a "cooperative" task farming system, which distributes work between a pool of worker machines, and allows workers to spawn new tasks.

In this section, we describe the key features of our prototype system. We begin by describing how *task dependencies* are used to express a distributed data flow (§3.1). We then introduce *spawn lists*, which enable workers to spawn additional tasks and modify the workflow dynamically (§3.2). We present two methods for executing Skywriting scripts in our system: a master-based interpreter (§3.3), and distributed execution (§3.4). Finally, we discuss how our system can unify different coordination languages within a single framework (§3.5).

3.1 Task dependencies

The recent major systems for distributed computing—including but not limited to MapReduce [11], Hadoop [2], Dryad [18], Condor [23] and BOINC [7]—form an equivalence class. Each system is capable of executing a collection of tasks in parallel on a network of computers, using the common *task farming* architecture. A *master* (alternatively: JobTracker [2], job manager [18], matchmaker [23] or scheduling server [7]) maintains a queue of tasks, and several *workers* loop forever retrieving tasks, processing them and storing the results.

There may also be dependencies between tasks: typically, this means that one task consumes the output of another. These dependencies may be explicit (as in Dryad's dataflow graph), implicit (e.g. all `map()` tasks must complete before any `reduce()` task begins in MapReduce) or trivial (i.e. a bag of independent tasks in BOINC or

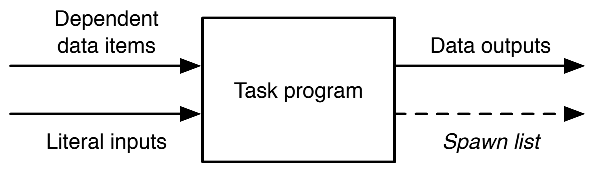

Figure 2: Structure of a task, showing inputs and outputs.

Condor). The dependency graph must be *acyclic*: otherwise the system would deadlock as two tasks would (directly or indirectly) depend on each other [18].

Figure 2 shows the structure of a task in our system. A task depends on one or more *data items*, which may be produced by other tasks: these are equivalent to Skywriting references (§2.1). Therefore a task becomes runnable when all of its input data items are available: data items are decoupled from the task that produces them, which enables additional dynamism in the dependency graph (see the following subsection). The outputs of a task must be named in advance, which allows the scheduler to ensure that dependencies are acyclic.

3.2 Spawn lists

A key feature of cooperative task farming is that a worker may spawn additional tasks in the execution of a job. This requires functionality that is similar to `fork()`-ing a process in UNIX, or spawning a function in Cilk [9]. Therefore, the workers must have a mechanism for accessing the task queue. We use spawn lists for this purpose: as Figure 2 shows, a task may optionally emit a list of task descriptors. The spawn list is a list of tasks to be scheduled after the task has completed, and it is populated *dynamically* at run-time. This feature makes it possible to express unbounded iteration in a task farming system.

For additional flexibility, a task may *delegate* its output to a task in its spawn list. This is particularly useful when implementing recursive, divide-and-conquer computations. For example, consider the case shown in Figure 3. Task A takes a single input, x, and produces a single output, y. However, A may decide that it can split its work into two parallel tasks: B and C. We want A's successors—which are waiting for y to be produced—to see the combined result of B and C. Therefore, A spawns a *continuation task*, D, to which it delegates output y, and which combines the results of B and C.

In order to maintain schedulability, we enforce two constraints: (i) a task must produce all of its outputs or spawn tasks to produce them, and (ii) a spawned task can only depend on the inputs and outputs of the task that spawned it. Together, these ensure that no task will become orphaned and that no cycles will ever form in the dependency graph. Of course, one corollary of delegation (and the Turing-powerfulness of Skywriting) is that a scheduled task may never become runnable, but we do not consider this to be a problem in practice. We prevent "fork bombs" by limiting the number of tasks that can be outstanding for a single job (or user).

