PROCEEDINGS OF SCIENCE

Multi-threaded event processing with JANA

David Lawrence[∗†]
Jefferson Lab
E-mail: davidl@jlab.org

The C++ event reconstruction framework JANA was designed to allow multi-threaded event processing with a minimal impact on developers of reconstruction software. How this is done in the JANA framework is given along with some discussion on how a multi-threaded event reconstruction framework can benefit performance for both I/O bound and CPU bound jobs. Some test results supporting both of these cases are presented.

XII Advanced Computing and Analysis Techniques in Physics Research
November 3-7, 2008
Erice, Italy

[∗]Speaker.

[†]Notice: Authored by Jefferson Science Associates, LLC under U.S. DOE Contract No. DE-AC05-06OR23177. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce this manuscript for U.S. Government purposes.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence.

PoS(ACAT08)062

1. Introduction

It has been well known for some time that microprocessor development would shift from a strategy of increased clock speed to one of an increased number of cores [1]. This has prompted a renewed look at parallelizing code in order to take advantage of the CPU power available in the next generation hardware. Software multi-threading is one of the most powerful tools in the parallel toolbox, more so even than hardware threads and SIMD architectures.

The *JANA* framework [2] is being developed for the GlueX experiment [3] which plans to start data taking at Jefferson Lab in 2014. The work presented here on *JANA* focuses on the multi-threaded event processing aspect of the framework. The hardware that will be available when GlueX starts in 2014 is anticipated to have as many as 100 cores per socket [1], strongly motivating the need for parallelization in the reconstruction software.

1.1 The need for parallelism

With the hardware landscape changing to accommodate ever increasing demands for computer processing power, the software must likewise be modified to take advantage of the hardware improvements. Optimizations at the hardware and compiler levels have limitations preventing them from maximizing the full potential of parallelization leaving it to the end-of-the-line software developers to implement. In general, parallel processing decreases the time it takes to complete a job. In the case of code development, this can decrease the turn-around time in the development cycle which add up to significant reductions in manpower. It can also efficiently utilize the available resource for large reconstruction jobs more suited to computer farms.

1.2 Multi-threading vs. Multiple Processes

Parallelization in event processing is not a new concept. Earlier experiments sought to improve overall throughput by implementing event dispatching schemes using either the network for multiple computers, or shared memory and multiple processes in a single computer [4]. This early experience indicates one obvious option for parallelization on a multi-core computer: multiple processes. There are clear advantages to this approach. Consider, for example, a system where multiple processes are launched and they communicate with both a dispatcher program and a accumulator program to retrieve unprocessed events and record processed ones respectively. For this system, the only data shared by the programs is that which is explicitly placed in shared memory giving some level of protection against corruption of the memory of one process by another. By contrast, in a multi-threaded system all global variables are automatically shared between the threads, simplifying the development of the code. In both cases one needs to go to some trouble to protect globals from simultaneous access during run time.

Figure 1 illustrates the contrast of systems using multiple processes vs. one that uses multiple threads. What is shown depicts the common situation of code development on a desktop computer in which one has a single input file that they want to process, placing the results in a single output file. Two options are drawn for the case of multiple processes. The first assumes a random access file format that allows each of the N processes to read events from a different part of the file. This requires N simultaneously open file descriptors which will work fine for a few threads, but may not scale well to a 100 core system (see section 3.2). The second option using multiple processes

PoS(ACAT08)062

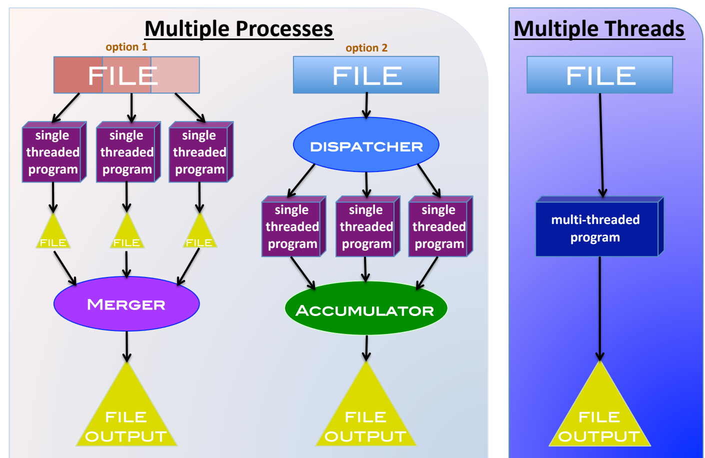

Figure 1: Simple schematic illustrating how solutions implementing multiple processes on the same computer tend to be more complex for the end user than a single process with multiple threads.

employs dispatcher and accumulator programs to coordinate the job. The point here is that both of the multiple process cases will require an additional layer in order to appear to the end user in the simple single input/single output form illustrated for the Multiple Threads case. This additional layer will need to use a scripting or forking mechanism to launch all of the different processes involved. The solution will not be as simple or elegant as a multi-threaded one and will require additional developer time[1].

