Non-deterministic parallelism considered useful

Derek G. Murray Steven Hand
University of Cambridge Computer Laboratory

1 Introduction

The development of *distributed execution engines* has greatly simplified parallel programming, by shielding developers from the gory details of programming in a distributed system, and allowing them to focus on writing sequential code [8, 11, 18]. The "sacred cow" in these systems is *transparent fault tolerance*, which is achieved by dividing the computation into atomic tasks that execute deterministically, and hence may be re-executed if a participant fails or some intermediate data are lost. In this paper, we explore the possibility of relaxing this requirement, on the premise that *non-determinism* is useful and sometimes essential to support many programs.

In recent years, the class of problems that distributed execution engines can solve has grown. The original MapReduce system addresses problems that can be decomposed into embarrassingly-parallel map and reduce phases [8]. Dryad extends MapReduce by supporting arbitrary directed acyclic graphs (DAGs) of tasks [11]. We recently developed a system called CIEL, which supports arbitrary DAGs in which a task may spawn a subgraph of tasks, hence enabling unbounded iteration [18]. Therefore, the set of all CIEL jobs is a strict superset of all Dryad jobs, which in turn is a strict superset of all MapReduce jobs. We have previously argued that CIEL is *universal*, since it can execute jobs that are specified in a Turing-complete language [17]. However, this universality merely implies that any algorithm can be implemented on CIEL; it does not necessarily follow that the implementation will be efficient.

Deterministic execution is a conservative requirement, rooted in the assumption that the participants in a distributed execution engine fail frequently. We briefly discuss the fault tolerance benefits of determinism in Section 2. While this assumption may be true for the large-scale clusters on which MapReduce and Dryad were originally developed, it is less obvious for the smaller clusters on which they are often deployed [1].

Our high-level argument is that deterministic execution should not be a requirement at the system level. Deterministic parallelism can be implemented on a system that allows non-determinism [4]; non-deterministic parallelism cannot be implemented on a system that provides only deterministic abstractions [3]. As we show in Section 3, there are several applications that benefit from non-determinism, and we believe that it would be useful to extend the benefits of distributed execution engines to these applications. Furthermore, in our proposed approach (described in Section 4) programs would be "deterministic by default" [5]; we simply extend the set of abstractions in CIEL to allow explicit non-determinism where desirable. If one must limit an individual job to deterministic execution, this can be enforced at the language level, or with a policy in the underlying engine.

In this position paper, we make three contributions:

1. We survey algorithms and applications for which non-deterministic parallelism would be useful (§3).
2. We outline a set of non-deterministic features that can be added to CIEL (§4.2).
3. We describe techniques for dealing with failure during a non-deterministic CIEL job (§4.3).

However, we first explain why deterministic execution has become *de rigeur* for distributed execution engines.

2 Benefits of determinism

Deterministic execution simplifies the provision of fault tolerance. The aim is to allow multiple replicas of a computation to execute (mostly) independently, in order to protect against one or more replicas failing. Bressoud and Schneider showed how a hypervisor could be used to intercept all "environment" (i.e. potentially non-deterministic) instructions and ensure that all replicas agree on the same value [6]. A similar technique was used in dOS, which extends Linux with the ability to run "deterministic process groups" [4].

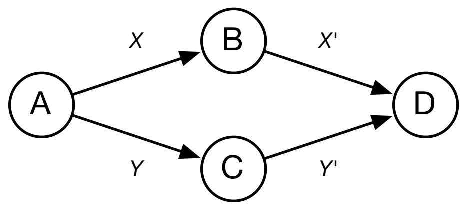


Figure 1: Task A produces outputs X and Y. If one of X or Y must be recomputed (due to failure), the system must ensure that both X and Y are consistent.

