Shared Memory High-Throughput Computing with Apache Arrow™

Geoffrey Lentner

Data Scientist. Research Computing. Purdue University.
glentner@purdue.edu @PurdueRCAC @GeoffreyLentner

Abstract

As the barriers to entry in scientific computing have lowered with languages like Python and their libraries, the demand for ever more sophisticated and capable frameworks that provide almost turn-key functionality has grown commensurately (see *keras* [1]).

For researchers who use the *pilot job* [2], *many-task* [3] design pattern for *high-throughput computing* [4] on modern systems, the *Apache Arrow* project [1], with its now included *Plasma* in-memory object store provides a high-level interface for sharing data structures between processes in a way that requires no serialization or copying of that data.

This poster outlines a common scenario in research computing in which a constraint is induced by the way memory is managed. The direct sharing of memory by a common middle data layer allows for simplified workflows that can operate at a more rapid cadence. A *real-world* example is provided; the goal is to raise awareness in facilitators who work with users to architect similar such high-throughput data pipelines.

The graphic in the middle of the poster showcases an anatomical diagram of a job script one might submit to a batch computing cluster which makes use of a hybrid design pattern — distributed task execution with locally shared memory.

There are multiple paradigms in research computing with differing requirements and limitations in terms of compute speed, memory, IOPS, etc. Traditionally, HPC has been defined by workloads involving large coupled tasks that require high-speed low-latency interconnects that allow an operation to be distributed across a cluster of homogeneous nodes. More common in recent years, the objective has diversified to include processing large volumes of data (either in size or in number), where the tasks are weakly coupled (if not entirely independent). In the many-task scenario, a common and effective design pattern is to submit one or more jobs to a cluster that iterate through a subset of the tasks — a pilot job.

This is not a new problem and tools exist for managing these workflows; e.g., *Launcher* [2], *GNU Parallel* [3]. There exists a subclass here; however, in which a potentially very large reference dataset is required for many or all of the individual tasks. This is problematic in a number of ways depending on the solution pursued by the user. (1) Attempt some form of *out-of-core* computing[5] where each task pulls in partial data as it's needed; or (2) the task execution is handled manually and each worker has its own copy of the reference data – holding it in memory between tasks. In the first case, a non-trivial amount of time is spent on data loading; in the second, there may not be enough memory for that level of duplication. In either case, we are spending time and resources not on the task, but the prerequisites.

Further, an exotic variant of the previously described workflow arises when these data analysis tasks come in batches where completing the tasks soon after arrival matters. In these cases, a form of job preemption or ahead-of-time scheduling would allow for a job to start before the data arrives and begin initial setup. In the case of a large reference dataset, being able to load that into memory ahead of time and access it with a comparatively no-overhead proxy reference offers a wholly new capability.

D

Real-time Forecasting and Recommendation Engine for Supernovae Events from LSST

Background

Supernovae [8] occur somewhere in the observable Universe with a somewhat regular frequency; however, even with the technological sophistication of modern observatories we only catch a few per night. This is simply a matter of knowing where and when to look. The Large Synoptic Survey Telescope (LSST) [4] has a wide-field camera and an observing strategy that will result in this number exploding to thousands of candidate supernovae per night. The challenge is that we won't be able to follow up on all of those events, and so must make data-driven decisions about which candidates to target. Further, our capacity as a community to make detailed follow-up observations differ across facilities and locations.

In order to collect the best possible data given limited knowledge, capability, and capacity, the *Time-Domain Astronomy* [6] group at Purdue University is developing a system that will take the incoming stream of alerts from multiple sources and feed them into a pipeline that builds a forecast of each supernova's light curve to establish confidence in the nature of the event and the merit in collecting additional data. This is fed into a recommendation engine that combines metrics from these forecasts with knowledge of the participating observing agents (i.e., facilities and individual astronomers) to provide suggested targets in a way that maximizes the science objectives of the community as a whole. The data collected by each observer can then be re-incorporated into the pipeline and recommendation engine — *active data collection*.