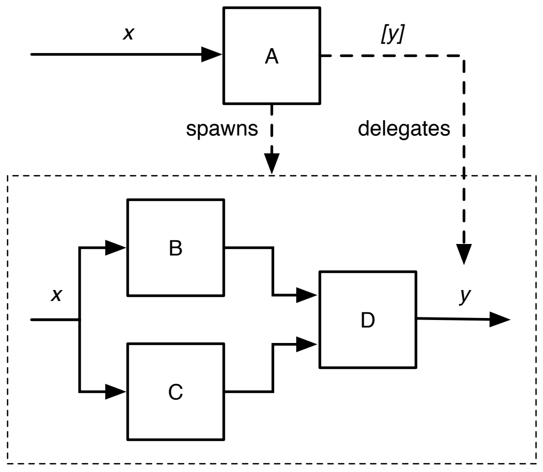

Figure 3: Example of output delegation for recursive tasks. In this example, Task A spawns B, C and D, and delegates its output, y, to D.

3.3 Master-based interpreter

We now turn to executing a Skywriting script on our system. The simplest approach is to run the interpreter in a thread within the master process. This is similar to the approach taken in Dryad [18] and Hadoop [2], which both maintain the graph of dependent tasks in a single process. In this model, the interpreter runs over the Skywriting script until it reaches a `spawn()` function or `*`-operator. At this point, control returns to the execution framework, which schedules a new task or fetches the referenced data, respectively. However, if the script attempts to dereference a future reference that has not yet been produced, the interpreter must block until that result is available.

The main advantage of this scheme is that it is extremely efficient for simple scripts, such as MapReduce- or Dryad-style jobs with no data dependencies. In this case, the interpreter terminates quickly, after spawning the necessary tasks. Since the interpreter and task scheduler are colocated in the same process, the task spawning latency is minimal.

For more complicated scripts that include data dependencies and hence may be long-lived, this approach may place too much load on the master. The master must maintain a thread for each running script, and handle all incoming dereferenced data. It may therefore become a bottleneck if several scripts dereference data concurrently. Furthermore, an uncooperative user could carry

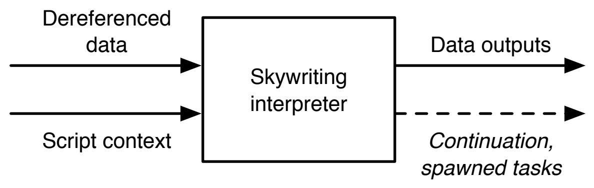

Figure 4: Structure of a Skywriting interpreter task. A continuation is spawned if the task blocks on unavailable data.

out a denial of service attack either by (i) scheduling a CPU-intensive script, or (ii) dereferencing a large volume of data. The following subsection addresses this problem.

3.4 Distributed execution

Interpreting a Skywriting script is a simple computation, and therefore we can run the *interpreter* as a task on the cluster. Since Skywriting is a functional language, we can easily exploit parallelism and data locality in the script environment itself. However, in order to make the interpreter fit within our cooperative task farming model, we must decompose a script into one or more subtasks, each comprising sequences of statements that can run in a single task.

An individual interpreter task takes an execution context (stack and environment) as input, executes one or more statements, and outputs the updated context. As Figure 4 shows, it may have other inputs and outputs:

Dereferenced data Any data dependencies are treated as inputs to the task. This ensures that the task is only executed when the necessary data have become available, and it enables the scheduler to choose the appropriate worker on which to execute the task, for better data locality.

Continuation If the task attempts to use data that are not yet available, it will spawn a continuation task to be scheduled when the data become available.

Other spawned tasks If the task contains any invocations of the `spawn()` function, it will also output a list of tasks to schedule.

The simplest way to subdivide a script is to block and create a continuation every time the $\star$-operator (§2.3) is encountered. The continuation task would then receive the dereferenced data as an input. However, if the script dereferences several data items in succession, this may not give the best performance. Therefore, we delay the creation of a new task until the first time the dereferenced data is used. We achieve this by creating lazily-evaluated thunks for each instance of the $\star$-operator.

Since Skywriting is a purely functional language, we can evaluate independent functions, and even individual expressions, as parallel tasks. However, it is clear that not every expression should be parallelised in this manner. We are investigating simple annotation-based and adaptive schemes as future work.

Distributed execution provides many advantages over a master-based interpreter. It reduces load on the master, which improves scalability. The system handles all necessary data movement, which simplifies the master and gives the potential to exploit data locality. The cluster scheduler can also account for the resources used by the interpreter, which improves resource isolation.

