Monotask in the real world [1/1]

- “Spend time doing what you're really good at and delegate out the rest”

- “In many professions, the ability to multitask has become a line item on every resume, but this needs to stop. The ability to monotask needs to be perfected in order to be truly successful. People need to re-evaluate their strengths and focus on getting one thing done well, and then move on to the next task”
Motivation [1/]

- Each job is divided into stages
- Each stage is divided into tasks
- Each task runs in a slot
**Motivation [2/]**

- **Read from network**
- **CPU processing**
- **Read/Write to disk**

- Single slot consuming different resources
- Slots in the same machine contend on different resources
Motivation [3/]

- How to reason about performance when a task bottleneck can change in a short time horizon?
  - Non deterministic
  - The more types of resource a task uses, the more vulnerable to bottlenecks

- Monotasks
  - Architecture in which the scheduling unit consumes a single resource
    - CPU, Disk, Network (memory is omitted)
    - Easier to reason about how these different factors contribute to performance
  - “Spend time doing what you're really good at and delegate out the rest”
Monotasks: Overview [1/]

- **Design principles**
  - Each monotask uses single resource
  - They execute in isolation
    - They **do not block or wait** for each other
  - Each resource has its own scheduler
    - So now contention is **visible**
  - Schedulers have **full control of a resource**
    - And they should **not be contradicted** by the OS
Monotasks: Overview [2/]

Map multitask
(one pipelined map task in Spark)

Reduce multitask
(one pipelined reduce task in Spark)
Monotasks: Overview [3/]

(a) Execution as today’s multi-resource tasks.

(b) Execution as monotasks. Arrows represent dependencies.
Monotasks: Scheduling [1/1]

Worker Node

Dag Scheduler

CPU Scheduler

Network Scheduler

Disk Scheduler

DAG Scheduler
Each multitask is organized into a DAG of monotasks

Per-Resource Schedulers
Each monotask is assigned into a specific scheduler
Monotasks: Scheduling [2/]

- Each specific scheduler has a queue
- Queues implement **Round-Robin between monotasks** in different phases
  - Maintain **high utilization by not slowing down phases**
- **CPU Scheduler**
  - One monotask per core, queue remaining
- **Disk Scheduler**
  - HDD
    - One monotask per disk, queue remaining
  - Flash
    - Allows for **concurrency** (parameter, default=4)
- **Network Scheduler**
  - Scheduling happens at the **receiver**
  - Control the **number of outstanding requests**
Monotasks: Evaluation [1/]

Figure 5: Comparison of Spark and MonoSpark for queries in the big data benchmark, using scale factor of 5, compressed sequence files, and 5 worker machines. Two configurations of Spark are shown: the default, and a configuration where Spark writes through to disk rather than leaving disk writes in the buffer cache.

Figure 6: Utilization of the most utilized (i.e., bottleneck) resource, and the second most utilized resource during stages in the big data benchmark, for both Spark and MonoSpark. Boxes show the 25, 50, and 75th percentiles; whiskers show 5th and 95th percentiles.

Figure 9: Utilization during the map stage of query 2c in the big data benchmark. With MonoSpark, per-resource schedulers keep the bottleneck resource fully utilized.
Monotasks: Evaluation [2/]

Figure 7: Comparison of Spark and MonoSpark for each stage of a machine learning workload that computes a least squares fit using 15 machines.

Figure 8: Comparison of runtime with Spark and MonoSpark for a job that reads input data and then computes on it, running on 20 workers (160 cores). Spark is faster than MonoSpark with only one or two waves of tasks, but by three waves, MonoSpark’s pipelining across tasks has overcome the performance penalty of eliminating fine grained pipelining.
Monotasks: Reasoning on Performance [I/]

- Now we know how much time a job spends on a given resource
  - We also have other metrics, like queue sizes for example
- How to use this to reason about performance under new scenarios?

Figure 10: Monotask runtimes can be used to model job completion time as the maximum runtime on each resource. This example has 4 CPU cores and 2 hard disks.
Monotasks: Reasoning on Performance [2/]

- First, calculate **Ideal Completion Time**
  - Time spent on a resource given a job

\[
I(X) = \begin{cases} 
\text{CPU} & \frac{\text{Total CPU Time}}{\text{Number of CPU}} \\
\text{NET} & \frac{\text{Total Transfer}}{\text{Throughput}} \\
\text{DISK} & \frac{\text{Total Transfer}}{\text{Throughput}} \\
\end{cases}
\]

\[
\text{Max} \left\{ \begin{array}{c}
\text{CPU} \\
\text{NET} \\
\text{DISK} \\
\end{array} \right\} = \text{Bottleneck}
\]
Monotasks: Reasoning on Performance [3/]

Second, estimate how performance will change by adding/removing resources

**Scenario 1**

1. 20 machines
2. 80 cores
3. 20 disks, 100 MB/s each
4. Job reads 20GB from disk

Job finishes in 100 minutes. In total, 85 minutes were spent in CPU and 15 minutes in IO. The ideal completion time is

1. CPU = 63.75 secs
2. IO = 20 secs

**Scenario 2**

1. 80 machines
2. 320 cores
3. 80 disks, 100 MB/s each
4. Job reads 20GB from disk

Using previous ideal time, the predicted values should be

1. CPU = 15.93 secs
2. IO = 20 secs
Monotasks: Reasoning on Performance [4/]

"for example, if a job took 10 seconds to complete on a cluster with 8 slots, it should take 5 seconds to complete on a cluster with 16 slots"

"These estimates are consistently incorrect, sometimes by a factor of two or more, because resource use is attributed equally to both jobs"
Monotasks: Reasoning on Performance [5/]

“We approximated this process in Spark by measuring the resource use on each executor while the big data benchmark is running in isolation.”

“We are able to model Spark performance only in a restricted case (when a job runs in isolation) and even in this case, the error was higher than the error for the same scenario using MonoSpark.”
Monotasks: Reasoning on Performance [6/]

Figure 18: Runtimes for three different jobs, each under different configured numbers of tasks per machine with Spark (e.g., Spark2 is Spark with 2 tasks per machine). MonoSpark automatically configures the number of tasks per machine, and performs at least as well as the best Spark configuration for all three jobs.

“MonoSpark automatically uses the ideal amount of concurrency for each resource, and as a result, performs at least as well as the best Spark configuration for all workloads”
Conclusions [1/1]

- Does Monotasks approach has to be faster than current spark?
  - Not at all, in this paper performance is just desirable
    - “I am usually a little better, and when not, I am just a little worse”

- Performance clarity
  - Well achieved?
    - It allows to reason about a certain set of resources
      - Elephant in the room: Memory
    - It seems to be very spark specific
      - or spark-ish specific

- Auto Configuration
  - Is this true for all resources?
    - Is the network configuration choice also the best possible degree of concurrency?
Selected Questions [1/1]

- Will the monotasks cause more serious job interfere when deploying into the same working machine?
- Does the monotask scheme lower the resource utilization?
- How does Monotasks maximize the utilization of heterogeneous resources/nodes?
- Can the ability for Monotasks to better determine the limiting resource be fed back into a resource allocation mechanism to improve utilization?
- Is it easy to do the decomposition for all systems? Any constraint? Maybe sometimes some job cannot be decomposed because it consumes different resources at the same time? What should we do then?
- Can Monotask perform well for latency-sensitive tasks?
Thank you! Questions?