Systems Qualifying Examination

Rohan Yadav

1 Introduction

Modern computer systems have evolved into parallel, distributed and heterogeneous machines, bringing fundamental challenges to the practices of efficient and productive programming. In order to achieve peak performance on these architectures, programmers must expose large amounts of parallelism, efficiently communicate between distributed memories, and offload key computations to accelerated processors. On the software engineering front, programmers desire their code to be modular, portable to new machines, and composable with other programs. Without proper programming abstractions, high performance and programmer productivity are often at odds, making programmers choose one of the two to take. To address these problems, researchers have developed a wide variety of *task-based programming models*, each with different feature sets and target domains. In this report, I provide an overview of the StarPU [2], Spark [8], and Ray [6] task-based runtime systems, as well as a methodology to comparatively evaluate the performance of task-based runtime systems through Task-Bench [7]. Next, I compare and contrast key features of these systems, and consider the implications of different design decisions on performance and programmer productivity.

2 Overview

Throughout the years, many task-based programming systems have been developed for different domains, including the high performance computing (HPC), cloud and cluster computing and the machine learning (ML) communities. As I describe later in this report, the values and problem domains focused on by each community lead to different designs and guiding principles of the different systems. Despite their different lineages, task-based programming models have a core unifying characteristic: the basic model of computation. In task-based models, computation is organized as a directed acyclic graph (DAG). Each node in the DAG represents an atomic unit of computation, and edges between nodes represent dependencies between computations. Through the representation of the computation as a DAG, task-based programming models lift computation from a stream of instructions to a higher level object that is abstracted away from a physical machine. This abstraction enables key transformations on the computation such as parallelization and scheduling of DAG nodes onto different processors across the machine.

After the DAG representation of computation, task-based programming models tend to diverge. Systems can diverge in aspects of how the DAG is created and represented, such as fully static specifications (TensorFlow [1]), algebraically specified (PARSEC [4]), or computed completely online (StarPU [2], Legion [3], and Ray [6]). Dependencies between tasks is another vector on which task-based systems differ: in some systems, dependencies are specified by the programmer (StarPU [2]), while other systems compute dependencies between tasks through different data models (Legion's [3] regions, Ray's [6] distributed objects and Spark's [8] RDDs).

Beyond the representation of computation, task-based models diverge in their support of various features, each of which are valued differently by varying communities. Such features include the raw efficiency of the runtime system (how much overhead does the system cost a user), the programmability of the system (how much *cognitive* overhead does the system cost a user) the underlying data model provided by the system, or whether the system can survive hardware faults.

I will now examine a task-based system from three different communities of interest: StarPU [2] from HPC, Spark [8] from Cloud Computing, and Ray [6] from ML. Then, I will discuss how the Task-Bench [7] framework provides an objective way to evaluate the performance of task-based systems in the face of the

varying capabilities that each system provides. Finally, I will return to the many described differences, and discuss how each affects performance and programmer productivity.

3 StarPU

The first, and oldest, system I discuss is the StarPU [2] runtime system from the HPC community. The HPC community was the first of the three communities discussed prior to start developing task-based programming systems, likely due to already feeling the pain from existing programming systems for their target machines. In particular, the HPC community had been programming distributed machines with MPI for years, while machines were evolving to contain deep memory hierarchies and nodes with accelerated processors. To program these machines effectively, programmers needed to move data between these accelerators' memories, overlap communication with computation, and schedule kernels onto these accelerators. The nature of MPI to have both explicit parallelism and explicit data movement led to programs that were difficult to write and that intertwined high-level computation with low-level data movement and synchronization. These difficulties motivated the creation of StarPU, a task-based system targeting multi-cores with heterogeneous processors attached.

StarPU allows for users to register efficient computational kernels (such as BLAS or other numerical libraries) as tasks, and provides both a task execution engine over heterogeneous architectures and a data management system that enables a coherent view of data over all memories in the system (such as GPU memory, CPU memory etc.). To execute tasks submitted by the user, StarPU contains a sophisticated scheduling system to decide the order and upon what processor to execute tasks.

StarPU provides a data model of global arrays. Users can register array data with StarPU, and describe to StarPU how this data is partitioned through different partitioning functions (such as blocked, tiled etc.). Lifting the data model into the runtime system is a key decision of StarPU that affects programmer experience: once data has been partitioned, the runtime system automatically transfers the correct data between memories, alleviating the programmer from manually communicating data.