Therefore, deterministic execution would appear to be a natural fit for distributed execution engines—such as MapReduce, Dryad and CIEL—which decompose computations into many sequential tasks that the engine executes in parallel. Consider the example in Figure 1. If task A is deterministic, and either of its outputs (X or Y) is lost due to failure, the missing data can be reconstructed by re-executing A. However, if A is non-deterministic (e.g. it randomly partitions elements into two sets for parallel processing), its re-execution may lead to inconsistent results (e.g. there may be elements missing from $X \cup Y$) [7].

3 Applications of non-determinism

Non-determinism can be used to improve the performance of many applications. In this section, we consider two examples: *asynchronous algorithms* (§3.1), which are well-suited to the loosely-coupled clusters on which distributed execution engines typically run; and *adaptive algorithms* (§3.2), which use introspection to make better decisions at run-time. In addition, we discuss the potential for implementing *interactive* jobs (§3.3).

3.1 Asynchronous algorithms

Many data-parallel problems can be solved more efficiently using *asynchronous algorithms*, which relax the synchronisation between processors, and allow each processor to consume data from other processors as the data are produced. Since the processors communicate asynchronously, timing variations cause data to be received in a non-deterministic order, and these variations will be magnified in a loosely-coupled environment such as a data center or cloud computing platform. Here, we consider two algorithms for which allowing non-determinism can improve the overall performance.

PageRank The PageRank (power iteration) algorithm has been studied extensively in the context of distributed execution engines, because it can be applied to very large data sets (such as web-hyperlink and social network graphs) [19, 21]. McSherry showed an elegant approach to computing PageRank, whereby nodes communicate the change in their probability (score) instead of their new value [15]. He also showed that the algorithm converges more quickly without requiring global synchronisation, and is robust to lost messages.

Kambatla *et al.* obtain some of these advantages with “asynchronous MapReduce”, which implements iterative algorithms using a combination of deterministic local and global iteration [13]. However, this approach still suffers from synchronisation overhead in the global barrier between the map and reduce stages. If tasks could be non-deterministic, asynchronous—perhaps even unreliable—messages could replace the barrier, which would further reduce the execution time.

Branch-and-bound The *branch-and-bound* algorithms are commonly applied to NP-hard combinatorial optimisation problems. Since these algorithms are computationally-intensive and contain large amounts of independent work, they are a natural fit for distributed execution engines. As a branch-and-bound algorithm evaluates solutions, it updates *bounds* on the cost of the best solution, which enables large subtrees of the search space to be pruned. In a parallel implementation, the bounds are shared between all processors.

Sharing bounds poses a challenge for a deterministic execution engine, since data can only be communicated at task boundaries [11]. Thus a processor will update its bounds locally, before a synchronous aggregation step that computes the global bounds. Ideally the updated bound would be transmitted as soon as possible, enabling other processors to prune the search space immediately and avoid wasted work. If non-determinism were allowed, a processor could broadcast updated bounds to other processors as soon as they are calculated. Furthermore, Budiu *et al.* have observed that many branch-and-bound algorithms have subproblems that vary greatly in cost, which makes it challenging to balance the work in each partition deterministically [7].

3.2 Adaptive algorithms

The optimal execution plan for a distributed computation is often time-dependent. For example, on a fairly-shared cluster, the maximum degree of parallelism that a single user enjoys will vary as other users submit jobs [12, 22]. Therefore, the optimal task granularity—e.g. the number of map or reduce tasks in a MapReduce-style computation—will vary accordingly. One solution to this problem would be to allow a job to *introspect* on the cluster, and discover the current number of free slots or historical load averages. Of course, introspection is non-deterministic from the job's point of view: querying the current load in the future would probably return a different value.