2. JANA's multi-threaded implementation

JANA is built using the POSIX pthread library, pthreads [6]. It is designed to allow the exact number of event processing threads to be specified at run time through an optional command line argument. An event is always reconstructed in a single thread, eliminating the need for mutex (un)locking calls by reconstruction code authors. This is described in more detail in section 2.2.

2.1 A modified factory model

Traditional factory models in object oriented programming use a generator class to generate objects of another class whose ownership is then passed on to the caller. In *JANA*, the "factories" maintain ownership of the objects, and only return const pointers. By only publishing const pointers outside of the factory class, the data integrity is all but guaranteed.

[1]It should be noted that one can use a 3rd party package such as PROOF [5] which already hides many of the details of implementation from the end user. One is then restricted to working within the confines of the existing framework. This paper, however outlines considerations in development of a new multi-threaded system for event processing with emphasis on how the design of the multi-threaded *JANA* package addresses them.

PoS(ACAT08)062

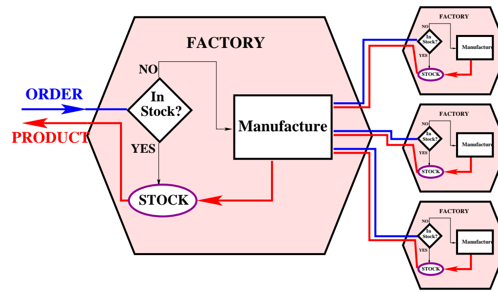

Figure 2: The modified factory model implemented by *JANA*.

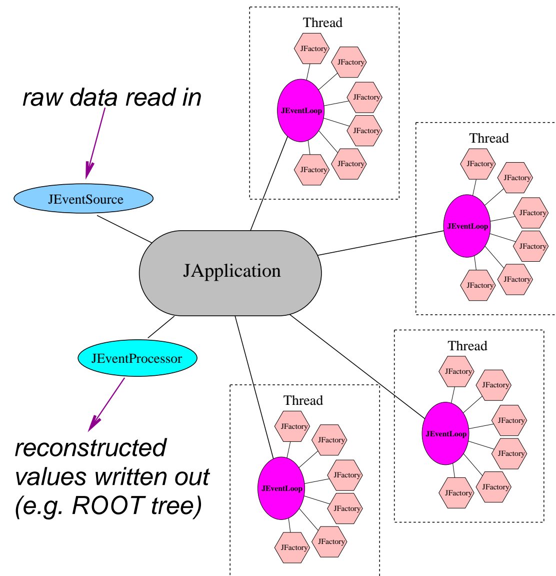

Figure 3: Diagram illustrating how *JANA* dedicates a complete set of factories to each processing thread eliminating the need for mutex locking during inter-factory communication. See text for more details.

Figure 2 shows a diagram illustrating the factory model used. The model follows what might be used in a manufacturing industry. Specifically, when an order comes into the factory, the existing stock is checked to see if the order can be immediately filled, and if not, the objects are manufactured using parts drawn from other factories as needed. Once the objects are created, they are placed in the factory's "stock" and const pointers returned. Subsequent requests to the same factory for the same event will receive a list of pointers to the same objects. This causes the usually CPU intensive manufacturing process to be invoked at most, once per event for a given factory. An additional advantage of this model is that since data is produced on demand, the factory calling sequence is handled automatically giving more flexibility in the coding.

2.2 The event processing engine in a thread

Performance in a multi-threaded application can be severely affected if mutex locks are used frequently. Minimizing mutex usage is achieved by designing the framework such that large CPU-intensive parts of the job can be done without the use of shared resources. Fortunately, the independent nature of individual events coupled with the large numbers of events that must be processed naturally lends itself to a design that requires few resources be shared between the processing

PoS(ACAT08)062

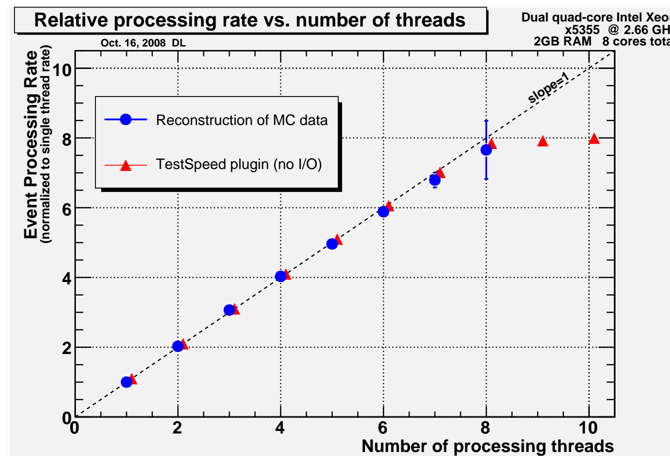

Figure 4: Event processing rate scaling with the number of threads.

threads. The *JANA* design shown in figure 3 illustrates this. In this figure, each processing thread consists of a *JEventLoop* object and a complete set of *JFactory* objects. A factory communicates with other factories in the same thread through the thread's *JEventLoop*. The key feature here is that *JFactory* objects never need to lock a mutex because the entirety of the event reconstruction is contained inside of a single thread. The only mutex locking that is required takes place in the *JEventSource* objects which read in the event and the *JEventProcessor* objects which write the events out. This is the minimum requirement since the input(output) is a single source(destination) stream, therefore requiring exclusive use by one event at a time.

