Goals

- Maximize use of the CPU where possible
- Eliminate existing “nested parallelism” deadlocks [ARROW-7001]
- Minimize change to existing code
- Continue to maintain a single thread code path
- Avoid excess threads

Executive Summary

The current default Arrow threading configuration is problematic. The thread pool is limited whose degree of parallelism is based on the processor. However, the code is written in a synchronous (blocking) style. Therefore, I/O and blocking waits (e.g. to wait for subtasks to complete) occupy a thread that should otherwise be in use.

Either Arrow should change the default to a tunable thread pool or it should change the code style to an asynchronous (non-blocking) style. The proposal here is to change to an asynchronous (non-blocking) style in a piecemeal fashion. This proposal also serves as a general purpose review of threading capabilities in Arrow, going over some terms and describing opportunities for improvement that will hopefully be tackled as part of this proposal.

Task Schedulers

There are many tools for writing parallel code (e.g. futures, parallel-for-each, actor framework) but they all rely on some underlying task scheduler. The task scheduler is responsible for allocated tasks to the CPU hardware in some way. An application may have multiple task schedulers (e.g. one for I/O and one for CPU work) or it may schedule everything on a single task scheduler.

In arrow, task schedulers correlate to the Executor interface. Task schedulers should be interchangeable. However, some task schedulers may place additional constraints on tasks. For example, requiring that tasks do not block.

1. Thread per task

This scheduler creates a brand new thread for every task. This is the default scheduler used by AWS. There is, every so often, discussion [1][2] that thread creation has gotten fast enough that pooling is no longer necessary.

Pros:

- Simple model, allows for synchronous code, minimal learning curve for new developers
- No support code (e.g. thread pools) required whatsoever.

Cons:

- Does not achieve the goal of minimizing the # of threads.
- Hands scheduling over mostly to the OS (there are some OS-specific thread priority options but nothing in the STL)
- Thread creation is probably still too slow (would need to profile)
- Can lead to excess concurrency/locking (likely that more CPU time will have to be spent in mutex/synchronization)
- Synchronous multithreading approaches will always have duplicate code to maintain the single threaded code path

2. Large thread pool

A “large” thread pool is allowed to have more threads than there are cores/concurrency units on the system. This allows for the creation of synchronous code and the OS can be relied on to schedule threads to the appropriate physical CPUs when they are ready. This is the approach used out of the box by major applications like .NET [3], MariaDB[4], and Java[5].

These thread pools can have static/fixed size but more often they are allowed to grow as needed and will shrink back down when needed. “Nested parallelism” Deadlock is avoided by simply having enough threads that every component of a job can get a thread.

Pros:

- Simple model, allows for synchronous code, minimal learning curve for new developers
- Plenty of examples to choose from. Large industry support proves it can work.
- Can use existing thread pool code to start with

Cons:

- Does not achieve the goal of minimizing the # of threads
- Hands scheduling over mostly to the OS (there are some OS-specific thread priority options but nothing in the STL)
- Synchronous multithreading approaches will always have duplicate code to maintain the single threaded code path
- Can require tuning (max threads, min threads, how fast to create threads, etc.) based on the workload (it’s possible Arrow workloads are similar/simple enough that this isn’t the case)

3. Limited thread pool (current Arrow default)

A “limited” thread pool limits the maximum number of threads, typically to a small number based on the parallelism available in the hardware. This requires an asynchronous approach. Tasks should not block. This may seem impossible but environments such as Javascript, Unity, and WPF have managed.

Limited thread pools are more popular in CPU heavy environments. Without much I/O there is little need to worry about synchronous/asynchronous. For example, this is the typical configuration recommended when using tools like OpenMP[6] or MKL[7].

Pros:

- Supports the goal of minimizing the # of threads
- Makes sense for compute kernels as they are CPU only [Need To Verify]
- Minimizing threading overhead
- Can use existing thread pool code to start with
- Very little tuning required

Cons:

- Requires asynchronous approaches which means support for additional tooling is needed.
- Requires asynchronous tools which means there is more of a learning curve for new developers

4. Event Loop

The event loop is something used by GUI frameworks and Node[8] and there is support for them in Python[9]. In some contexts, such as non blocking file I/O, it has special meaning. However, when talking about general task scheduling, an event loop is indistinguishable from a thread pool with one worker. Event loops are therefore not relevant to this document.

