A Common Substrate for Cluster Computing

Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ion Stoica
University of California, Berkeley
{benh,andyk,matei,istoica}@cs.berkeley.edu

Abstract

The success of MapReduce has sparked many efforts to design cluster computing frameworks. We argue that no single framework will be optimal for all applications, and that we should instead enable organizations to run *multiple* frameworks efficiently in the same cloud. Furthermore, to ease development of new frameworks, it is critical to identify common abstractions and modularize their architectures. To achieve these goals, we propose Nexus, a low-level substrate that provides isolation and efficient resource sharing across frameworks running on the same cluster, while giving each framework freedom to implement its own programming model and fully control the execution of its jobs. Nexus fosters innovation in the cloud by letting organizations run new frameworks alongside existing ones and by letting framework developers focus on specific applications rather than building one-size-fits-all frameworks.

1 Introduction

It has become increasingly clear that different cloud applications benefit from different programming models. MapReduce [8] is attractive for its simplicity, but this simplicity makes it difficult to express some computations. This led to the development of higher-level, domain-specific abstractions that use MapReduce as an execution substrate, like Sawzall [15] and Pig [14], and of more general programming models that provide their own execution substrate, like Dryad [12]. However, making a programming model too general increases complexity (Dryad is arguably more complex than MapReduce) and decreases the opportunity to optimize programs using application-specific knowledge. We have identified several applications that are hard to express efficiently even in Dryad. We believe that no single general programming model for the cloud will exist. Instead, multiple cluster computing frameworks will emerge, providing various programming models with different tradeoffs. In this context, it is critical to identify abstractions and system constructs that facilitate the development of such new frameworks.

As new programming models and new frameworks emerge, they will need to share computing resources and data sets. For example, a company using Hadoop [3] should not have to build a second cluster and copy data into it to run a Dryad job. Sharing resources between frameworks is difficult today because frameworks perform both job execution management and resource management. For example, Hadoop acts like a “cluster OS” that allocates resources among users in addition to running jobs. To enable diverse frameworks to coexist, Nexus *decouples job execution management from resource management* by providing a simple resource management layer over which frameworks like Hadoop and Dryad can run. Nexus provides a “slot” abstraction, in which frameworks may run “tasks” that do arbitrary work. Along with a mechanism called “slot offers”, the fine granularity of slots lets Nexus achieve more efficient resource sharing across frameworks than would be possible with traditional coarse-grained cluster scheduling systems like Torque [17]. Nexus is analogous to a “cluster hypervisor”: it provides isolation and multiplexing while giving each framework a high level of control over its own execution.

Nexus is beneficial even to organizations that *only* wish to run Hadoop. Two concerns in Hadoop today are scalability and robustness of the master process. Because the master manages *all* jobs running in the cluster (so it can perform resource allocation), it must be highly robust, and must scale to hundreds of jobs across thousands of nodes. Any scheduling logic that goes into the master must be tested extensively, because a bug in the master will crash the whole cluster. Consequently, companies like Yahoo! and Facebook run 6-10 month old versions of Hadoop even though new versions provide significant performance and functionality improvements. If these organizations ran Hadoop on Nexus, they would be able to run separate Hadoop masters for each job, mitigating the scaling and robustness problems. Furthermore, they could even run a stable version of Hadoop for production jobs and a faster but less stable version for experimental jobs simultaneously. The Nexus master can be made robust and scalable more easily than the Hadoop master because it is a smaller codebase, performs simpler logic, and rarely needs to change. In this way, Nexus resembles a microkernel architecture: the component that must be the most reliable (the resource manager) is made as small as possible.

This paper is organized as follows. In Section 2 we show examples of applications to which MapReduce and Dryad are not well suited. Section 3 describes the Nexus architecture. Section 4 describes our current prototype. We survey related work in Section 5 and present a discussion and future work in Section 6.

2 Beyond MapReduce and Dryad

Although MapReduce and Dryad support many applications, these frameworks have limitations. We list four examples of applications for which these frameworks are not ideal. In all four cases, one could envision extending MapReduce and Dryad to better support the listed application. However, this would require extensive engineering and testing to ensure that the changes do not break existing applications, and would increase the complexity of these systems. Nexus provides an alternative: a developer can build a framework more suitable than MapReduce or Dryad for her application, and this framework can share a cluster efficiently with MapReduce and Dryad.

1. Iterative Jobs: Many machine learning algorithms start with a random value for a parameter and repeatedly evaluate a function of this parameter over a dataset to compute a direction in which to move the parameter. Each evaluation can be expressed as a MapReduce job. However, systems like Hadoop launch new worker processes for each job, which must re-read the input data, because they do not know that the jobs are related. In many applications, the amount of data per node is small enough that it would be more efficient to keep it in memory between iterations.