3. Performance Measurements

3.1 CPU bound Jobs

Figure 4 shows the event processing rate of a multi-threaded process as a function of the number of processing threads. In the ideal case, the rate will increase linearly with the number of processing threads up until they equal the number of available cores on the system. The plot shows reconstruction of simulated data (blue) which includes charged particle tracking through a solenoidal field. Also shown are results using a special CPU-intensive testing plugin (red) which includes no I/O and extends out beyond the number of available cores into the saturation region. This plot illustrates that with this design, a task that requires similar CPU resources to reconstruction of real data can scale almost linearly with the number of processing threads (i.e. cores) up to 8 cores.

3.2 IO bound Jobs

It is estimated that by 2015, CPU's will be available with more than 100 cores [1]. These cores, being in the same computer, will still share the same disk drive and so may become I/O limited, even for CPU intensive tasks. Multi-threading can provide some performance benefits here compared to the first class of multi-process solution described in section 1.2 in which individual processes read from separate files. For example, 100 processes reading from 100 different parts of the disk (either

PoS(ACAT08)062

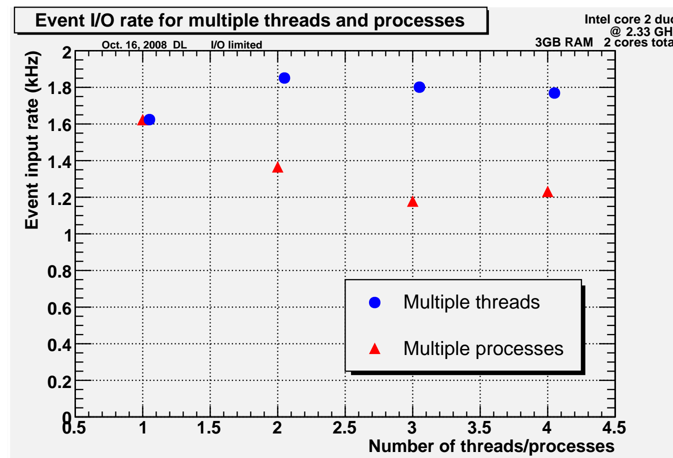

Figure 5: Event processing rates with the number of threads for IO bound jobs.

different files, or different parts of the same file) causing the read head to jump continuously. By contrast a 100 thread process will be reading events from a single file and dispatching them internally allowing the read head to follow a continuous stream with far fewer seeks. Figure 5 shows the results from a simple test that illustrates this. In the plot, the blue circles represent the total event reading rate for an I/O bound job using multiple threads. The red triangles show the same thing, except multiple instances of the same job using only a single thread were used. For the latter case, the separate processes were reading from different files. In both cases, care was taken to “clear” the disk cache by filling it with data from a source not used in the test prior to beginning the test. It is, of course, possible to design an efficient multi-process system that uses a single read process and dispatches the events to worker processes either through shared memory or socket connection. A shared memory solution will require pre-allocation of the shared memory block size while a socket connection will invoke additional system overhead for communication.

It should be noted that these benefits will likely disappear if solid state drive (SSD) technology becomes common in the future.

4. Summary

Event reconstruction in High Energy and Nuclear Physics is a CPU intensive task well suited for a multi-threaded framework. A framework has been shown that minimizes mutex locking allowing near perfect scaling in the event processing rate to be achieved for CPU intensive tasks. It has also been shown that some benefit can be realized from multi-threading in tasks where there is competition for I/O resources. This benefit is likely to become larger as the number of cores increases causing the number of competitors to increase.

Thanks to Elliott Wolin and Mark Ito of JLab for proofreading this document and giving me excellent feedback and suggestions.

PoS(ACAT08)062

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 Corp. White Paper, 2005.

[2] D Lawrence. Multi-threaded event reconstruction with jana. *Journal of Physics: Conference Series*, 119(4):042018 (6pp), 2008.

[3] A. R. Dzierba. Qcd confinement and the hall d project at jefferson lab. *hep-ex/0106010*, 2001.

[4] D. P. Weygand. The Data acquisition system for Brookhaven experiment 852. *Science at the KAON Factory Proceedings, Vancouver vol. 1* 6 p*, 1990.

[5] Maarten Ballintijn, Rene Brun, Fons Rademakers, and Gunther Roland. The proof distributed parallel analysis framework based on root. arXiv:physics/0306110v1, June 2003.

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

PoS(ACAT08)062