Implementation

The system is comprised of many elements, including a collection of agents that stream data through a distributed message broker that lets all other components subscribe to events, a database that houses all observational data collected from external sources in addition to a library of reference light-curves, a web-api that allows both internal and external systems to query for data and recommendations, and a daemon that automatically submits jobs (i.e., the pipeline) to one of Purdue's high performance computing clusters on a regular schedule.

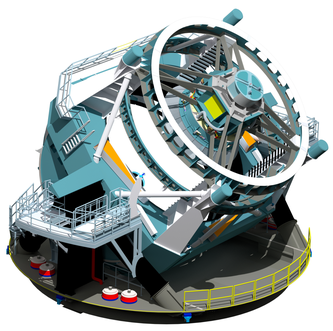
The pipeline component of the system is a batch job that gets submitted *ahead* of the anticipated stream of alerts for that evening. The job stands up a virtual cluster of engines (ipyparallel) and an in-memory object store (Plasma) on each unique host. All shared/reference data is pre-loaded into the store (e.g., a library of synthetic light curves). The driver application gets a client connection to the engines' controller and awaits on a queue, dispatching incoming alerts as triggers. Each computing engine has access to the entire body of shared data via keys to its local Plasma store.

Each engine has a client connection to the plasma store on its host and pre-acquires a proxy-reference to the shared data objects. Further, many of the events from previous nights have to be re-forecast so the observational data for those sources are pre-queried and loaded as well. In this way, conducting a single forecast takes considerably less time because all the necessary assets are already in memory.

What is LSST?

"The Large Synoptic Survey Telescope is a revolutionary facility which will produce an unprecedented wide-field astronomical survey of our universe using an 8.4-meter ground-based telescope. LSST leverages innovative technology in all subsystems: the camera (3200 megapixels, the world's largest digital camera), telescope (simultaneous casting of the primary and tertiary mirrors; two aspherical optical surfaces on one substrate), and data management (30 terabytes of data nightly, nearly instant alerts issued for objects that change in position or brightness). This innovation on all fronts has attracted some prominent donors who are innovators in technology, institutional members, and hundreds of other scientists."

https://www.lsst.org/content/lsst-general-public-faqs



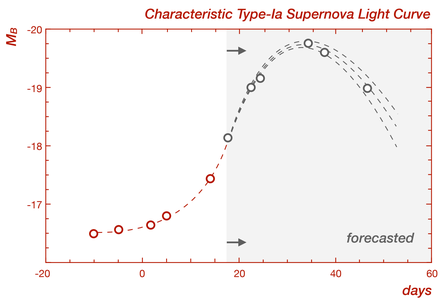


Figure D1 (above): An idealized example light curve demonstrating the general idea behind the forecasting. The gray shaded area represents the forecasted light-curve.

Figure D2 (left): Rendering of the Large Synoptic Survey Telescope (under construction).