2. Nested Parallelism: As parallel programming libraries for clusters are developed, it becomes attractive to compose them. For example, a machine learning job might want to call a parallel matrix multiplication library from each task. Neither MapReduce nor Dryad support clean nesting of data flows: for example, if a Dryad vertex launches a computation as a second Dryad job, it has to stay running while the second job executes, consuming a node in the cluster. Nested parallelism is expressed more naturally in other programming models, like Cilk [6].

3. Irregular Parallelism: In applications like graph search, the data flow graph is not known in advance. For example, a branch-and-bound algorithm might wish to add objects to explore onto a "work queue" and prioritize them based on how attractive they look. Neither MapReduce nor Dryad allow this level of control over the execution order of tasks.

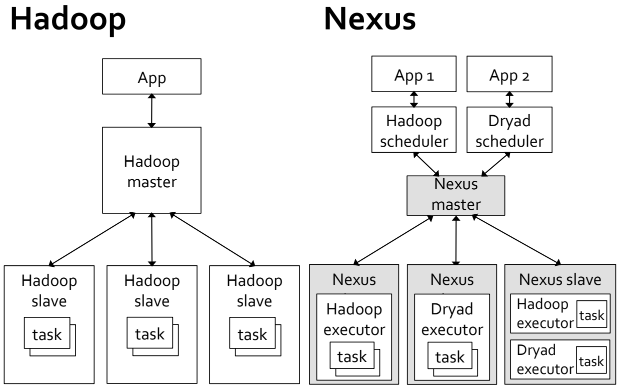

Figure 1: Comparing the architectures of Nexus and Hadoop. The shaded regions make up Nexus. Hadoop can run on top of Nexus, alongside Dryad.

4. Existing Parallel Applications: It would be useful to run parallel applications like `distcc` [2] on the same infrastructure as Hadoop. However, expressing these applications as maps and reduces is awkward. Nexus allows more natural wrapping of these applications into tasks.

3 Nexus Architecture

3.1 Overview

The goal of Nexus is to provide isolation and efficient resource sharing across cluster computing frameworks running on the same cluster. To accomplish this, Nexus provides abstractions of computing resources and an API to let frameworks access these resources. Nexus exports two abstractions: tasks and slots. A *task* represents a unit of work, such as a map task in MapReduce. A *slot* represents a computing resource in which a framework may run a task, such as a core and some associated memory on a multicore machine. We rely on standard OS facilities (e.g. `setrlimit` in Linux) to isolate slots. We are also considering using virtual machines.

Nexus employs two-level scheduling. At the first level, Nexus allocates slots between frameworks using fair sharing. At the second level, each framework is responsible for dividing its work into tasks, selecting which tasks to run in each slot, and as we shall explain, deciding which slots to use. This lets frameworks perform application-specific optimizations.

Figure 1 shows the architecture of Nexus. Like Hadoop, Nexus has a *master* process that controls a *slave* daemon running on each node in the cluster. Each framework that uses Nexus has a *scheduler* process that registers with the master. Schedulers launch tasks in their allocated slots by providing *task descriptors*. These descriptors are passed to a framework-specific *executor* process that Nexus launches on slave nodes. A framework assigned multiple slots on the same node has one executor per slot. Executors are also reused for subsequent tasks run in their slot. This amortizes initialization costs and lets executors cache data shared across tasks in memory which is highly beneficial for jobs like the iterative one described in Section 2. Finally, Nexus passes *status updates* about tasks to schedulers, including notification if a task fails or a node is lost. The elements of Nexus – schedulers, executors and tasks – map closely to components of frameworks like Hadoop and Dryad. Nexus factors these elements out into a common layer.

Nexus purposely does not provide abstractions for storage and communication. We concentrate only on allocating computing resources (slots), and allow tasks to use whichever storage and communication libraries they wish. For example, we expect many sites to install Hadoop's HDFS distributed file system for storage. We do, however, plan to build communication and storage abstractions for framework developers to use on top of Nexus.

3.2 Slot Assignment

The main challenge with Nexus's two-level scheduling design is ensuring that frameworks are allocated slots they wish to run in. For example, a MapReduce framework needs to run maps on the nodes that contain its input file to avoid reading data over the network. Nexus addresses this issue through a mechanism called *slot offers*. When a slot becomes free, Nexus offers it to each framework in turn, in order of how far each framework is below its fair share. Each framework may accept the slot and launch a task in it, or refuse the slot if, for example, it has no data on that machine. Refusing a slot keeps the framework below its fair share, ensuring that it is offered future slots before other frameworks. If the framework has still not found a data-local slot after waiting for some time, it can always accept a non-local slot. While it may seem counterintuitive that refusing slots can be beneficial to frameworks, experience with job scheduling in Hadoop [4] has shown that this simple policy is enough to achieve 99% data locality[1] and high fairness in workloads with dozens of jobs.