However, because spawning tasks involves network communication with the master, the job latency can increase, which is noticeable for very short tasks. In our implementation, we first attempt master-based interpretation (§3.3), and only switch to distributed execution when the script first attempts to perform a dereference.

3.5 Unifying other frameworks

Cooperative task farming is strictly more general than the task farming schemes used in MapReduce, Hadoop, Dryad and other frameworks. Other authors have remarked that it would be useful to combine multiple frameworks on the same cluster [17]. We concur, and we are implementing compatibility layers for Hadoop and DryadLINQ. Our approach is to transform a Hadoop or DryadLINQ job into the equivalent Skywriting script, and execute it using our cooperative task farming system. We must also provide Hadoop and DryadLINQ driver programs to execute MapReduce tasks and DryadLINQ vertices, respectively. This contrasts with Nexus [17], which uses lightweight OS virtualisation to run existing frameworks in isolated containers. We prefer to unify the frameworks under a common job representation, which raises the possibility of combining Hadoop and DryadLINQ execution in a single job.

4 Related work

Several other projects have investigated new programming models for cloud computing, beyond MapReduce and Dryad.

There have been many projects that add programming language support for data-intensive computing. DryadLINQ uses the Language Integrated Query (LINQ) extensions in recent .NET languages to allow users to specify queries on top of distributed data sets [26]. Yahoo's Pig is a dataflow language that can be used to define a graph of dependent Hadoop jobs [21]. Facebook's Hive provides a more declarative, SQL-like query language that can also generate a plan for Hadoop jobs [24]. None of these languages supports unbounded iteration or recursion as a first-class construct: however, each can be straightforwardly compiled into Skywriting, and we are pursuing this as described in Subsection 3.5.

The Berkeley Orders Of Magnitude (BOOM) project also introduces a new programming model for data-intensive computing: "declarative, data-centric programming" [5]. The authors use the Overlog logic language to reimplement key pieces of distributed infrastructure, such as Hadoop and HDFS. They reproduce previous enhancements to these systems using Overlog, and show that it is possible to add new features (such as Paxos replication [6]) with a small number of Overlog statements. However, BOOM-MapReduce maintains the traditional `map()` and `reduce()` interface to Hadoop, so it does not change the programming model from the user's perspective. Skywriting, by contrast, is an attempt to enhance the programming model for the cluster user.

CGL-MapReduce adds unbounded iteration to the MapReduce programming model [13]. A CGL-MapReduce computation has an additional "merge" stage that combines all `reduce()` outputs together and evaluates a termination condition on them. Therefore, CGL-MapReduce can keep intermediate data in RAM between iterations, if the input data are sufficiently small. However, the programming model is limited to MapReduce-style computation, and it is not possible to compose CGL-MapReduce jobs into larger computations. It is straightforward to express iterative MapReduce jobs as part of a Skywriting script.

Zaharia *et al.* have developed the Spark cluster computing framework, which is optimised for iterative and interactive computation [27]. Spark uses "resilient distributed datasets", which are cached in RAM between jobs, and which can be reconstructed if a machine fails. A driver program—written in Scala—manipulates these datasets through an optimised job submission interface, which provides sufficient performance for interactive applications, and a substantial improvement over the equivalent Hadoop invocations. The main goal of Skywriting has been to obviate the need for a driver program, which enables the whole computation to run in the cluster. This reduces job submission overhead, and improves reliability because the execution engine provides fault tolerance for the whole job. However, Spark—like DryadLINQ—provides the benefits of a strongly-typed high-level language, and we are investigating how to augment an existing language with Skywriting's distributed execution features.

Spark runs on top of the Nexus "cluster operating system" (recently renamed *Mesos*), which shares Skywriting's aim of unifying and extending existing programming models for data-intensive computing [17]. However, Nexus takes an operating system approach that contrasts with our language-based approach. A Nexus cluster can be shared between several frameworks at once, including Hadoop, Spark and MPI. Nexus uses lightweight OS virtualistion to isolate the different frameworks, and a second level of scheduling to allocate resources between the frameworks. Existing framework schedulers then perform task allocation for their individual jobs. Nexus achieves the aim of running many frameworks on a single cluster, but it focuses on sharing physical resources between existing cluster computing frameworks. By contrast, we advocate (automatically) translating computations from different frameworks into the unified Skywriting representation, which makes it possible to mix paradigms (e.g. MapReduce and Dryad) in a single job.