Timing information could also be used to address imbalance in the workload. For example, there are many straggler detection algorithms that attempt to mitigate long-running tasks that hold up execution before a barrier [2, 8, 11, 24]. The *backup task* strategy—whereby a second copy of an individual task is started on another machine and the two tasks race to completion—relies on non-deterministic behaviour, and hence must be implemented by the execution engine. If the job itself had access to this timing information, it would be able to customise its straggler detection algorithm to be more appropriate for its workload. For example, in the branch-and-bound example where the distribution of computation across tasks is skewed, backup tasks will not improve performance (and will in fact uselessly consume cluster resources); it would be preferable to suspend execution after a time-out expires [7]. In this case, the default policy is undesirable, and non-deterministic abstractions would enable the separation of policy and mechanism.

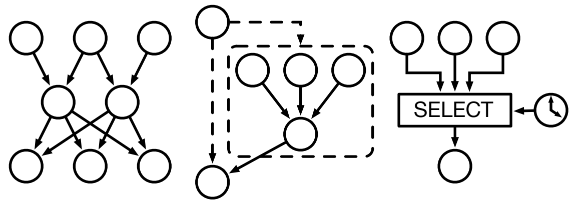
Adaptive algorithms raise a challenge about how to deal with failure, which we will discuss in Subsection 4.3. Simply recording the results of introspection and replaying them may lead to undesirable results. For example, the cluster load may have increased since the first execution, so that using the previously-optimal task granularity would lead to the creation of too many tasks, and, in turn, to an inefficient balance of load.

3.3 Supporting interactivity

Most execution engines were developed for batch-oriented computation [8, 11, 18], but there has recently been interest in interactive querying of large data sets. The state of the art solutions either use a non-deterministic, interactive "driver program"—e.g. a command-line shell—to submit jobs to a deterministic execution engine [23], or a special-purpose distributed system that may return non-deterministic results [16].

In interactive applications, minimising the user-perceived latency is important. However, when the application combines results from distributed resources, stragglers can have a devastating effect on latency. To this end, special-purpose systems such as Dremel and Amazon's e-commerce platform achieve low latency by sacrificing determinism (and, in some cases, consistency). Dremel supports queries that return when any $x\%$ of records have been processed [16]. Requesting an Amazon web page causes many parallel requests to be made on the back-end, but the response must be sent before a hard deadline expires [9].

If computation and interaction can be overlapped, there is an advantage to supporting interactivity in a distributed execution engine. As an example, interactive web applications can use *continuations* to capture the control flow at the points where interaction with the browser is required [20]; Skywriting uses a similar technique to represent parallel algorithms with data-



(a) Static (b) Dynamic (c) Non-deterministic

Figure 2: Example task graphs in CIEL.

dependencies [17]. Many web applications—such as airfare search engines—must perform a large amount of computation and incorporate data from a variety of sources, including user feedback. Therefore, there is potential synergy between continuation-based web servers and distributed execution engines, which an interactive execution engine could support.

Non-deterministic interactive applications are the next frontier for what an execution engine can compute. Though CIEL can simulate a Turing machine, Goldin and Wegner have proposed "interactive computation" as a more expressive model than Turing-computability [10]. The key insight is that Turing machines can compute mathematical *functions* of their inputs, whereas many interactive applications—e.g. consider an operating system or a word processor—cannot be represented as functions.

4 A non-deterministic execution engine

Having motivated the case for non-determinism in a distributed execution engine, we now propose extensions to CIEL that support non-determinism and enable a larger class of applications.

4.1 CIEL recap

CIEL is a distributed execution engine that executes *dynamic task graphs* across a loosely-coupled cluster of commodity machines [18]. For simple jobs, CIEL executes a DAG of tasks and their dependencies (Figure 2(a)) in the same manner as Dryad: i.e. by topologically sorting the tasks according to their dependencies, and executing runnable tasks in parallel [11]. However, CIEL also enables data-dependent control flow, by allowing tasks to *spawn* child tasks and *delegate* production of their outputs to their children (Figure 2(b)). This allows CIEL to execute iterative and recursive jobs.

CIEL jobs are typically specified as Skywriting scripts [17]. Skywriting is a language that facilitates the construction of task graphs with its built-in `spawn()` function. This function returns a *future*, which can be passed to subsequent tasks in order to build a dependency graph. Data only flows via the dependency graph, which ensures that Skywriting is deterministic by default.