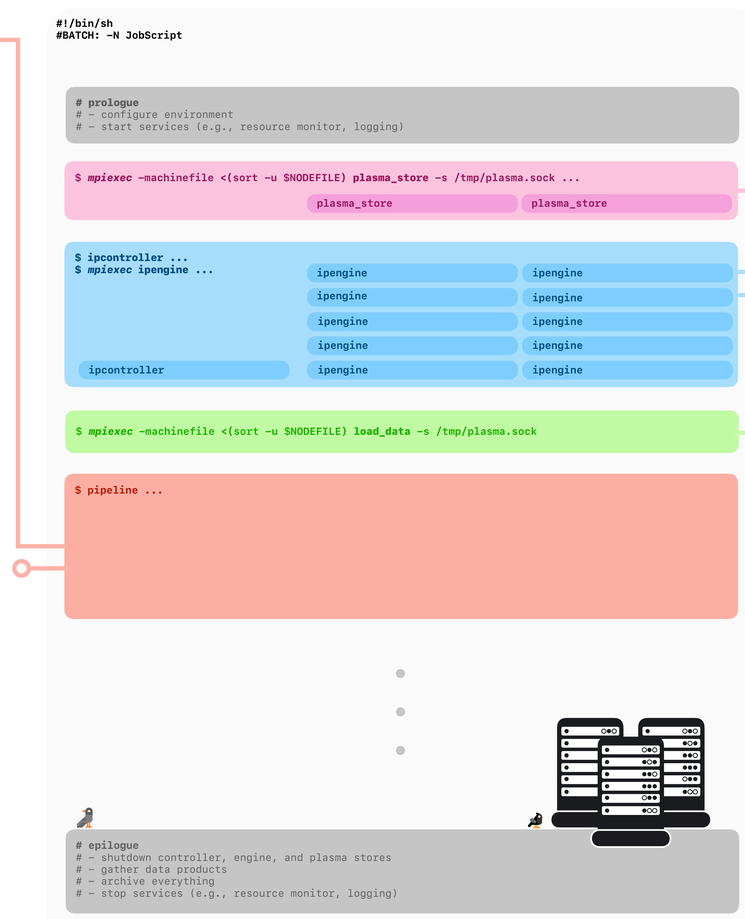


Figure C1 (above): An anatomical diagram of a batch job script including a distributed execution framework (*ipyparallel*) and localized memory sharing framework (*plasma*). In a nutshell, the engines are launched via MPI along with one Plasma store per host. Before scheduling tasks, preload the shared data assets in the same way the Plasma stores were launched. Choosing a consistent location, e.g., /tmp/plasma.sock, means that a particular engine is agnostic to which host it is running on and acquires its client connection via that same file path, regardless.

Figure C2 (right): Benchmarks for throughput in *bytes sec*$^{-1}$ for each of *out-of-core*, *plasma-store*, and *local-memory*. The task was to compute the mean value of an array of numbers (float64) equal in size to the number of bytes indicated. This task was executed for each of 2^3, 2^4, …, 2^{31} bytes for each of 1-24 concurrent workers for each of the three methods. The overlapping shaded curves show the ±1σ region over 100 trials (*local-memory*, *plasma-store*) and 10 trials (*out-of-core*), respectively, for each worker (so 100 trials for a single worker but 2400 trials for 24 concurrent workers). The results for each of 1-24 concurrent workers are plotted independently (the overlaps). In the zoom-box we can see the steady growth region for each method. The numeric values indicate the end point of the central axis of the curve for that number of workers; i.e., many workers incurs some small proportional cost. Unsurprisingly, having datasets in-memory gives ~1 order of magnitude better efficiency. The computation itself should be consistent between access methods; the order of magnitude difference in the *out-of-core* method arising almost entirely from the cost of going to disk.* The difference between *local-memory* and *plasma-store* is then entirely as a result of memory access or some other related overhead in access management.

The key takeaway is that near *local-memory* access speeds are possible without paying to penalty of duplication.

* Carried out on Purdue's Brown cluster using the *Lustre* file system.

A

Apache Arrow and the Plasma In-memory Object Store

Arrow defines a standard format for efficiently representing data in-memory (see below). *Feather* represents that format manifest on disk; *Arrow* buffers as a file. If one were to create a memory-map to such a file it would allow for larger datasets to be processed *out-of-core*. Writing and accessing a file in this way via the `/dev/shm` shared-memory file system on a Linux/BSD system lets *other* programs access the same data. *Plasma* is this concept formalized as part of the *Arrow* project. A dedicate program runs as a service (the "store"), and other programs can put/get data objects to/from the store. When a program accesses an object from the store it receives a proxy-reference that can be treated as though it were a normal buffer.