5. Mixed

Other tools used a mixed approach. Node[8], for example, has an event loop which handles I/O and UI tasks, as well as a thread pool of worker threads. The Quickstep[21] project uses a scheduler thread (which is an event loop) and a pool of worker threads. The scheduler thread is allowed to run certain tasks that can help synchronize the work spread out across the worker threads.

Current Status

There are many different approaches when considering the details of parallel approaches. However, there is one significant decision that must first be made. This choice is between a synchronous “more threads than CPU parallelism” approach or an asynchronous “limited parallelism” approach.

The default in Arrow is currently a limited thread pool although there is little asynchronous code. Instead, Arrow simply takes a conservative approach and underperforms when I/O is involved (e.g. datasets do not allow multi-threaded file readers). I think the limited thread pool is the correct approach to take and effort should be made to make the code more asynchronous to increase performance.

It is not an obvious clear cut decision between the two. There is plenty of support for the “many threads” approach and it has been proven successful for a variety of tasks. Yet, I think Arrow fits the mental model of a “library” (as opposed to a framework or application) and there is a desire to keep it “small” in many ways including limiting the # of threads it uses, the RAM it uses, and well as the bandwidth or I/O that it uses. In addition, the past decade has seen a flourishing of asynchronous tooling. Furthermore, later in this document there is discussion about task scheduling. Asynchronous tooling provides a “default” API for task scheduling and puts less control in the hands of the OS. Finally, asynchronous approaches can lead to less lock contention, fewer context switches, and less concern about race conditions.

Future Work

Alternate Schedulers

[ARROW-TBD]: [LOW PRIORITY] Experiment: Measure performance using a many-threads task pool
[ARROW-TBD]: [LOW PRIORITY] Experiment: Measure performance using a thread-per-task model

Multi-Queue Thread Pool (work stealing)

The current thread pool implementation uses a single queue of tasks. By using multiple queues and some kind of work stealing strategy processor locality can be improved which may improve performance.
[ARROW-10117] *Performance: [C++] Implement work-stealing scheduler / multiple queues in ThreadPool*
[ARROW-10544] *Medium Priority - Performance: [C++] Provide per-processor Executors to support pinning workflows to a core*

Priority Thread Pool

The current thread pool has no notion of priority. In some cases ([ARROW-TBD: Need to prioritize CSV parse tasks above CSV conversion tasks]) the ability to prioritize jobs may be able to improve performance. Note: There is likely some overhead in this prioritization and it may need some benchmarking and investigation.

[ARROW-8767] Low Priority : Feature - [C++] Make ThreadPool task ordering configurable

Synchronous Concurrency Utilities

There are a variety of synchronous concurrency utilities such as concurrent queues, mutexes, spin locks, etc. Many of these are present in the STL. Further discussion of most of these are beyond the scope of this document.

Concurrent Queue

One low level synchronous primitive that is not solidly present in Arrow is a general purpose concurrent queue. There are a number of individual implementations, arrow::dataset::WriteQueue, arrow::detail::ReadaheadQueue, arrow::internal::ThreadPool::pending_tasks_ but none of these are truly general purpose. Not only could we cut down on some complexity by centralizing on a single implementation but it would also be useful for debugging/traceability (more on that later) and allow for investigating lock free queues to cut down on synchronization overhead.

There are some great open source options like Boost [12]

[ARROW-11588]: [MEDIUM-PRIORITY] Utility: Add a general purpose multi-producer / multi-consumer queue.

Asynchronous Concurrency Utilities

It can be difficult to write asynchronous code. This chiefly involves breaking long jobs into smaller tasks and then expressing the dependencies between the tasks. In addition, some state / data needs to be passed from one task to the next. There is a large variety of approaches and techniques that have been developed to tackle this problem. For the most part these coexist. Futures and Reactive streams and Actor graphs can all coexist and communicate amongst themselves, although each one adds technical debt as well as complexity & learning curve to the code base.

