EPJ Web of Conferences **245**, 01022 (2020) https://doi.org/10.1051/epjconf/202024501022
CHEP 2019

JANA2 Framework for Event Based and Triggerless Data Processing

David Lawrence[1,*], *Amber* Boehnlein[1,**], and *Nathan* Brei[1,***]

[1]Thomas Jefferson National Accelerator Facility

Abstract. Development of the second generation JANA2 multi-threaded event processing framework is ongoing through an LDRD initiative grant at Jefferson Lab. The framework is designed to take full advantage of all cores on modern many-core compute nodes. JANA2 efficiently handles both traditional hardware triggered event data and streaming data in online triggerless environments. Development is being done in conjunction with the Electron Ion Collider development. Anticipated to be the next large scale Nuclear Physics facility constructed. The core framework is written in modern C++ but includes an integrated Python interface. The status of development and summary of the more interesting features are presented.

1 Introduction

JANA2 is a second generation multi-threaded event processing framework being developed at Jefferson Lab. The first generation JANA[1] framework has been successfully in use by the GlueX[2] experiment at JLab for several years. It was one of the first frameworks written with multi-threading as a primary requirement. The goal of JANA2, like all frameworks, is to provide a means of organizing algorithms and applying them to experimental data (see figure 1). Additional features to support things such as calibration constants, configuration parameters, and external resource access (e.g. field maps) are all included. JANA2 is a near complete rewrite of the framework utilizing more modern C++ features (e.g. C++11 and later). It builds upon the knowledge gained from over a decade of experience with JANA by keeping the features that worked well while implementing new features to enhance its performance in a modern HENP environment. A description of the core architecture is given in the following section followed by a section describing some new features in JANA2, a section on performance, and finally a brief section on the schedule.

2 Core Multi-threading Architecture

At its core, JANA2's job is to take data in one form and apply some algorithms to it to produce data in a more refined form. Formally, it does this with two layers of organization. The outermost layer uses a queue-arrow mechanism where data is in a queue and an "arrow"[3] is assigned to pull data from the queue, process it, and place the result in another queue. Figure 2

*e-mail: davidl@jlab.org
**e-mail: amber@jlab.org
***e-mail: nbrei@jlab.org

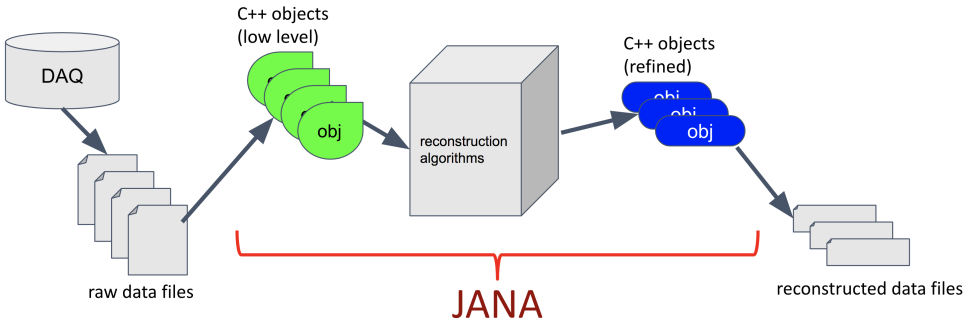

Figure 1. Diagram of where JANA fits in a traditional triggered DAQ system where data is recorded first to files and then replayed later to produce high level reconstructed objects. JANA's core purpose is to organize algorithms and efficiently apply them to low-level data objects in order to produce higher level ones.

shows a visual representation of this. The simplest and most common configuration would be to have just two queues (an input and an output) with the arrow containing a complete collection of algorithms capable of reconstructing the event. The second layer of organization is inside the arrow where JANA2 manages the algorithms along with their inputs and outputs (more on that later). The system allows for an arbitrarily complex chain of queues and arrows where algorithms can be split into different arrow types. Arrows may be either sequential or parallel. Assigning threads to arrows is how JANA2 can exploit the concurrency capabilities of the hardware both vertically and horizontally.

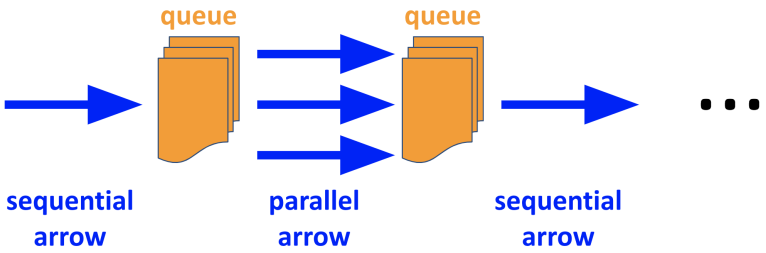

Figure 2. Visual representation of the arrow-queue design at the core of JANA2. Threads are assigned to arrows by the JANA system in a way that optimizes throughput based on the number of available threads. See text for details.

JANA2 uses the C++11 std::thread interface. This may map to different underlying thread packages on different compilers. For common Linux systems using gcc, this is *pthreads*. Though JANA should work with any threading system due to adherence to the standard, it is developed and tested on Linux and MacOS environments.