To complement the data model, StarPU's tasks (called *codelets*) are units of computation that are abstracted away from a particular processor. Programmers can supply an implementation of a task for any architecture they wish to target, and StarPU handles execution the task on the available processor architectures. When registering a task, programmers specify the data that a task will use, and in what way the task will use the data, such as reading or writing to it. This information coupled with data registrations allows for StarPU to infer the most up-to-date version of program data and automatically move data into memories accessible by each launched task.

In this original StarPU [2] paper from 2012, users must specify dependencies between tasks when launching them, and then StarPU extracts parallelism from the user-specified dependencies. This is approach differs from systems like Legion [3], which instead determine dependencies between launched tasks at runtime through a dynamic analysis, and has pros and cons. The benefit of such an approach is that it is clear to users where dependencies between their tasks are, enabling simpler debugging and analysis when seeing unexpected program performance. However, there are downsides in both correctness and performance: without runtime analysis, users can either under-specify dependencies (leading to correctness problems), or over-specify dependencies (leading to performance problems due to less parallelism). As a result of these downsides, recent versions of StarPU have added an execution mode where the runtime performs dynamic dependence analysis to extract parallelism without user specified dependencies. Automatic dependence analysis was likely also added to aid programmer productivity when constructing large task graphs. In particular, a task graph with $O(n)$ nodes can have up to $O(n^2)$ edges, as each task may depend on all prior launched tasks. Such large task graphs are difficult to maintain by hand, leading to another challenge faced by manual dependence specification.

The prior discussion described how users of StarPU construct the DAG representing their computation. To actually execute the DAG in parallel, StarPU must schedule the tasks onto different processors in the machine, and thus the authors describe various scheduling systems built on top of a scheduling interface that users can implement (though mostly pre-written schedulers are used). These schedulers can perform priority based scheduling, or use estimates of task execution times to derive a schedule. Modern versions of StarPU even have a full system simulator to explore different ways of scheduling tasks.

The emphasis on task scheduling in StarPU comes from the guiding principle that it is the runtime system's job to automatically schedule tasks and utilize the heterogeneous hardware available on a machine. In contrast, systems like Legion [3] are of the opinion that performance impacting decisions such as scheduling order and placement are not decisions the runtime can make, and are decisions specific to particular applications. If it is the runtime system's responsibility to schedule and place tasks, then the scheduling algorithms utilized become extremely sophisticated to achieve good performance on a variety of different applications. As the results in the StarPU paper show, different applications benefit from different scheduling algorithms, leaving it up to the user to pick the best one for their application. A benefit of this approach though is that it is simpler for users — the runtime system automatically utilizes the available processors on the machine. The Legion approach yields the opposite effect — specifying an application's *mapping* is often conceptually difficult, but the ability to exert fine-grained control over task scheduling and placement makes it easier to achieve peak performance on each application.

4 Spark

The next system I consider is Spark [8], built to solve very different problems in the cloud and cluster computing community than the HPC problems targeted by StarPU. While the focus of StarPU was utilizing heterogeneous hardware to accelerate scientific computations like simulations and dense linear algebra, Spark instead focuses on the challenges of writing efficient data analytics computations on commodity clusters where data starts in a distributed file system like HDFS. The motivation of Spark came from performance challenges that arise when using the MapReduce [5] framework, which was the dominant data analytics framework in the years prior to Spark.

In particular, Spark was created to address the challenges of running interactive and iterative workloads on MapReduce. MapReduce programs operate over sets of records, and users supply two functions to the framework: a map function that takes a record and outputs a tuple of a key and value, and a reduce function that takes a set of key-value tuples with same key, and computes an aggregated value. MapReduce then in parallel applies the map function to all input records, and applies the reduce function to all records that have the same key. Tolerance to hardware failures is a key feature of MapReduce, and is enabled by MapReduce storing intermediate output from map and reduce computations into a distributed file system. While writing intermediate data to disk enables fault tolerance for MapReduce, it leads to significant performance degradation when the same data needs to be operated on multiple times, such as in iterative or interactive applications. Therefore, Spark introduces the concept of *Resilient Distributed Datasets* (RDDs) to enable both fault tolerance and flexible computation on distributed sets of records.