In general, slot offers let frameworks use arbitrary criteria for selecting slots without requiring Nexus to be aware of these criteria. For example, while a MapReduce job wishes to run on nodes that have its input data, the iterative job in Section 2 might prefer to reuse its previous slots, to take advantage of data loaded into memory by its executors.

A second concern is how to reassign slots when frameworks' demands change (e.g., a new framework registers). In normal operation, Nexus simply reassigns slots to new frameworks as tasks from overallocated frameworks finish. As long as tasks are short, this is sufficient to redistribute slots quickly. For example, if the average task length is one minute in a 100-node cluster with 4 slots per node is, then there will be about 6.7 tasks finishing per second. If we need to reassign, say, 40 slots (10% of the cluster), we only need to wait 6 seconds. To ensure that frameworks are not starved even when some tasks are long, Nexus can also *reclaim* a slot by killing the task executing in it after a timeout. Nexus reclaims most-recently launched tasks from each framework first, minimizing wasted work. Because frameworks running on large clusters need to tolerate losing tasks due to hardware failures, reclamation does not impact them significantly – it is as if they never received the slot. We are also considering mechanisms for letting frameworks choose which slots they prefer to lose.

4 Implementation Status

We have implemented a prototype of Nexus in approximately 2,000 lines of C and C++. By limiting the scope of Nexus's concerns we were able to focus on optimizing for performance and scalability; for example, we used asynchronous I/O techniques for network communication. A framework registers and unregisters with Nexus using the API presented in Table 1. Table 2 lists the callbacks that Nexus uses to communicate with a framework for task management. Once registered, a framework receives slot offers and task status updates from the Nexus master via the `schedule` and `status` callbacks. The Nexus slave passes a task descriptor to an executor by invoking its `execute` callback. Preliminary evaluation shows that Nexus can schedule thousands of tasks per second, an order of magnitude more than Hadoop. We have also implemented an iterative machine learning job (logistic regression) as described in Section 2 against Nexus, and measured factor of 10 performance gains from executor reuse.

5 Related Work

5.1 Infrastructure as a Service

Cloud infrastructures such as Amazon EC2 [1] and Eucalyptus [13] allow for sharing between users by allowing virtual machines in a shared cloud to be rented by the hour. In such an environment, it would be pos-

Function	Description
`nexus_register(scheduler, executor)`	Register framework's scheduler and executor with the Nexus master.
`nexus_unregister()`	Terminate the current framework.

Table 1: Nexus API.

Function	Description
`schedule(slot_id)`	Invoked when Nexus offers a slot to a framework.
`status()`	Invoked when Nexus conveys information about a scheduled task to a framework.
`execute(task_id, args...)`	Invoked in the slot the framework scheduled the task.

Table 2: Nexus framework callbacks.

sible for separate frameworks to run concurrently on the same physical cluster by creating separate virtual clusters (i.e., EC2 allocations). However, VMs can take minutes to start and sharing data between separate virtual clusters is difficult to accomplish and can result in poor data locality. Nexus provides abstractions (i.e., tasks and slots) that eradicate the need for many applications to use heavyweight VMs. In addition, Nexus allows frameworks to select where to run tasks via the slot offer mechanism, allowing multiple frameworks to share data while achieving good data locality.

5.2 Cluster Computing Frameworks

MapReduce was originally described in [8]. Sawzall [15] and Pig [14] built richer programming models, like relational operators, on top of MapReduce. However, running all jobs over MapReduce can be inefficient: for example, a Pig query that is broken into a series of MapReduce jobs cannot pipeline data directly between them.

Dryad [12] addresses this problem by providing a more general execution model: data flow DAGs. Clustera [9] provides a similar execution model. These frameworks are valuable because they supply common execution abstractions, like worker vertices and channels, to applications. However, the set of abstractions provided is still limited. For example, vertices in Dryad form a DAG, which makes it difficult to write the iterative job in Section 2 where one would like to send a parameter to a worker, have it compute a function, and use this result to compute a new parameter for the same worker. Rather than trying to build an even more general set of abstractions than Dryad, Nexus lets Dryad layer over it, but also enables other frameworks to be written against its lower-level interface of tasks.

5.3 Cluster Scheduling Systems

Nexus differs from previous cluster scheduling systems in two aspects: its slot offer mechanism and the fine-grained nature of slots.