2.1 Factory Sets (i.e. algorithm collections)

A core function of JANA is to manage the algorithms used for event processing. In JANA, these are referred to as *factories* with the algorithm being a subclass of *JFactory*. The JFactory class is itself a template whose sole template parameter is the class of the type of object

that JFactory provides. Thus, the output data type is an intimate part of the JFactory class itself. This actually means that no explicit declarations need to be made to JANA about the output data type that an algorithm produces. When an algorithm is run on an event, it produces a list of zero or more objects of its output data type. These are passed as pointers to const objects so other JFactory objects that consume them as inputs cannot change their content.

The input data objects are also not explicitly declared to JANA. Requests for the desired objects are made from within the JFactory callback during event processing. This provides an important capability: *Data on Demand*. Specifically, an algorithm can request some inputs, analyze their contents and based on what it finds, decide whether to request additional inputs. This is particularly useful for high level trigger systems where a keep/toss decision could be made on inputs from algorithms that are fast and cheap while more expensive algorithms are only activated for those events that require it. Contrast this with a system that requires users to specify to the framework either through code structure or run time configuration which algorithms need to be (or may need to be) activated on demand. The on demand feature in JANA comes as a natural consequence of the core design as opposed to being a special feature that was added to provide it.

A complete set of JFactory objects needed for full event reconstruction are grouped into *JFactorySet* objects. Each event will have a JFactorySet assigned to it during its lifetime. When running with multiple threads, multiple events will be processed simultaneously and each will have its own JFactorySet. Since each JFactory object is assigned to a single event and can only be accessed by one thread at any point, JFactory objects are allowed to maintain state using data members. This is a capability not available in frameworks where a single instance of a given algorithm object is used by all threads. The typical use case for maintaining state is to store important calibration constants. Most events in a job will use the same calibration constants as the previous event processed by the JFactory. Having these in data members of the class eliminates the need to retrieve them for every event. JANA keeps track of boundaries in run number and/or event number where calibration constants may change. When it sees that a JFactory object crosses one of these boundaries, it activates a callback in the JFactory so that it can refresh any internal data members.

3 New features

Some new features have been added in JANA2 that were not present in the original JANA. This section highlights some of these.

3.1 NUMA awareness

Thread affinity and locality can be an important factor in overall performance on NUMA systems, where keeping processing threads close to the memory they are using can speed up memory access significantly. Because this depends on the combination of hardware and software, JANA2 makes affinity and locality user-configurable. There are three affinity strategies: *None*, which defers mapping threads to cores to the operating system, *ComputeBound*, which prioritizes pinning threads to hardware cores (at the expense of crossing NUMA domains), and *MemoryBound*, which prioritizes pinning within a single NUMA domain (at the expense of using hyperthreads as well as physical cores). JANA2 inspects the processor topology to determine an ordering of cpu ids that matches the given affinity strategy. When the user scales up to a desired number of threads, JANA2 uses this ordering to choose the cpu where the next thread gets pinned. Separately, the user may set a locality granularity, e.g. *Global*, *SocketLocal*, *NumaDomainLocal*, and *CoreLocal*. If the user sets *CoreLocal*, for instance, JANA2

will try to ensure that each worker thread that processes an event is pinned to the same core. Increasing locality granularity decreases memory movement, at the expense of load balancing. To mitigate the load balancing problems without incurring a scheduler overhead, the user may enable work stealing.

3.2 Python API

Python is now a very standard tool in both Physics and Data Science in general. JANA2 supports Python3 through an (optional) embedded interpreter or wrapped as a Python module. Python support is still in development, but already has experimental features in the current release. Two layers of support are available. The first is high level orchestration where configuration parameters can be set and overall process control can be exercised. For example, starting, pausing, and resuming a run. Monitoring the progress of a run, and even dynamically changing the number of threads while a job is processing are all possible in the current release (2.02). The second layer will allow deeper integration into a JANA job by implementing *JEventProcessor* classes in Python with access to the objects themselves on an event by event basis. Currently, this is being developed using pybind11[4].

3.3 Streaming Support

Streaming Readout (SRO) is fast becoming a technology of interest in HENP. The advance of computing and network technologies are enabling larger data bandwidths from front end digitization devices. This makes it possible to eliminate the need for Level-1 electronic triggering in favor of a downstream software trigger which can act on full events as opposed to information from only a subset of detectors. Figure 3 shows a sketch that illustrates how a traditional triggered system works. Here, "event"s are defined as regions of time surrounding a possible physics event. In an SRO system, the concept of "event" is exchanged for a time frame. Frames are much larger than their event counterparts (several μs vs. 100's of ns) and may contain several interesting physics interactions. A JANA2 based SRO system would implement a queue of frames with an arrow transforming the frames into something that looks more like traditional triggered events in the following queue. A second arrow could then implement additional filters or event reconstruction depending upon the needs of the specific experiment.