RDDs are the data model provided by Spark, and the computation DAG is built through operations on RDDs. RDDs are read-only distributed sequences of records that are accessible only through coarse-grained parallel operations such as *map*, *reduce*, *filter*, *join* etc. This is in stark contrast to lower-level systems like StarPU that allow for arbitrary computations that can perform fine-grained reads and writes to data. While restrictive, this model of computation has been shown to encompass a wide variety of data analytics computations, and can re-implement many specialized systems developed at the time.

The limited set of operations permitted on RDDs enable the runtime system to perform *lineage-based* recovery of RDDs in the face of faults. This means that whenever a piece of an RDD is lost due to a node failure, the system can replay all operations applied to the RDD to recover the lost component. Lineage-based recovery enables Spark to avoid dumping intermediate computations to stable storage, and can instead keep intermediate results in memory throughout the programs lifetime. Keeping RDDs in memory enables large speedups on iterative and interactive applications that repeatedly operate on the same data. As a result, Spark essentially supersedes MapReduce — it offers the same performance as MapReduce for a single-shot operations (literally providing map and reduce operations), while offering increasing speedups as the data is repeatedly operated on.

5 Ray

The final system I consider is Ray [6], a distributed system built for emerging AI applications, specifically in the context of reinforcement learning (RL). Ray started out as a Spark library for distributed RL com-

putations, but the authors moved away from Spark due to the inflexibility of the programming model and the high overheads in Spark (discussed later). The authors lay out the following set of constraints for a distributed system to effectively support RL applications:

1. Support tasks with granularity on the order of milliseconds.
2. Support varying task execution lengths, where some tasks take hours and others take seconds.
3. Support systems with heterogeneous processors (e.g. CPUs and GPUs).
4. Support both stateless (tasks) and stateful (actors) models of computation.
5. Support dynamic creation and execution of tasks.

The proposed set of constraints eliminate other existing task-based systems. Constraints 1-3 eliminate data analytics frameworks such as Spark, constraint 4 eliminates HPC systems like StarPU, and constraint 5 eliminates existing ML frameworks like TensorFlow. Therefore, the authors set out to build Ray, which satisfies all of these constraints.

The API of Ray is very simple: users add the annotation `@ray.remote` to the top of a function definition to convert it into a task, or to the top of a class definition to make it an actor. Users can invoke a function as a task by calling `f.remote(...)`. All tasks return *futures*, which can be passed to further task launches to specify dependencies between tasks. Ray provides functions `get` and `wait` to explicitly block until the result of a future is ready. Finally, users can specify the resources a task needs (such as the number of CPUs and or GPUs) through extra arguments to the `@ray.remote` annotation. Ray's API is notably simpler than HPC systems like StarPU and Legion — this has obvious benefits, but also subtle drawbacks, discussed later. To implement this API, Ray's architecture has two main components: 1) a distributed hash table, called the Global Control Store (GCS) and 2) a distributed task scheduling framework.

The GCS maintains all state in Ray, and all other components make queries into the GCS to learn information about runtime state. Information about the launched tasks, scheduling decisions about tasks, and the objects backing future handles are all stored within the GCS. Thus, when an application attempts to retrieve the return value of a task, it makes a query into the GCS. Since all runtime state is stored in the GCS, all other components of the system can be stateless. Finally, the GCS is fault tolerant, enabling the entirety of Ray to be. Since tasks, results, and dependencies are all stored in the GCS, the system can transparently recreate the results of failed tasks in a similar lineage-based approach as Spark.

The scheduling infrastructure in Ray uses a two-tiered approach, with local schedulers on each node, and a configurable set of global schedulers. When a task is submitted, it first goes to the node-local scheduler for scheduling. If the node is overloaded or cannot satisfy the tasks' resource requirements, it gets lifted to a global scheduler for scheduling. Finally, the scheduling system is locality driven, placing scheduling tasks on nodes that their arguments reside on. This scheduling system is more opinionated than StarPU: Ray handles all scheduling and encodes heuristics specific to the domain of RL, and users cannot exert more fine grained control.

Ray's architectural choices differ significantly from task-based systems in the HPC community. HPC task-based systems tend to avoid centralized state and don't exhibit separate independently scalable components like the GCS and distributed scheduler. Instead, HPC systems tend to couple all components to each rank in the computation. This is likely influenced by the difference in elasticity common in the two computing environments: AI/ML systems share the elastic compute environments as cloud systems, while HPC applications are generally run in environments where the machine size is fixed.