Callbacks

Callbacks are the bread and butter of asynchronous development. In C++ there are a variety of ways that these can be expressed. Since asynchronous programming will involve using these callbacks a lot more it may be helpful to review them briefly.

Declaring: Avoid templates + () operator unless needed

Callbacks can be declared using a template. For example:

```
template <typename T, typename V, typename MapFunction>
std::vector<V> Map(const std::vector<T>& input, MapFunction fn);
```

While this approach is the most flexible and the most performant it is also less readable and harder for the IDE to provide meaningful intellisense as it is not clear what constraints are on MapFunction [Citation needed?, this is just my hunch/experience at the moment]. Additional SFINAE constraints can be placed on the declaration to make it clearer but the end result is often still not much more readable. Finally, it leaves the lifecycle management of the callable completely in your hands (although this can sometimes be a good thing if you want to avoid copies of incoming state).

Declaring: std::function

The STL provides std::function which is a more readable way of declaring functions. It creates a type erased wrapper around your function and it will make a copy of your callable that is then owned by the function and copied when the function object is copied. In addition, it is more difficult (and sometimes impossible) for the compiler to optimize function calls away or inline. These lead to some performance overhead and care may be needed in critical sections. However, when this is not a concern, these should be preferred as it makes the code more readable and clearer.

```
template <typename T, typename V>
std::vector<V> Map(const std::vector<T>& input, std::function<V(T)> fn);
```

Implementing: Lambda & function

Callbacks can be implemented in a variety of ways. Two simple callbacks are static functions and lambdas. Both are welcome for simple callbacks that do not have much state.

Implementing: Callable objects

Lambdas with long capture lists start to get difficult to read. If your callable needs a lot of state then it can be cleaner to create a class that implements the () operator. If the state is move-only you won't be able to assign your instance to a std::function. Instead you can then create a lambda that captures a shared pointer to the callable object.

Implementing: Bound self pointer

Another approach to complex callbacks that need a class full of state is to bind a shared pointer to this. You can utilize std::enabled_shared_from_this to get a shared reference to this. By convention this pointer should be called "self". It's also possible to use std::enable_shared_from_this with std::bind to bind to a member function.

FnOnce

Arrow has defined `FnOnce` which is a lightweight version of std::function that can only be called once and will be disposed of immediately afterward. For any kind of one-shot lambda this type should be preferred over std::function to enable a faster release of resources.

[ARROW-11191] *Low Priority: Performance - Use FnOnce for TaskGroup's tasks instead of std::function*

Futures

Callbacks suffer from error propagation problems. In addition, when code starts to pass around a lot of callbacks it becomes difficult to understand the flow. Promises/Futures work around this problem by allowing tasks to be chained together in such a way that any failure will short-circuit the chain. In addition, the “when all” and “when any” methods can provide a simple API for fork/join parallelism.

The STL provides an implementation of futures & promises but the implementation does not support continuations (then/when all/when any). Partly for this reason and partly for better seamless interaction with arrow::Result and arrow::Status an internal arrow::Future class has been developed. The arrow Future class can also be (and sometimes is) compared/contrasted with a similar implementation in Apache Mesos[22].

Arrow futures currently require a heap allocation for the result and the callbacks. The futures themselves are copyable as they keep a shared pointer to their state which contains both the result and the callbacks. These futures can be chained with Then/All (Any could be added fairly easily).

There are a number of tasks that could improve the existing futures implementation in arrow although they are not currently high priority:

[ARROW-TBD] [LOW-PRIORITY] Utility: Add “when any” to futures (probably don’t bother until needed)
[ARROW-10625] [LOW-PRIORITY] Performance: [C++] Optimize Future<> creation
[ARROW-TBD] [LOW-PRIORITY] Robustness: Add abandoned logic to futures
[ARROW-8732] Low priority - Utility: [C++] Let Futures support cancellation

Coroutines

Coroutines allow for cooperative multithreading. They are often combined with some kind of event loop to achieve asynchronous development. The underlying mechanisms are different than those used by futures but the end result often ends up being quite similar. It may be more performant in some scenarios but on the other hand things like fan out to multiple CPUs may not be as clear as when done in futures.

