Multi-threaded Event Reconstruction with JANA

David Lawrence
Thomas Jefferson National Accelerator Facility, Newport News, Virginia, USA
E-mail: davidl@jlab.org

Abstract. The C++ reconstruction framework JANA has been written to support the next generation of Nuclear Physics experiments at Jefferson Lab in anticipation of the 12GeV upgrade. The JANA framework was designed to allow multi-threaded event processing with a minimal impact on developers of reconstruction software. As we enter the multi-core (and soon many-core) era, thread-enabled code will become essential to utilizing the full processor power available without invoking the logistical overhead of managing many individual processes. Event-based reconstruction lends itself naturally to multi-threaded processing. Emphasis will be placed on the multi-threading features of the framework. Test results of the scaling of event processing rates with number of threads are presented.

1. Introduction
Computing hardware is currently undergoing a significant technology shift toward multi-core CPUs. The new generation of CPUs will employ multiple cores on a single die [1] (see figure 1). In order to fully utilize the power of this next generation of CPUs, applications will need to make a shift towards a multi-threaded programming style. Event processing in modern HENP experiments is a natural place to make use of multi-threaded technology. Reconstruction programs, however, are often large collaborative efforts with contributions to the software coming from scientists separated as much by geography as by programming experience. As such, a reconstruction framework is needed that is not only thread enabled, but allows the developers of the reconstruction code to be largely free from the nuances of multi-threaded code development. The JANA framework has been designed specifically to satisfy both of these requirements.

2. JANA Factories
In order to understand JANA’s implementation of threads, one first needs to understand a little about how the basic framework functions.

The JANA framework is built upon the idea of data factories. Figure 2 shows a flowchart that illustrates the factory mechanism. The factory model, however, differs slightly from from the traditional software factory model. The JANA factory model mimics the mechanism used by industry to fulfill requests for manufactured products. The general idea is as follows: When data is requested from a factory (i.e. an order is placed) the factory will first check to see if it has the items in stock. If so, it returns them right away. If not, then it has to manufacture them using parts obtained from other factories.

For physics analyses, the items produced by the factories are objects containing a specific type of data. The key difference from the traditional software factory model is that the JANA factories don’t actually transfer ownership of the objects. They only provide const pointers to
them. This way, if multiple consumers (other factories or end users) ask for the data produced by the factory, they will all get pointers to the same set of objects in memory. By passing const pointers, factory data is protected from manipulation by its consumers.

There are a couple of strong benefits to this approach. The first is that data objects are generated only “on demand”. A factory is only asked to generate its data objects if asked for by the user. This means CPU cycles are not wasted on generating objects unless those objects are actually needed.

Another benefit is that it allows a modular hierarchy to be built without requiring that it be written down explicitly. Factory authors only need to know what data they need for inputs in order to make their objects. Likewise, a user only needs to know what type of objects they want without any specific knowledge of the dependencies that must be fulfilled in order to make those objects. For instance, if a user asked for a list of reconstructed particles, the particle factory may ask for the charged particle track objects and the calorimeter shower objects. The factory that makes the track objects would then ask for the calibrated hit objects from the drift chambers while the factory that makes showers will ask for calibrated hits from the calorimeters. Calibrated hit factories will ask for raw data hits objects etc...

In JANA, the factories are never called directly by the user. The user only requests the pointers of the desired class of data objects from a JEventLoop object. The factories belonging to the JEventLoop object are searched until the one providing the desired data type is found.

In this model, a JEventLoop object and its complete set of factories comprise an event processing engine. It pulls events from the source as needed and invokes the callback methods of the user supplied object(s) for each event. As described in the next section, the engine can be duplicated in many threads.

3. Multi-threading in JANA

One of the design goals for JANA is to make a system that can be run with multiple threads while imposing minimal programming requirements on the developers of reconstruction code. Ideally, the developers would not have to consider threading issues at all, allowing the thread-enabled optimizations to be done only in the framework itself. By placing a complete processing engine in each thread, the factories themselves only need to be thread-safe, not thread-enabled. In other words, factory writers don’t need to create/destroy threads or lock/unlock mutexes. Figure 3 shows a schematic diagram of how JANA organizes objects in a multi-threaded application. In the figure, the JApplication, JEventSource, and JEventProcessor objects are not unique to any

![Figure 1](image.png)

**Figure 1.** Plot taken from the Intel white Paper “Platform 2015: Intel Processor and Platform Evolution for the Next Decade”[1]. This indicates Intel’s expectation that multi-core and many-core CPUs will drive the next generation of computer performance enhancement.
specific thread, but rather, are used by them all. In contrast, the JEventLoop and JFactory objects live only in the thread in which they were created. The factory objects of a given thread may then access one another and the JEventLoop object without needing to lock a mutex. JANA uses the POSIX pthreads[2] implementation of threads.

Mutex locking is required for both the input and output of the application since those resources are shared by all processing threads. The JEventSource object automatically takes care of this for the input. For the output (JEventProcessor), however, the end user must take the responsibility for mutex locking. Consider a typical example where the output is a set of histograms or trees. The JEventProcessor class must ensure that the factory-generated objects are obtained outside of a mutex lock. The calls to actually fill the histograms/trees must be made while inside of a mutex lock. The primary requirement for effective multi-threading is that the bulk of the computations are done outside of the mutex lock and in the factories. Authors of JEventProcessor classes can effectively erase any benefits from multi-threading by accessing factories while inside of a mutex lock. This is a design choice that results from absolving the factory authors from writing thread-enabled code. This choice is driven by the fact that the largest and most complex portion of the analysis code will be contained in the factories.