6 Task-Bench

The three discussed systems are very different, with different data models, target applications and APIs. These differences compound when considering the large landscape of task-based systems not discussed in this report. Despite the number of task-based systems that exist and have active users, there have been very few performance comparisons between different systems in the literature due to the difficulty of performing this comparison. Before Task-Bench, the standard approach to compare different runtime systems would

be to implement a mini-app in each system, and then perform strong and weak scaling experiments. This approach has several limitations:

1. Due to the many differences in between runtime systems, it may be difficult or impossible to implement a certain mini-app. For example, implementing an application with complex inter-node dependencies like an HPC application is likely very difficult in Spark. Additionally, such a comparison requires expertise in both compared systems, otherwise differences may be the result of programmer error.

2. The large number of runtime systems and mini-apps makes this approach suffer from a multiplicative blowup, where each runtime system would need to implement each mini-app.

3. Strong and weak scaling experiments don't directly characterize the performance of the runtime system. Strong scaling introduces factors such as increasing communication, and large initial problem sizes in weak scaling experiments can hide the overheads of a runtime system.

Task-Bench addresses these challenges by introducing a system that enables the benchmarking of task-based systems through direct construction of computation DAGs, and introduces a new metric called Minimum Effective Task Granularity (METG) to classify the performance of a runtime system under different circumstances.

Instead of full-fledged mini-apps, Task-Bench explicitly models a target computation DAG, and exposes this DAG through an API to the runtime system. Task-Bench's DAG contains information about what tasks to launch, and the dependencies between tasks. Task-Bench supports many different common DAGs, such as stencils, trees, FFTs and sweeps. Given this DAG, the runtime system must construct a DAG satisfying the same tasks and dependencies and execute it. Once a system implements this interface, all benchmark DAGs within Task-Bench run on the system, turning the multiplicative problem described earlier into an additive problem: each new runtime system just needs to implement the Task-Bench interface. This simple API enabled the Task-Bench authors to implement benchmarks for 15 different runtime systems across different domains and programming models.

Since Task-Bench can express various computation DAGs in different runtimes, the next question is how to evaluate the performance of a given runtime system on a particular DAG. Task-Bench proposes the notion of METG, which intuitively measures the smallest task granularity a runtime system can support. More concretely, the METG is parameterized by a percentage x, such that METG(x) is defined as the smallest task granularity such that the application achieves at least x% of peak performance, i.e. at least x% of the total runtime was spent performing application work. The METG is directly applicable to understanding the weak and strong scaling performance of a system: METG(x) is the smallest problem size where the runtime can weak scale with x% efficiency, and is also the problem size where the performance drop when strong scaling will hit x%. The authors show that the minimum METG achieved at scale by any system is 100 microseconds, putting a floor on the kinds of applications that can effectively utilize modern and upcoming supercomputers.

The METG is best measured under qualifications such as the number of nodes, the presence of accelerators, and the dependency pattern between tasks. The Task-Bench evaluation shows that different systems can experience large variations in METG between these environments and less stressful environments such as trivial parallelism. However, the METG only has a correspondence to application performance when the size of application tasks is comparable to the METG size — when an application has tasks significantly larger than the METG, then the cost of the runtime system has no affect on performance.

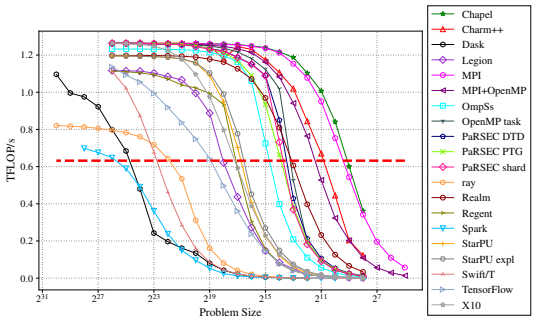
Overall, the results from Task-Bench show the following trends:

- Explicitly parallel systems like MPI have the lowest METG, followed by implicitly parallel HPC programming models, and finally data analytics and cloud frameworks.

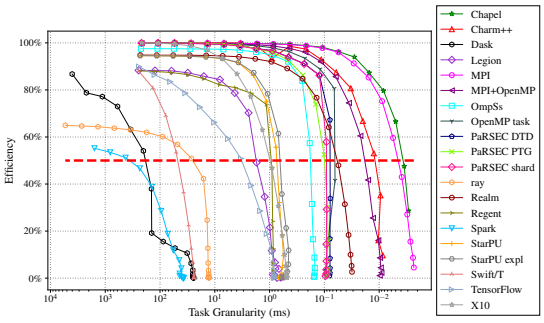
- METGs can increase by an order of magnitude between a system's single node and hundred node performance.