Finally, coroutines have traditionally required some kind of language support. C++ is adding this in C++20[23] but that won’t be available for some time. Boost is able to provide coroutines[16] but there is some awkward passing around of sources and sinks. If someone wanted to tackle using coroutines for asynchronous computing it would be good to show a proof of concept taking some existing piece of Arrow and converting it to use coroutines.

Asynchronous Generators (iterators)

An asynchronous generator is a function that returns a future each time it is called. Eventually a call will result in a future that resolves to an end token. Since the generator might be consumed in parallel it could be that several futures resolve in an end token. These form a direct analogue to iterators and we have implemented some utilities in Arrow for working with these generator objects in order to achieve reuse where iterators are used today.

Asynchronous generators allow for basic stream processing in a parallel fashion. It is probably fair to say that asynchronous generators are comparable to reactive programming. One significant difference is that asynchronous generators are pull-based where reactive programming is push-based. If we adopt a reactive programming library we can replace the asynchronous generators. Some important operations that they provide are…

- Visit - applies a visitor function to each item emitted by the source generator. By default the source will be visited serially but an optional parallelism can be provided to allow for multiple visitors to run concurrently. An example of this is in use in the asynchronous CSV reader which visits each block of data to generate parse tasks.
- Collect - collects each item generated by a source generator into a vector.
- Buffer - adds a “readahead” buffer which will drain the source generator in parallel and queue up the results into a buffer. This allows the source to keep running (which is important in a file reading task for example where we don’t want the I/O thread to be idle) and can also help even out jittery task streams.
- Transform - transforms a source generator by applying a function to each item generated from it. This transformer can emit 0, 1, or many items for each source item. This can allow for a variety of operations like CSV end-of-line realignment, filtering, etc.
- Background - creates an asynchronous generator from a synchronous iterator by creating a background thread (that is outside the task scheduler) to iterate through the synchronous iterator and complete futures as it proceeds. This allows synchronous code (e.g. the current synchronous filesystem code) to be joined to asynchronous code (e.g. the asynchronous CSV reader).

Parallelism in Asynchronous Generators:

Asynchronous generators provide for pipeline parallelism. The generator functions themselves are not threadsafe and are generally not synchronously reentrant (you cannot ask for the next future while you are currently asking for a future). However, steps in the pipeline may be asynchronously reentrant (you can ask for the next future while a previously asked for future is being completed). This is controlled by readahead. Readahead reads multiple futures from a generator and queues them.

Not all pipeline steps are asynchronously reentrant. For example, a decompression task typically would not be. On the other hand a parsing task probably would be.

Asynchronous generators may also be able to handle fan-out parallelism by mapping blocks into lists of smaller tasks (kind of like a flat-map operation) at which point pipeline parallelism of the smaller tasks is equivalent to fan-out parallelism of the larger tasks. These could then be reduced by some kind of reducer node. I haven't implemented this out fully at this point.

It's also possible that the returned futures have implicit fan-out parallelism. For example, a sorting task could break a block into smaller blocks, create a tree of futures to sort and reduce into a sorted block, and then return that tree of futures with each call to the generator.

[ARROW-TBD] [LOW-PRIORITY] Performance: Add support for parallel visit
[ARROW-500] Meidum priority: Utility - [C++] Implement concurrent IO read queue for file-like sources

Reactive Programming

Reactive programming does to asynchronous sources what LINQ[19]/range[20] do to synchronous iterables. It provides an extensive set of utilities for operating on streams in an asynchronous fashion. For a good overview see the C++ reference implementation here [10]. A simpler interactive demo can be seen here [11]. Although there is a fairly steep learning curve to reactive programming it does offer a fairly extensive set of capabilities.

At the moment it is not clear that the needs of Arrow justify pulling in an external library. I am also not personally familiar with the library and do not know how well it is maintained or how performant it is. If someone were willing to do the work to investigate it I would be happy to help replace the existing asynchronous generators code.

Actor

The actor pattern is a way to build up a graph of tasks that communicate with each other. Futures/Promises/Asynchronous generators are good at fairly linear stream processing (perhaps with fan out and reduce interspersed throughout) however they can get a little more tricky when loops and non-fan-out branching is involved. The actor pattern allows for more complex task graphs to be constructed. In addition, there is implicit queuing between each of the nodes which can make it easier to create task graphs that are robust to a variety of inputs.