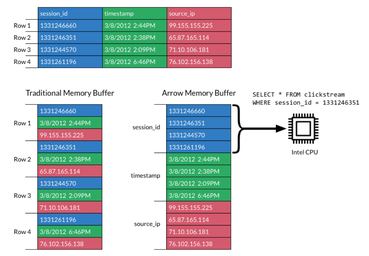


Figure A1 (above): Comparative diagram of the blocked-columnar data format defined by Apache Arrow. This format improves data locality relative to storing records in the traditional scheme.

What is Arrow?

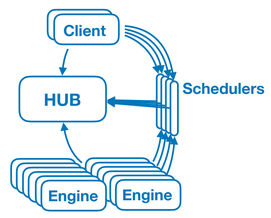
"Apache Arrow is a cross-language development platform for in-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware. It also provides computational libraries and zero-copy streaming messaging and interprocess communication. Languages currently supported include C, C++, C#, Go, Java, JavaScript, MATLAB, Python, R, Ruby, and Rust." [5]

B

Distributed Computing with IPython Parallel

In *many-task* computing it is common practice to program the task as a self contained entity (i.e., a script or function) and then use some kind of separate framework to manage the execution of these tasks. The challenge with regard to memory usage is whether the execution engine can retain data structures between tasks. In the classic scenario an individual task is contained within a shell script and all the tasks are enumerated by a file which is processed some number at a time (à la *GNU Parallel* [3]).

IPython Parallel (ipyparallel) is a framework that allows Python functions to be mapped to remote *engines* (which are extensions of the *IPython kernel* used by *Jupyter*). Data is serialized before being sent to the engines. An effective strategy for scaling a workflow involving large volumes of data and/or hundreds to thousands of engines is to map not the data itself but the metadata specifying information about how to otherwise acquire said data (e.g., file paths instead of their contents). On a high performance computing cluster this can be quite robust given a distributed, networked file system — e.g., *Lustre*.



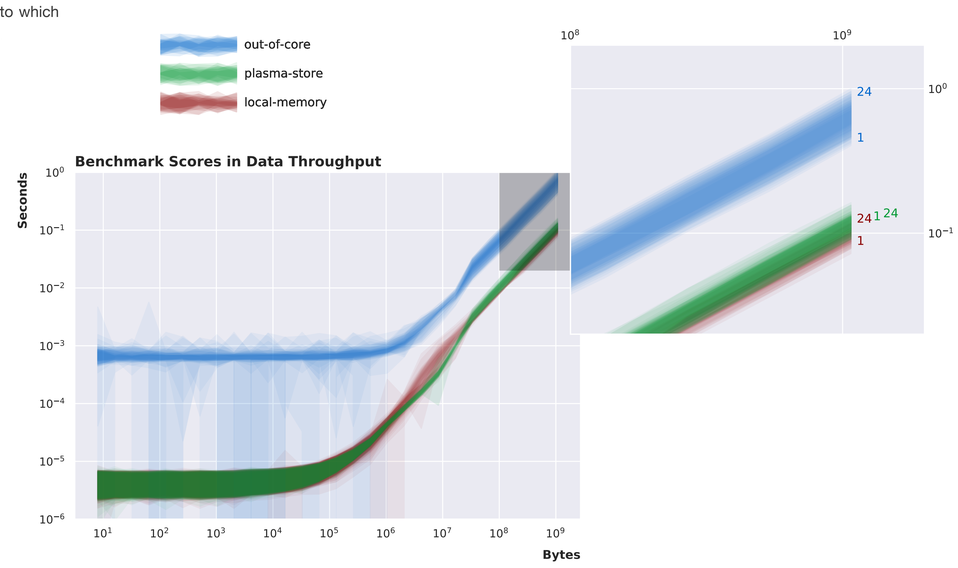
Often, it is not merely the particular data which needs loaded by the engines, but some potentially large dataset common to all the tasks, e.g., a library of reference data. It is of course possible to pre-load the data on each engine and hold a global reference; however this would result in a duplication equal to the number of engines. Even with sizable memory per node the per-core memory is typically only a few Gigabytes.