4. Multi-threading vs. Multiple Program Instances
One obvious alternative to using multiple threads is to simply run multiple instances of a single threaded program. The main advantages to to running the program multiple times simultaneously are:

- no programming restrictions when writing a “non-threaded” application.
- programs don’t incur the overhead of the potentially expensive use of mutexes.

Some of the disadvantages are:

1 For users of ROOT, a globally available lock is provided in the TThread library [3]. At the time of this writing, this was known not to work as advertised and one needs to implement their own mutex to properly lock the ROOT globals.
Figure 3. Schematic diagram of how objects are related to one another in a multi-threaded JANA application. A more detailed description can be found in the text.

- More bookkeeping by the user in separating the input(s) into more or less equal segments
- More bookkeeping by the user in keeping track of outputs and chaining them together after processing
- More memory usage which leads to more L2 cache misses.

The memory usage in particular will vary quite a bit depending on the specifics of the job. For jobs producing histograms, the RAM required to hold the histograms for $N$ processes will be $N$ times more than that required for running 1 process with $N$ threads. See section 4.3 below.

In general, with larger memory footprints, there are more demands for the L1 and L2 cache on the processor. This can result in more cache misses requiring more RAM accesses. This can have a significant effect on the overall processing speed as cache accesses are generally much faster than RAM accesses.

Consider also the implementation of a Level 3 farm for processing on-line data. The input data stream will need to be split and dispatched to many nodes in the farm. If each node is running $N$ processes, then it will (barring a more complicated system) have $N$ times the number of network connections. This means the dispatcher on the front end and the recombiner on the backend will each also need $N$ times the number of network connections.

4.1. Event Buffering

JANA uses an input buffer to store events for use by the event processing threads. This reduces the amount of time spent inside of the mutex lock to be only as long as it takes to copy a pointer
as opposed to keeping the lock while reading in the entire event.

JANA makes use of a condition wait mechanism when filling the event buffer. This keeps the event buffer filler thread asleep until a slot is opened due to an event processor thread pulling an event out of the buffer. The event buffer filler thread is therefore not using up CPU cycles when it should be idle.

4.2. Benchmarks for Track Finding on Simulated Data for GlueX

JANA has been tested on several multi-core/processor machines using Monte Carlo data simulated for the GlueX[4] experiment at JLab. The main CPU-intensive task being performed by the reconstruction program was charged track finding in the central and forward drift chamber systems. The data included simulated noise hits and position values smeared using approximate detector resolutions. The data set consisted of 10k events, each with 4 charged particle tracks propagating through a constant magnetic field along helical paths. The average processing time per event on a single 2.4GHz opteron is about 8.1ms.

Figure 4 shows a plot of the relative processing rates vs. the number of threads(red) or processes(blue) for tests performed on a dual core 3.0GHz Xeon and quad AND 2.4 GHz opteron 850 (SMP). All rates are normalized to the single thread rate for the given machine in order to compare how well they scale relative to one another. The total event processing rate scaled well for both machines up to the number of cores available (4 for the opteron, 2 for the Xeon). The total rate of the Xeon machine necessarily leveled off when the number of threads/processes exceeded the number of available cores.

Figure 5 shows a later test of the scaling of JANA’s processing rate on an 8 core machine. For this test, fake events were generated dynamically and multiple CPU intensive operations were done to simulate CPU usage for reconstruction. For this test, the scaling begins to drop off after 6 threads due to the processing rate approaching the rate of event generation.

4.3. Memory usage

Running many instances of a single, large program can quickly deplete the available memory on a system. In the many-core era[1] this can be ameliorated somewhat by using shared libraries rather than static linking. Dynamic allocations from the heap, however, will still be duplicated for each process. In the specific case of histograms and trees, using multiple threads has the advantage of allocating memory for each output histogram or tree only once. In the tests
Figure 5. Scaling due to multiple threads for an octal processor UltraSparc computer running Solaris 10. This used fake data generation and fake reconstruction to test the system. The blue line indicates the limit of perfect scaling.

described in section 4.2, only a few 2-D histograms were created. Each new process allocated an additional 10MB on the heap. When running with multiple threads, however, this 10MB was allocated only once and the same output histograms filled by all threads\(^2\). Having a larger, active memory footprint will result in more cache misses which can severely degrade the performance of the CPU since RAM access can be 1 to 2 orders of magnitude slower.

5. Conclusions
The JANA C++ reconstruction framework is designed for multi-threaded, event reconstruction, enabling it to take full advantage of hardware based on multi-core CPU or multiple processor SMP machines. The API is designed to easily allow scientists to develop reconstruction code with minimal understanding of how to write thread-safe code.

Running a single JANA process with multiple threads analyzes events at rates highly comparable to that of running multiple single-thread processes. Multiple threads also help lower memory requirements.

References
[1] S. Borkar H. Mulder P. Dubey S. Powlowski K. Kahn J. Rattner D. Kuck. Platform 2105: Intel processor and platform evolution for the next decade. Technical report, Intel Coorp. White Paper, 2005.

[2] Ieee std. 1003.1 (pthreads), 2004.

[3] ROOT User’s Guide 5.08, Chapter 24: Threads ppg. 355-363 (pg 357), 5.08 edition, 2005.

[4] A. R. Dzierba. Qcd confinement and the hall d project at jefferson lab.

\(^2\) This was observed using the \textit{pmap} program on a running process. Also worth noting: When the reconstruction code was placed in shared libraries, the stack allocation was reduced by only about 830kB per process. This will be much larger with a program that does full reconstruction.