The C++ Actor Framework (CAF)[12] appears to provide a solid implementation although, again, I am not personally familiar with it and it has some amount of learning curve. It also requires GCC >= 7 which is a problem at the moment. If it is needed at some point in the future it should not require any complete refactor and could be brought in for targeted areas of the code where it is needed.

One of the more interesting features provided by CAF is the ability to distribute a task graph across network boundaries (i.e. on several machines). At the moment however, this is not something in Arrow's scope.

Other

There are many approaches to building up graphs of tasks to submit to a scheduler and this document is in no way an exhaustive list. For example, there are various relational algebraic approaches, the quickstep paper I referenced earlier has its own approach, and countless academic papers on the topic. This should not be too much of a concern. Asynchronous approaches are more compatible with each other than with synchronous approaches. The work of breaking up long running synchronous thread jobs into small asynchronous tasks is similar across all approaches. If a new approach is found to be ideal for some reason it is likely we have not wasted too much effort.

Converting Arrow to Asynchronous

In order to use task schedulers that are limited to a fixed number of threads it is important to convert the tasks to be asynchronous. This means the tasks cannot block. This prevents the possibility of deadlock (the tasks that can get work done are not scheduled because all threads are in use and the tasks that can't get work done are stuck on the thread pool blocking) and it improves performance (if your # of threads is equal to your # of processing units then any blocking is wasted CPU cycles, assuming there is work that can be done).

Essentially this involves breaking tasks up into smaller tasks. Where tasks would block before they instead split. The half of the task after the blocking wait becomes a continuation added on the half of the task before the split. For example,

```
PrepareRead();
auto buf = Read();
ProcessRead(buf);
```

Should instead become,

```
PrepareRead();
ReadAsync().Then([] (Buffer buf) {
  ProcessRead(buf);
});
```

This can be done piecemeal. Converting the I/O routines themselves (e.g. the filesystem interfaces) to be asynchronous is a separate subject addressed later in this document. If the I/O routines remain synchronous then they can be run on a background thread. As long as the blocking is not happening on the asynchronous thread pool then it is valid. Also, if there are no waits and there is no possibility for deadlock then it is possible to leave in existing parallelism. For example, the CSV reader used TaskGroup for dispatching conversion tasks (instead of future fan-out followed by when all). Since these conversion tasks were simple and deadlock safe they were left alone.

I have converted the CSV reader to asynchronous (while leaving the synchronous paths for the serial reader [24]) and this work can serve as a case study for how to do the conversion.

In this CSV reader task I used asynchronous generators and I plan to keep using those going forwards. That may or may not be an approach that satisfies all needs. Perhaps we will even need to use multiple approaches. This should not be too much of a concern. In general, asynchronous approaches are compatible with each other. Since the primary goal of breaking a long running thread task into smaller non-blocking tasks is achieved regardless of how those tasks are joined back together.

Although this section is somewhat brief it may represent the bulk of the work recommended by this document.

[ARROW-TBD]: Add / convert asynchronous parquet reader
[ARROW-TBD]: Add / convert asynchronous IPC (flight/feather)
[ARROW-TBD]: Convert dataset handling to asynchronous
[ARROW-10846]: Medium priority - [C++] Add async filesystem operations
[ARROW-11262]: Medium priority - [C++] Move decompression off background reader thread into thread pool
[ARROW-7001]: High priority - [C++] Develop threading APIs to accommodate nested parallelism

Extra Topics

Public asynchronous APIs

This may seem somewhat redundant but it would be a great help to offer public asynchronous APIs. In theory, this could be done even if Arrow stuck with a synchronous thread pool implementation. The main point would be to allow the caller to get off whatever thread pool they were on (e.g. python event loop, C# synchronization context, etc.) and have the operation execute on Arrow's thread pool (which may or may not be asynchronous). The operation would then return to the caller's thread pool and mark the future complete once the task is finished. I think it is important here to interface with existing language utilities. For example, if the caller is coming from python we should use Python coroutines [17] and when coming from .NET we should use Task [18].