Of the frameworks listed in the previous section, only Clustera [9] describes multi-user support. Clustera uses a heuristic incorporating user priorities, data locality and starvation to match work to idle nodes. However, Clustera requires each job to explicitly list its locality needs (e.g. by listing its input files), meaning that needs that cannot be listed in this manner (e.g. reusing a node to reuse a worker process) cannot be taken into account. In contrast, Nexus's slot offer mechanism lets jobs use whichever heuristics they desire to select a slot.

Batch cluster schedulers like Torque [17] give each job a fixed block of machines at once, rather than letting a job's allocation scale up and down in a fine-grained manner, and do not account for data locality. Grid schedulers like Condor [18] support locality constraints, though usually at the level of geographic sites. Condor uses a "ClassAd" mechanism match node properties to job needs. As in Clustera, this means that job needs that cannot be expressed as ClassAds cannot be accounted for. Nexus gives jobs finer control over where they run through slot offers.

6 Discussion and Future Work

By letting diverse cluster computing frameworks coexist efficiently, Nexus accelerates innovation in the cloud and encourages development of specialized frameworks rather than one-size-fits-all solutions. Nexus is strongly inspired by work on microkernels [5], exokernels [10] and hypervisors [11] in the OS community and by the success of the narrow-waist IP model [7] in computer networks. Like a microkernel or hypervisor, Nexus is a stable, minimal core that provides performance and fault isolation to frameworks sharing a cluster. Like an exokernel, Nexus aims to give frameworks as much control over their execution as possible. Finally, like IP, Nexus encourages diversity and innovation in cluster computing by providing a "narrow waist" API which lets different frameworks run over shared hardware.

We have already written some applications directly against Nexus, and are currently porting Hadoop to run over it. However, to ease the development of new frameworks, we plan to build a stack of higher-level abstractions over Nexus for framework developers to use. For example, if Nexus represents the IP of cluster computing, then a fault-tolerance library providing a concept of "reliable tasks" might represent the TCP. Other reusable abstractions that we intend to build are a data flow DAG library similar to Dryad and a task-queue library similar to Intel Threading Building Blocks [16] for frameworks that wish to employ nested or irregular parallelism.

Using Nexus, we intend to explore both how to provide better programming models for existing cloud applications and how to enable other applications to benefit from cloud computing. Some of the new applications we wish to explore include:

1. **Scientific Computing:** Many scientific applications are not communication-intensive and could therefore run on commodity clusters, but scientists typically write such applications using libraries like MPI rather than MapReduce.
2. **Parallel Build/Test Tools:** It would be beneficial to run build tools like `distcc` and distributed unit testing tools on the same infrastructure as Hadoop clusters to maximize utilization.
3. **Web Servers:** It may even be possible to use Nexus to share resources between batch computing frameworks and interactive web applications. Web servers are stateless, so they can be started and stopped dynamically as "tasks" as long as load balancers are aware of this. This would let an organization use a portion of its cluster for web serving during the day and batch applications at night, improving data center utilization.

Finally, from an operational standpoint, Nexus allows an organization to use multiple cluster computing frameworks without having to allocate separate hardware to each one. For example, if a new implementation of MapReduce appears that is faster than Hadoop on some workloads, an organization can switch a portion of its jobs to it. If an organization wishes to run two isolated instances of Hadoop, or two different versions of Hadoop, it can do this as well. Lastly, if a developer builds a specialized framework optimized for only one type of application, an organization can run this framework alongside Hadoop.

7 Acknowledgments

We would like to thank Randy Katz, Anthony Joseph, Scott Shenker, Krste Asanović, George Necula, and the anonymous reviewers. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, a National Science Foundation Graduate Research Fellowship[2], and the Berkeley RAD Lab sponsors: Sun, Google, Microsoft, HP, Cisco, NetApp, VMWare, Amazon and Facebook.

References

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

[2] distcc: a fast, free distributed c/c++ compiler. http://distcc.samba.org/.

[3] Hadoop. http://lucene.apache.org/hadoop.

[4] Hadoop scheduling discussion. http://issues.apache.org/jira/browse/HADOOP-4667.

[5] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, pages 93–113, 1986.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[7] V. G. Cerf, Robert, and E. Icahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22:637–648, 1974.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] D. J. DeWitt, E. Paulson, E. Robinson, J. F. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995.

[11] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM Systems Journal*, 18(1):111–142, 1979.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, pages 59–72, 2007.

[13] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of Cloud Computing and Its Applications [Online]*, Chicago, Illinois, 10 2008.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[15] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[16] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Mul ti-core PRocessor Parallelism.* O'Reilly, 2007.

[17] G. Staples. Torque - torque resource manager. In *SC*, page 8, 2006.

[18] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation - Practice and Experience*, 17(2-4):323–356, 2005.

Notes

[1] That is, 99% of map tasks read from local disk.

[2] Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.