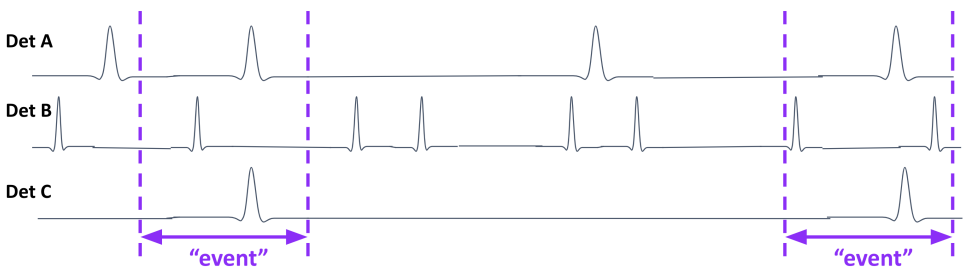

Figure 3. Traditional triggered systems record regions of time surrounding a possible interesting physics interaction as an "event". This sketch illustrates this. For streaming mode, the concept of an "event" transitions into a time frame. See text for details.

4 Performance

JANA2 has built-in testing features to measure the multi-threaded scaling performance. Results from some testing done both at NERSC and the JLab SciComp farm are shown in figure 4. Multiple processor types are shown in the plots with varying numbers of physical cores. The kinks occur at points where the number of threads matches the number of physical cores. Subscribing additional threads relies on hyperthreading which has reduced performance relative to a full core. In the case of the Knight's Landing (KNL) architecture, there are three hyperthreads available for each full core leading to multiple kinks. The scaling of course depends heavily on the parallelizability of the user supplied algorithms that are run. The design of the API for JANA2 has been carefully crafted to try and eliminate the need for user code to ever implement synchronization points (e.g. mutex locks).

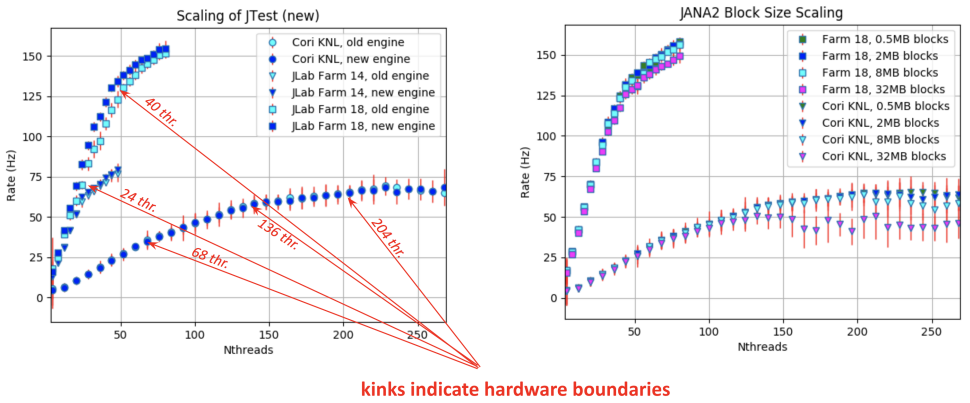

Figure 4. Plots demonstrating the scaling of JANA2 processes on multiple architectures including traditional Intel CPUs (e.g. Haswell) and KNL. The plot on the left shows event rate scaling as a function of the number of processing threads. The plot on the right shows similar scaling but for the size of the data blocks. In both cases, kinks in the distribution come at boundaries where the number of physical cores are exhausted and hyperthreading is activated. For the KNL architecture, up to 3 hyperthreads per full core are available so 4 distinct regions are observed.

5 Schedule and Outlook

JANA2 is now in a fully functional alpha release stage. The software is publicly available via GitHub at the link below. A short tutorial is available as part of the code base but can also be viewed through the GitHub website. It is currently used in the e^{JANA} package[5] that is part of the EIC development and a port of GlueX code into JANA2 is currently underway[6][7] and expected to be completed sometime in the summer of 2020. The first full production release will be available by late summer/early Fall 2020.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177. This work is supported by JLab LDRD grant 2013.

References

[1] D. Lawrence, Journal of Physics: Conference Series **119**, 042018 (2008)

[2] H. Al Ghoul, E.G. Anassontzis, F. Barbosa, A. Barnes, T.D. Beattie, D.W. Bennett, V.V. Berdnikov, T. Black, W. Boeglin, W.K. Brooks et al., AIP Conference Proceedings **1735**, 020001 (2016), https://aip.scitation.org/doi/pdf/10.1063/1.4949369

[3] Wikipedia contributors, *Arrow(Computer Science)* (2004), [Online; accessed 22-March-2019], https://en.wikipedia.org/wiki/Arrow_(computer_science)

[4] W. Jakob, J. Rhinelander, D. Moldovan, *pybind11 — seamless operability between c++11 and python* (2016), https://github.com/pybind/pybind11

[5] Electron Ion Collider, *EIC software* (2020), https://eic.gitlab.io

[6] N. Brei, D. Lawrence, *JANA2: Multi-threaded HENP Event Reconstruction* (2020), https://zenodo.org/badge/latestdoi/117695469

[7] *JANA2 online documentation* (2020), https://jeffersonlab.github.io/JANA2