4.2 Adding non-determinism to CIEL

To support non-determinism, we propose adding the following features to CIEL and Skywriting:

Non-deterministic references In CIEL, the inputs to and outputs from each task are represented by *references*. Non-deterministic tasks will produce non-deterministic outputs, so it is necessary to *taint* the relevant references (and, by induction, the outputs of tasks that depend on non-deterministic inputs) so that special failure-handling routines may be employed (see §4.3). For example, user interaction could be represented by a task that produces a non-deterministic reference.

Select In a deterministic execution engine, if a task has d dependencies, the scheduler must block that task until all d dependencies are fulfilled [3]. However, it may be possible to begin some work when some subset of the dependencies are available. Therefore, we could add to Skywriting a `select()` function, which takes a set of references and an optional timeout, and returns when at least one of the references becomes available, or else the timeout expires (Figure 2(c)). Using `select()` would taint the outputs of the selecting task as non-deterministic.

Signals To send data to a task asynchronously, we could add a `signal()` function, which takes a reference to a task's output, and an arbitrary data structure. The data structure would be delivered asynchronously to the task. This could be used to implement asynchronous algorithms: a monitor task would `select()` on outputs from tasks, and `signal()` the new data to other tasks. Signals could also be used to abort a task if its output is no longer needed. Sending a signal to a task would taint the outputs of the receiving task as non-deterministic.

Introspection Each piece of information that the cluster exposes would be implemented as a Skywriting function, e.g. `current_load()`. Furthermore, introspection could be used to query the local machine about its capabilities: in a heterogeneous cluster, this would allow a job to specialise its code based on, for example, whether the current machine has a GPGPU or a large amount of RAM. Using any of the introspection functions would taint the outputs of the current task as non-deterministic.

4.3 Handling failure

The main challenge that non-determinism poses is how to deal with failure. If a deterministically-computed object is lost, it can simply be reconstructed by re-executing the task that produced it. However, it may not be safe to do this for an object that is computed non-deterministically (as we discussed in §2). In this subsection, we sketch some of the possible solutions, and discuss their relative advantages and disadvantages.

Record and replay The most conservative approach is to record the non-determinism and deterministically replay it upon failure [14]. The main advantage of this approach is that it enables transparent fault tolerance with no programmer intervention. However, as we discussed in §3.2, naïvely replaying the same results may lead to a sub-optimal outcome when introspection is used. Moreover, depending on the source of non-determinism, this approach may require a large amount of input to be recorded, and this record must be stored durably since it is not possible to reconstruct it. Therefore, the performance overheads of this approach probably outweigh any advantages due to exploiting non-determinism, though it would be useful to support some form of replay for debugging purposes.

All-or-nothing The opposite extreme would be to consider any failures to be fatal to the whole job. Such an approach would only be practical in an extremely-reliable system where data loss is highly unlikely. However, it may be appropriate for systems that are implemented entirely within a single failure domain, such as an individual multi-processor computer. In such cases, the all-or-nothing approach could be implemented in a lightweight manner that gives better performance than the existing CIEL implementation.

Bounded non-determinism Some computations have deterministic outputs, but may use non-determinism internally to obtain a result more efficiently. For example, a commutative and associative aggregation function of several inputs may be computed in any order, but will yield the same result [8, 11]. Therefore, the developer could specify annotations that *bound* the non-determinism within a subgraph of the overall task graph, and remove non-deterministic taint after the boundary. CIEL would re-execute the whole subgraph if either an intermediate value or the result were lost during subsequent computations.