[ARROW-1013]: Medium priority - [C++] Add asynchronous RecordBatchStreamWriter
[ARROW-1009]: Medium priority - [C++] Create asynchronous version of StreamReader

Switching to Asynchronous I/O Libraries

The majority of the Arrow code base could be made asynchronous while keeping the underlying file I/O synchronous. The file I/O could run on a background thread (or multiple background threads). All of the performance goals and deadlock avoidance goals could be achieved in this way.

However, Arrow also needs to be able to run in a single thread mode [Citation needed?, I assume we do, because we do a lot of work to support this]. Arrow currently achieves this with duplicate code paths (e.g. SerialTableReader vs ThreadedTableReader) and that will be even worse if we add asynchronous versions of the filesystem interfaces (SerialTableReader will need Readable::Read but AsyncTableReader will use Readable::ReadAsync) since we will need duplicate paths for all of the filesystem implementations.

On the other hand, if we force filesystem implementations to be asynchronous, then we could remove these duplicate paths. There would no longer be a need for SerialTableReader. If a customer wanted to restrict Arrow to a single thread they would simply use a single threaded executor. This may even have better performance than SerialTableReader because it would allow the single thread to be used for conversion while I/O is being serviced. This would only be the case for slow I/O sources (like AWS) as there is true non-blocking option for file-based I/O. For that situation, if the end user were ok with having background thread(s) for I/O then they could remove any per-call thread creation and still get the better performance.

If we chose to go down this route we would need to change the existing filesystem implementations. Libuv[13] has become a popular choice for this as it offers an asynchronous filesystem API that is also cross platform. It is used both by Julia and by Node so it is well tested and supported.

[ARROW-10846]: Convert all filesystems to be asynchronous

Filesystem Scheduling

Some filesystems support high concurrency / queue depth (e.g. AWS) while others do not (e.g. a HDD connected directly to the OS). Although even the most basic disk can support some concurrency, if nothing else, a concurrency of at least 2 will allow the OS to setup and queue reads. Some filesystems benefit from sequential reads (HDD/SDD/maybe NFS) while others do not gain any advantage (e.g. AWS).

For a concrete example, consider a request to read a 1GB CSV file from an AWS filesystem. The current Arrow implementation iterates over I/O synchronously. Assuming a block size of 1MB then ~1024 requests will be made to AWS. Each request will not start until the previous request has finished. This is obviously not ideal.

Even if the underlying filesystem is an SSD drive this is not ideal. SSD drives typically benefit both from high queue depths [15] and from sequential reads.

Some of this may be mitigated by OS scheduling. For example, if you request 1MB from the OS that may actually result in a number of queued reads to the underlying disk. On the other hand, the OS is likely not to do the correct thing when it comes to disks that prefer sequential reads.

For example, consider a dataset read that needs to read 20 100MB files from the disk. If the dataset issues 20 reads concurrently to the OS then the OS will likely (Linux will by default) service these reads in a method that prefers latency over throughput which is not what we want (given the entire operation is not complete until all files are read in).

This topic requires considerably more investigation and it is unclear how much benefit there is to gain but it is likely there is some benefit from doing our own scheduling internally. If the benefit requires us to queue up our reads (e.g. for reading from AWS) then it may result in a change to the filesystem APIs (currently one asks for a single block and does not indicate that they want to read a sequential range of blocks).

[ARROW-11583] Medium Priority : Feature - [C++] Filesystem aware disk scheduling

Add External Job Priority API

Earlier in this document we talked about adding priority into the task scheduler. This feature would allow for specifying intra-job (within job) priorities. By job I am referring to a top-level Arrow API call. So for example, request to read a single file or a dataset. If we want to allow jobs themselves to be prioritized (prefer I/O jobs over compute jobs for example) then we would need to expose the priority API. Is the job priority something that is an extra argument on all of the calls or is some other approach taken? One would also need to decide how job-level priorities interact with task-level priorities. For example, if a parsing task is high priority within a CSV job but the CSV job is low priority then what priority does the parsing task get?