- Complex dependence patterns and the use of accelerators raise the METG.

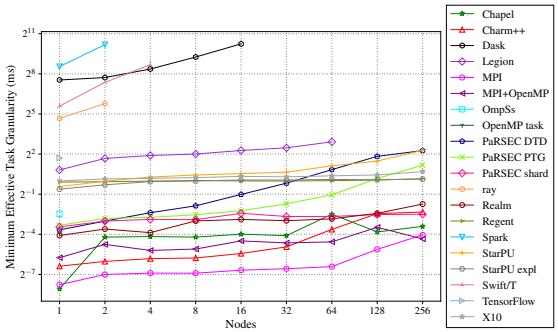
- In the face of load imbalance and when overlapping communication and computation, asynchronous task-based systems can provide benefits of bulk synchronous systems like MPI.



(a) TFLOPS/s achieved by each system as a function of problem size. Better results are higher and to the right.



(b) Percentage of peak performance achieved by each system as a function of problem size. Better results are higher and to the right.



(c) METG for each system on different node counts. Better results are lower.

Figure 1: METG experiments with a stencil dependence pattern including Ray. In (a) and (b), Ray's peak performance is lower than other systems due to the experiment being run on a weaker CPU than the other experiments.

As part of the qualifying exam, I implemented Ray within Task-Bench (`https://github.com/StanfordLegion/task-bench/pull/85`) and performed some preliminary experiments on the Sapling cluster. Overall, the process was relatively straightforward due to the simplicity of Ray's API and the future-passing based dependency construction. The results are not directly comparable with the results from the Task-Bench paper due to the different CPUs available on the Cori supercomputer, but the trends present in the results are enough to discuss. In particular, Ray has an METG higher than HPC systems like StarPU but lower than data analytics frameworks like Spark. I'm not surprised to see a sharp increase in METG when moving from 1 to 2 nodes, as the architecture of Ray seems more optimized for the case when tasks have dependencies that do not cross node boundaries.

7 Discussion

Having discussed several runtime systems from different communities, I now compare and contrast important features and design decisions between the systems, and evaluate these features' affects on the performance of the runtime system and the ease of use to end users (programmer productivity). I first consider features that are common to all runtime systems (though supported to different degrees), then delve into features commonly seen in HPC systems and then features seen in Cloud/ML systems.

7.1 Common Features

Runtime Performance and Scalability. While runtime performance may seem like a strange topic to include as a feature, it has been a core component of the design of every system investigated. Each system denote that the systems they build upon have task creation overheads that are too large, and the lower overhead tasks of the new system enable new application development. Runtime performance and scalability is a feature that is directly relevant to programmers as it sets a lower bound on the type of applications that can be developed within a system, and the less performant a system is, the more programmers have to be aware of the runtime system and work around invoking it. For example, high task creation overheads in Spark led the Ray developers to create Ray, as the performance of the runtime system locked them out of developing the applications that they wanted to. Despite performance being a key component of runtime system design, Task-Bench showed that systems can show orders of magnitude differences in runtime overhead. As shown by Task-Bench, HPC systems tend to have significantly lower overheads than Cloud/AI systems, as HPC computations often involve fine grained tasks such as stencil updates in a simulation, and compete against the zero-overhead MPI. The higher overhead systems like Ray and Spark have overheads that suit the larger granularity of tasks in their common set of applications.

Abstraction of Computation. All task-based systems I consider lift the definition of the desired computation away from the physical resources present on a machine, to different degrees. Systems like Spark don't leak anything about the target machine model to the end user, and operate entirely via transformations on RDDs. Ray and StarPU operate at a lower level than Spark by allowing for construction of a DAG through task launches, and allow for specialization of tasks to different kinds of hardware. However, the structure of the computation is never dependent on the structure of the machine. The benefits of this feature have been discussed heavily in these works: programmers gain productivity, application portability and often increased performance due to the mapping of computation onto the machine by the runtime system. This lifting does not come for free however, as even the fastest runtime systems have some overhead over MPI.