5 Conclusions, status and future work

In this paper, we have introduced Skywriting, a new script language for coordinating distributed computation. Skywriting is more expressive than existing coordination languages, and is particularly suited to expressing iterative and recursive computation. We have also introduced cooperative task farming, which is an extension of task farming that supports dynamically-growing jobs, such as those expressed in Skywriting scripts. Together, Skywriting and its execution engine allow any distributed computation to be implemented in a single job.

We have implemented a prototype of Skywriting that includes all of the features described in this paper. The interpreter and distributed execution engine comprise approximately 4000 lines of Python code. The source code and a tutorial are available from the project website: `http://www.cl.cam.ac.uk/netos/skywriting/`.

We are continuing to develop Skywriting and our prototype execution engine. In future work, we also intend to address the following questions, among others:

- How can we incorporate message-passing computation efficiently in our framework? Other frameworks handle this by maintaining frequently-used data in RAM [13, 27], and we seek general abstractions that can provide this support in Skywriting.
- Can we execute a Skywriting script across multiple clouds? For example, we might store web service data in Google App Engine [3], but want to process it on Amazon EC2 [1].
- Could we achieve benefits by integrating Skywriting with an existing high-level language? Existing frameworks harness the strong typing and library support of mature languages, which further simplifies distributed programming [26, 27].
- Does the Skywriting programming model extend to multi- and many-core systems? As message passing becomes more prevalent, there is a need for new programming abstractions beyond shared memory [8].

Acknowledgments

Thanks are due to our colleagues Malte Schwarzkopf, Chris Smowton and Anil Madhavapeddy, who have contributed to the further development of Skywriting. We would also like to thank the anonymous reviewers for their constructive comments and suggestions, and Matei Zaharia for information about Spark and Nexus.

This work is supported by an AWS in Education Research Grant from Amazon Web Services.

References

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.

[2] Apache Hadoop. `http://hadoop.apache.org/`.

[3] Google App Engine. `http://code.google.com/appengine/`.

[4] JSON. `http://www.json.org/`.

[5] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of EuroSys* (2010).

[6] ALVARO, P., CONDIE, T., CONWAY, N., HELLERSTEIN, J. M., AND SEARS, R. I do declare: consensus in a logic language. In *Proceedings of NetDB* (2009).

[7] ANDERSON, D. P. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th international workshop on Grid Computing* (2004).

[8] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of SOSP* (2009).

[9] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput. 37*, 1 (1996).

[10] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of WWW* (1998).

[11] DEAN, J., AND GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In *Proceedings of OSDI* (2004).

[12] ECMA INTERNATIONAL. *ECMA-262: ECMAScript Language Specification*, 5th ed. 2009.

[13] EKANAYAKE, J., PALLICKARA, S., AND FOX, G. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of eScience* (2008).

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of SOSP* (2003).

[15] HALSTEAD, JR., R. H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[16] HEYDON, A., LEVIN, R., AND YU, Y. Caching function calls using precise dependencies. In *Proceedings of PLDI* (2000).

[17] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., SHENKER, S., AND STOICA, I. Nexus: A Common Substrate for Cluster Computing. Tech. Rep. UCB/EECS-2009-158, University of California, Berkeley, 2009.

[18] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys* (2007).

[19] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of SOSP* (2009).

[20] OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA, U. Automatic optimization of parallel dataflow programs. In *Proceedings of USENIX* (2008).

[21] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of SIGMOD* (2008).

[22] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005).

[23] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: the Condor experience. *Concurrency: Practice and Experience 17*, 2–4 (2005), 323–356.

[24] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTONY, S., LIU, H., AND MURTHY, R. Hive – A Petabyte Scale Data Warehouse Using Hadoop. In *Proceedings of ICDE* (2010).

[25] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *Proceedings of SOSP* (2009).

[26] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of OSDI* (2008).

[27] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *Proceedings of HotCloud* (2010).