Checkpointing If it were impossible to reconstruct the result of a bounded non-deterministic subgraph by starting from scratch, we could checkpoint its outputs, and (if necessary) replicate its storage across multiple machines. This approach is used in DryadOpt, which divides a branch-and-bound computation into multiple Dryad jobs, each of which is internally non-deterministic, and materialises the results after each round [7]. It is also the *only* fault-tolerance mechanism in Piccolo, which has a programming model based on updates to partitioned key-value tables [19]; hence Piccolo can support non-deterministic jobs such as web crawling.

Application-specific handling The final alternative is to expose failure to the application developer, who is best able to handle it. For example, CIEL already uses *error* *references* to signify that an application-level error has occurred, and these could be extended to cover missing non-deterministic objects. This would necessitate error-checking code wherever a non-deterministic reference could arise. Alternatively, if failure is deemed unlikely, we could extend Skywriting with exception blocks.

The appropriate failure-handling technique will depend on the characteristics of the job (e.g. its expected duration and the source of non-determinism) and the cluster (e.g. the mean time between failure).

5 Conclusions

In this paper, we have argued that non-determinism deserves first-class treatment in distributed execution engines. We believe that there exist sufficiently capable developers who can use these features to develop better programs on top of distributed execution engines. Moreover, we note that systems like MapReduce and Dryad have led to many unexpected applications, and we believe that adding more expressivity to CIEL will improve developers' ability to devise creative solutions that we have not yet imagined.

We have started out with a language, Skywriting, that is deterministic by default, and these proposals do not weaken that property. Existing Skywriting programs will continue to work as expected, and—unlike in the shared-memory multithreaded case—developers are not forced to contend with non-determinism in order to achieve simple tasks. Non-determinism has been unfairly tarnished by the challenges of writing correct multithreaded programs: we hope that the arguments and simple abstractions herein will rehabilitate it, and lead to more efficient and diverse parallel programs.

References

[1] PoweredBy – Hadoop Wiki. `http://wiki.apache.org/hadoop/PoweredBy`, accessed 13th April, 2011.

[2] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of OSDI* (2010).

[3] AVIRAM, A., WENG, S.-C., HU, S., AND FORD, B. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of OSDI* (2010).

[4] BERGAN, T., HUNT, N., CEZE, L., AND GRIBBLE, S. D. Deterministic Process Groups in dOS. In *Proceedings of OSDI* (2010).

[5] BOCCHINO, JR., R. L., ADVE, V. S., ADVE, S. V., AND SNIR, M. Parallel programming must be deterministic by default. In *Proceedings of HotPar* (2009).

[6] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. In *Proceedings of SOSP* (1995).

[7] BUDIU, M., DELLING, D., AND WERNECK, R. DryadOpt: branch-and-bound on distributed data-parallel execution engines. In *Proceedings of IPDPS* (2011).

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

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP* (2007).

[10] GOLDIN, D., AND WEGNER, P. The interactive nature of computing: refuting the strong Church-Turing thesis. *Minds and Machines 18* (2008), 17–38.

[11] 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).

[12] 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).

[13] KAMBATLA, K., RAPOLU, N., JAGANNATHAN, S., AND GRAMA, A. Asynchronous algorithms in MapReduce. In *Proceedings of Cluster* (2010).

[14] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36* (1987), 471–482.

[15] MCSHERRY, F. A uniform approach to accelerated PageRank computation. In *Proceedings of WWW* (2005).

[16] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: interactive analysis of web-scale datasets. In *Proceedings of VLDB* (2010).

[17] MURRAY, D. G., AND HAND, S. Scripting the cloud with Skywriting. In *Proceedings of HotCloud* (2010).

[18] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of NSDI* (2011).

[19] POWER, R., AND LI, J. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of OSDI* (2010).

[20] QUEINNEC, C. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of ICFP* (2000).

[21] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., 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).

[22] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEY, K., SHENKER, S., AND STOICA, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys* (2010).

[23] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: cluster computing with working sets. In *Proceedings of HotCloud* (2010).

[24] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce performance in heterogeneous environments. In *Proceedings of OSDI* (2008).