Implicit Parallelism. All systems considered in this report (and many more) have some sort of implicit parallelism, or dynamic discovery of parallelism within an application. The runtime system is responsible for the discovery and exploitation of parallelism within an application, and the programmer only needs to indicate to the system that some dependencies may exist. While supporting implicit parallelism was shown to increase METG, it offers tremendous benefits in both performance and programmer productivity. Through dynamic discovery of parallelism, the system can exploit parallelism between unrelated components within an application, and achieve speedups from parallel execution of these components. Implicit parallelism yields even larger benefits in programmer productivity. Explicit parallelism often results in tying code to hardware resources, decreasing code portability. Through implicit parallelism, the runtime system manages ordering between tasks, allowing programmers to avoid race conditions. Finally, implicit parallelism helps programmers write composable programs: because the runtime system will realize parallelism and construct dependencies, programmers can combine multiple libraries or use computations that spawn tasks as subroutines without worrying about application correctness.

Runtime Data Model. The final common feature I consider is the presence of a data model exposed by the runtime system. The most restrictive data model is Spark, which restricts all operations to be transformations on RDDs. The next is StarPU, which has a data model of global multi-dimensional arrays that the user can partition and launch tasks over. Finally, Ray's data model provides distributed objects that can be viewed from anywhere on the machine. A unifying factor of these choices is that they lift the physical data that tasks operate on up from pointers to arbitrary memory, enabling the runtime system to perform communication between memories. Supporting implicit communication is a feature that tears down a huge barrier of entry for programmers and enables programmers without significant distributed programming expertise to develop distributed applications. Implicit communication additionally allows the runtime system to ensure that the correct data is communicated, lifting the burden of correctness from programmers. Finally, explicit communication often encodes a particular data partitioning into an application, making it difficult to modify the partitioning of the data without changing the entire application. Due to the importance of communication, most implicitly parallel task-based systems have some sort of data model that enables the

runtime system to move around data to the tasks that need it. Interestingly, task-based systems for a single processor, such as Cilk, do not provide such a data model because all memory is always accessible by every task.

Finally, I want to evaluate the costs and benefits of having a higher or lower level data models like StarPU or Ray. In particular, I believe that a more expressive runtime data model can lead to improved application performance and long term programmer productivity; in contrast, a runtime data model like Ray can lead to difficulties when efficiently composing software but provides an easier initial user experience.

When the runtime system is aware of the structure of underlying data (i.e. it is more than opaque futures), it can find parallelism between tasks that operate disjoint pieces of the data structures, and efficiently communicate only updated data when tasks write and then read from different views of the same data. These models enable programmers that are well-versed in the programming model to construct partitions in ways that maximize parallelism and minimize communication with minimal code changes. The drawback is that these expressive models require up-front programmer specification of how data is intended to be partitioned, and how the computation is broken up into tasks. As a result, newer programmers have difficulty building a mental model of how to express their desired application within the framework.

At the other end of the spectrum, systems like Ray that only expose a future-based object model are simple for new programmers to grok and quickly develop in, as the semantics are nearly identical to straight-line code. However, building higher-level data structures such as distributed arrays requires re-implementing many of the same concepts used in a system like StarPU at the application level. Understanding what ranges of data are stored in each future and transmitting the necessary components or supporting different partitions of the same array is something that has to be implemented at the application level, pushing challenging distributed systems problems onto end users.

This dichotomy is somewhat reminiscent of static versus dynamic typing choice in mainstream programming languages: static types require more work from the programmer but avoid runtime checks and catch mistakes, while dynamic languages are easier to start programming with but often are the source of challenges when used to build large scale system.

7.2 HPC Features

Mapping Control. Control over mapping of tasks to specific processors is a feature that arises in HPC programming models, where systems ask the programmer to specify policies about mapping tasks and data towards processors and memories. In contrast, systems like Ray and Spark completely remove the user from such decisions, and handle mapping entirely within the runtime system. Task-Bench shows that systems that allow for fine-grained user control over mapping can achieve low overheads compared to systems that don't, so I believe that this feature comes from a tradeoff between application performance and ease of use. Many HPC applications require fine-grained allocation of tasks and data to specific processors and memories to achieve peak performance, and such decisions can be specific to individual applications. The HPC community has historically valued performance over ease of use, and thus this design decision is not surprising. Having to specify an optimal mapping is challenging even for expert programmers, and thus Cloud/AI systems let the runtime system decide such a mapping to lift this burden from users. These heuristics can lead to bad mappings for certain applications but keep the programs easy to write.