Instead, loading a single copy or such reference data into a Plasma in-memory object store before scheduling tasks means each engine now effectively has the full volume of the system's memory available to it at a near-zero overhead.

Figure B1 (above): Schematic diagram of the components that make up IPython Parallel.

C

Performance Measurements



Footnotes

[1] *Keras* is a high-level library for developing deep neural networks that abstracts away the complexities of the algorithms and linear algebra involved. It has lowered the barrier to entry for deep learning to the extent that even novice programmers can be productive almost immediately. [5]

[2] A *pilot job* is a batch job submitted to a computing cluster in which the job script is not the task itself but merely acquires the resource. The job manages the task execution manually or with a tool such as *GNU Parallel*. The idea being that the individual tasks are short enough in duration that submitting them to the scheduler would be ineffectual both in the limit in allowed total jobs as well as the latency involved in the scheduling software.

[3] *Many-task computing* (MTC) is a relatively new paradigm to be defined that bridges the gap between high-performance computing (HPC) and high-throughput computing (HTC). MTC is characterized by many small, weakly coupled (or entirely independent) tasks.

[4] *High-throughput computing* (HTC) is a paradigm in which the emphasis and constraint is not necessarily the compute speed but the volume of tasks completed over a long period of time. In some cases it is characterized by the rate at which data is being processed (as opposed to FLOPS). It can be distinguished from *high performance computing* (HPC) by the required low-latency interconnects. HTC can be distributed across disparate systems and even administrative boundaries.

[5] *Out-of-core computing* is a design pattern where data is lazy-loaded at the last possible moment before an operation and then immediately offloaded. This is necessary in situations where there is not enough memory capacity for the full dataset and it must be decomposed into blocks that can be operated on iteratively.

[6] *Time-Domain Astronomy* (TDA) is a branch of astrophysics that focuses on sources which change over time (either transient or periodic). Supernovae are only one such example; other transient phenomena include Gamma-Ray Bursts, micro-lensing, and asteroids.

[7] A *light-curve* is a time-series representation of the brightness of an astrophysical source, often used to characterize supernovae. Different types of supernovae can be distinguished by the shape of their light-curve.

[8] A *Supernova* (plural: supernovae) is the explosive "death" of certain types of stars. Supernovae can out-shine their host galaxy for the life of the event and are an active area of research in modern astrophysics.

[9] *Astronomical Surveys* stand in contrast to "traditional" observing behavior in that it is not typically an individual person or group interested in a particular astrophysical source; but rather, a dedicated (entirely or in part) facility that makes observations of the sky in a regular pattern – usually making both the raw data and finished data products available in an online archive.

References

1. Apache Software Foundation. 2019. Arrow. A cross-language development platform for in-memory data. https://arrow.apache.org
2. Lucas A. Wilson, John M. Fonner, Oscar Esteban, Jason R. Allison, Marshall Lerner, and Harry Kenya. 2017. Launcher: A simple tool for executing high throughput computing workloads. Journal of Open Source Software (Aug. 2017). https: //doi.org/10.21105/joss.00289
3. Ole Tange. 2018. GNU Parallel 2018. Ole Tange. https://doi.org/10.5281/zenodo.1146014
4. M. Jurić, et al. LSST Project. 2015. The LSST Data Management System. ArXiv e-prints (Dec. 2015). arXiv:astro-ph.IM/1512.07914 https://docushare.lsst.org/docushare/dsweb/Get/LSE- 163
5. Keras Team. 2018. Keras. The Python Deep Learning Library. https://keras.io

Acknowledgements

This work has supported by computational resources provided by Information Technology at Purdue, West Lafayette, Indiana.

Adorable birds courtesy of kurzgesagt.org. *"In a Nutshell — Kurzgesagt"*.