[ARROW-8626] Low Priority : Feature - [C++] Implement "round robin" scheduler interface to fixed-size ThreadPool
[ARROW-8847] Finished/To-Review : Feature - [C++] Pass task size / metrics in Executor API
[ARROW-8667] Low Priority : Feature - [C++] Add multi-consumer Scheduler API to sit one layer above ThreadPool

Add Job Cancellation API

Users may wish to be able to cancel running tasks. For example, consider an SQL query console where the user issues a long running query and then decides later to cancel it and run something more targeted. Or some kind of dataset management application where the user is copying data from one location to another and decides to cancel the copy. Cancellation is not necessarily a part of threading but they share a lot of similar concerns and so it may make sense to tackle as a later part of this endeavor. In addition, threading utilities like cancellable futures may make implementing cancellation easier.

[ARROW-8732] Feature - Low priority

Debugging / Tracing Tools for Concurrency

Writing effective concurrent code can be difficult. There are a number of tools that could be developed that would improve our ability to debug. Where it would be too expensive to keep track of this information it might still be a benefit to keep track of it when running in debug mode. This list is likely incomplete, feel free to add to it.

Queue Usage Visualization

When any kind of job is doing readahead/pipelining there are queues placed between the different pieces of the job. Knowing how these queues behave over the lifetime of the job can help understand which pieces of the job are performing well and which pieces are falling behind. This could be done via sampling, or by triggering performance counters / events when the queue fills up or is exhausted. This may also require naming the queues but in most cases that shouldn't be a prohibitive requirement.

As a concrete example, consider the current asynchronous CSV reader implementation. The background I/O thread fills up a queue which is then processed by parser tasks. The parser tasks add jobs to the thread pool (which is its own sort of queue). If the queue from the I/O thread is full but we aren't utilizing all cores then it is possible that some improvement can be made. On the other hand, if the queue from the I/O never fills up then we are I/O bound and there may not be much we can do.

[ARROW-TBD]: Debugging - Add visualization into queues

Thread Pool Visualization

When investigating jobs that create a lot of tasks for performance or for bugs it can be helpful to know which tasks were executed, in which order, on which processor. Logging this information shouldn't be too difficult but visualizing it could be more complex. The timelines view in VisualVM[14] is one example of how this might be done.

[ARROW-TBD]: Debugging - Add thread pool visualization

Concurrency Performance Counters

There are a number of places we could probably add in performance counters. For example, how many tasks are being created? How many continuations? Average task time? For most of these it may be best to wait until they are needed to implement them.

References

[1] https://github.com/aws/aws-sdk-cpp/issues/5
[2]
https://stackoverflow.com/questions/14351352/does-asynclaunchasync-in-c11-make-thread-pools-obsolete-for-avoiding-expen
[3] https://docs.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool
[4] https://mariadb.com/kb/en/thread-pool-system-status-variables/#thread_pool_max_threads
[5] http://gee.cs.oswego.edu/dl/papers/fj.pdf
[6]
https://stackoverflow.com/questions/10178108/multithreading-openmp-how-many-parallel-threads#:~:text=8%20cores%20can%20only%20run,(a%20few%20maybe%20OK).

[7] http://www.diracprogram.org/doc/release-12/installation/mkl.html
[8] https://nodejs.org/en/docs/guides/dont-block-the-event-loop/
[9] https://docs.python.org/3/library/asyncio-eventloop.html
[10] https://github.com/ReactiveX/RxCpp
[11] https://rxmarbles.com/
[12] https://www.boost.org/doc/libs/1_63_0/doc/html/lockfree.html
[13] https://github.com/libuv/libuv
[14] https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/threads.html
[15]
https://serverfault.com/questions/459508/why-does-storages-performance-change-at-various-queue-depths
[16] https://theboostcpplibraries.com/boost.coroutine
[17] https://docs.python.org/3/library/asyncio-task.html
[18]
https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl
[19] https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/working-with-linq
[20] https://ericniebler.github.io/range-v3/index.html#tutorial-quick-start
[21] http://www.vldb.org/pvldb/vol11/p663-patel.pdf
[22] https://mesos.apache.org/api/latest/c++/classprocess_1_1Future.html
[23] https://en.cppreference.com/w/cpp/language/coroutines
[24] https://github.com/apache/arrow/pull/9095