Accelerator Support. HPC computations have traditionally seen large benefits from accelerators, and distribute computations across clusters of accelerators. As a result, HPC task-based systems have been built with accelerators as a first class concept, and understanding of the memory hierarchy to complement accelerators for computation. The systems enable controlling the kind of memory that data resides in, automatically enforcing coherence between views of data in different memories, and supporting efficient communication directly between accelerator memories. In contrast, the accelerator story for the Cloud and AI systems is underdeveloped. Spark has recently added support for GPUs, but high task overheads make it difficult to utilize GPUs effectively. Ray has the ability for users to specify that tasks utilize GPUs, but is not able to effectively program with the accelerator memory hierarchy. The architecture of the global distributed object store makes it difficult to keep distributed data in accelerator memory while communicating it to accelerators on different nodes. As a result, Ray applications end up utilizing collective communication libraries like NCCL and pin objects manually in GPU memory.

7.3 Cloud/AI Features

Fault Tolerance. Fault tolerance has been a core feature of cloud-based data analytics systems since MapReduce, and fault tolerance is a key design point in Ray as well. At the same time, a few HPC systems have limited fault tolerance support, while many eschew it all together. This dichotomy arises due to the platforms that each community deploys their applications on: Cloud/AI systems are deployed on commodity clusters or spot cloud instances where failures are both extremely common and expected. In contrast, HPC systems often run on top-of-the-line machines where failures are very rare. However, this does not mean that HPC systems cannot benefit from fault tolerance, as fault tolerance erases failures from the set of situations the programmer needs to handle. With fault tolerance support, HPC systems could avoid creating checkpoints of intermediate data, or more efficiently utilize resources in the cloud. The main question in this area is whether an HPC system can support fault tolerance while keeping the METG low. Task-Bench results showed that all systems that do support fault tolerance have METGs at least an order of magnitude higher than HPC systems without fault tolerance. It is an open question if it is possible to include fault tolerance in a task-based system with as many features as an HPC system with similar overhead.

Elasticity. Cloud/AI systems support adding and removing nodes from a cluster during a computation to dynamically scale in response to load from an application. HPC systems generally do not have full-fledged elasticity support due to similar reasons as the lack of fault-tolerance: the standard HPC workflow does not consist of dynamically allocated resources. Cloud/AI systems that support elasticity achieve it by decoupling system components to enable independent scalability. For example, Spark has a coordinator node and a set of worker nodes that just execute tasks the coordinator sends them, and Ray's global control store allows any newly added nodes to view the entire state of the system. In contrast, HPC systems are generally built with node-level architectures that are not separable like Spark and Ray.

8 Conclusion

In this report, I studied and described task-based runtime systems from the HPC (StarPU), Cloud Computing (Spark) and AI/ML (Ray) communities. I then discuss an approach (Task-Bench) to quantitatively evaluate the performance of disparate runtime systems. Finally, I explored different sets of features that are common between task-based systems from different communities and discussed features supported only by systems from certain communities, along with evaluating how these features affect performance and programmer productivity. My conclusions and observations from studying these works are collected below:

- The main unifying factor of all task-based programming models is the abstraction of computation as a DAG. Beyond this unifying factor, task-based systems diverge in supported features and design decisions to achieve those features.
- Some features are supported to different extents by many task-based runtime systems include the performance of the runtime system itself, support for implicit parallelism and a data model provided by the runtime. However, the extent to which these features are supported and the flexibility of their implementations yield different results in application performance and programmer productivity.
- HPC specific features such as control over application mapping and memory hierarchy awareness enable higher end application performance at the cost of more complicated programming APIs.
- Cloud/AI specific features such as fault tolerance and elasticity increase programmer productivity around tolerating machine failures versus checkpointing methods in the HPC community, but come at a cost of runtime system performance.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng,

X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[2] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2011), 187–198.

[3] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2012), SC '12, IEEE Computer Society Press.

[4] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemariner, P., and Dongarra, J. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing 38*, 1-2 (2012-00 2012), 27–51.

[5] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (jan 2008), 107–113.

[6] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. *CoRR abs/1712.05889* (2017).

[7] Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K. S., Lee, W., Cao, Q., Bosilca, G., Mirchandaney, S., Treichler, S., McCormick, P. S., and Aiken, A. Task bench: A parameterized benchmark for evaluating parallel runtime performance. *CoRR abs/1908.05790* (2019).

[8] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (USA, 2012), NSDI'12, USENIX Association, p. 2.