Study of a Fine Grained Threaded Framework Design

C D Jones
Fermilab, P.O.Box 500, Batavia, IL 60510-5011, USA

E-mail: cdj@fnal.gov

Abstract. Traditionally, HEP experiments exploit the multiple cores in a CPU by having each core process one event. However, future PC designs are expected to use CPUs which double the number of processing cores at the same rate as the cost of memory falls by a factor of two. This effectively means the amount of memory per processing core will remain constant. This is a major challenge for LHC processing frameworks since the LHC is expected to deliver more complex events (e.g. greater pileup events) in the coming years while the LHC experiment's frameworks are already memory constrained. Therefore in the not so distant future we may need to be able to efficiently use multiple cores to process one event. In this presentation we will discuss a design for an HEP processing framework which can allow very fine grained parallelization within one event as well as supporting processing multiple events simultaneously while minimizing the memory footprint of the job. The design is built around the libdispatch framework created by Apple Inc. (a port for Linux is available) whose central concept is the use of task queues. This design also accommodates the reality that not all code will be thread safe and therefore allows one to easily mark modules or sub parts of modules as being thread unsafe. In addition, the design efficiently handles the requirement that events in one run must all be processed before starting to process events from a different run. After explaining the design we will provide measurements from simulating different processing scenarios where the processing times used for the simulation are drawn from processing times measured from actual CMS event processing.

1. Introduction

HEP has been able to transition into the multi-core CPU era by exploiting the fact that batch systems just treat the additional cores as additional batch slots. However, this style of data processing may not be possible in the future due to memory limitations in future architectures. Historically, memory per unit cost has increased at the same rate as the number of transistors in a CPU[1]. Given that the doubling in transistors is being used to double the number of cores, it means the amount of memory per core we can afford is now constant at 2GB/core. Unfortunately, the complexity of LHC events is only increasing due to the larger number of interactions per crossing, i.e. pileup, being delivered by the accelerator. This complexity leads to more memory usage by our processes which are already near the 2GB resident memory limit. Finally, there are up coming technical limitations on connecting many cores to shared system memory[2] which could further reduce the memory that can be efficiently accessed by a core.

One way to deal with increased memory needs while having a constant memory per core is to use forking[3]. Forking allows the different child processes to each process their own sequence of HEP events while sharing any memory that was setup by the parent process before the forking. This allows sharing of configuration and condition information between the children. Unfortunately, we do not believe this is sufficient in the long run since the greater event complexity leads to greater memory needed for a single event. Even this year we find the shared memory between forked processes is less than the private memory used by each child process alone. Our conclusion is we need to be able to use multiple cores to processes a single event. Such an ability is naturally accommodated by using threads.

In this paper we will first discuss the CMS experiment's framework design and what parts lend themselves to parallelization. We will then move on to discussing the libdispatch[4] threading model. Next we show how the threading module can be used to implement a simplified version of a fine grained threaded HEP framework. We conclude with performance measurements made using the simplified framework.

2. Framework Design

The CMS framework[5] uses the following concepts:
Event: The Event holds all data related to a triggered readout of the CMS detector. We use the lower case 'event' when referring to the read-out beam crossing from which the data in the Event is derived.
Source: Decides which events are to be processed.
Module: A Module is the smallest unit of work handled by the framework. There are three different types of Modules: Processors, Filters and OutputModules. **Processors** read data from the Event and create derived data which is placed back into the Event. **Filters** read data from the Event and decide if that event is worth processing further. **OutputModules** read data from the Event and write it out for long term storage.
Path: A Path holds a sequence of Filters. The Path runs the Filters in order and stops processing an event if one of the Filters rejects the event or once all Filters have had a chance to process the event.
EndPath: EndPaths hold OutputModules. Each EndPath must wait until all Paths have completed since the decision to store an event is based on the result of the Paths.

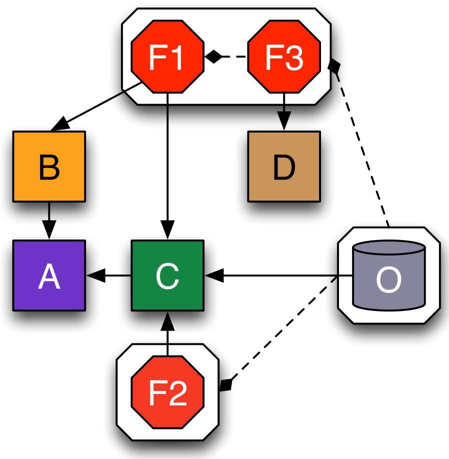

Figure 1. The figure shows an example of the dependencies between the different framework concepts. Filters are shown as red octagons and are placed in Paths represented by the white octagons. The OutputModule is represented by the grey cylinder and is placed in an EndPath shown by the white octagon. The four Producers are represented by the four squares. The solid lines show data dependencies. E.g. F1 depends on data from B and C and both B and C depend on data from A. The dashed lines show processing order dependencies. So F1 must run before F3 while both Paths must run before the EndPath.

The amount of parallelization we can gain is constrained by the dependencies between instances of the concepts. Given that events are independent of each other, we are able to run multiple events in parallel. Paths are also independent of each other and therefore lend themselves to processing the same event in parallel. Like Paths, EndPaths are independent of each other, however they cannot be run simultaneously until all Paths have finished processing a given event. However, if the same Filter appears on multiple Paths we want to guarantee that the Filter is run only once per event. Unlike Filters on a Path which must be run in sequence for a given event, the OutputModules in an EndPath can be run in parallel for a given event since they are independent. The final amount of parallelization that can be exploited is Producers can be run in parallel for a given event as long as the data they need from other Producers can be made available to them when they need it. In addition, we'd prefer not to run a Producer if its data is never needed because neither a Filter nor an OutputModule asked for its data nor did they ask for data which was derived from the Producer's data. Figure 1 shows an example of the dependencies we've just discussed.

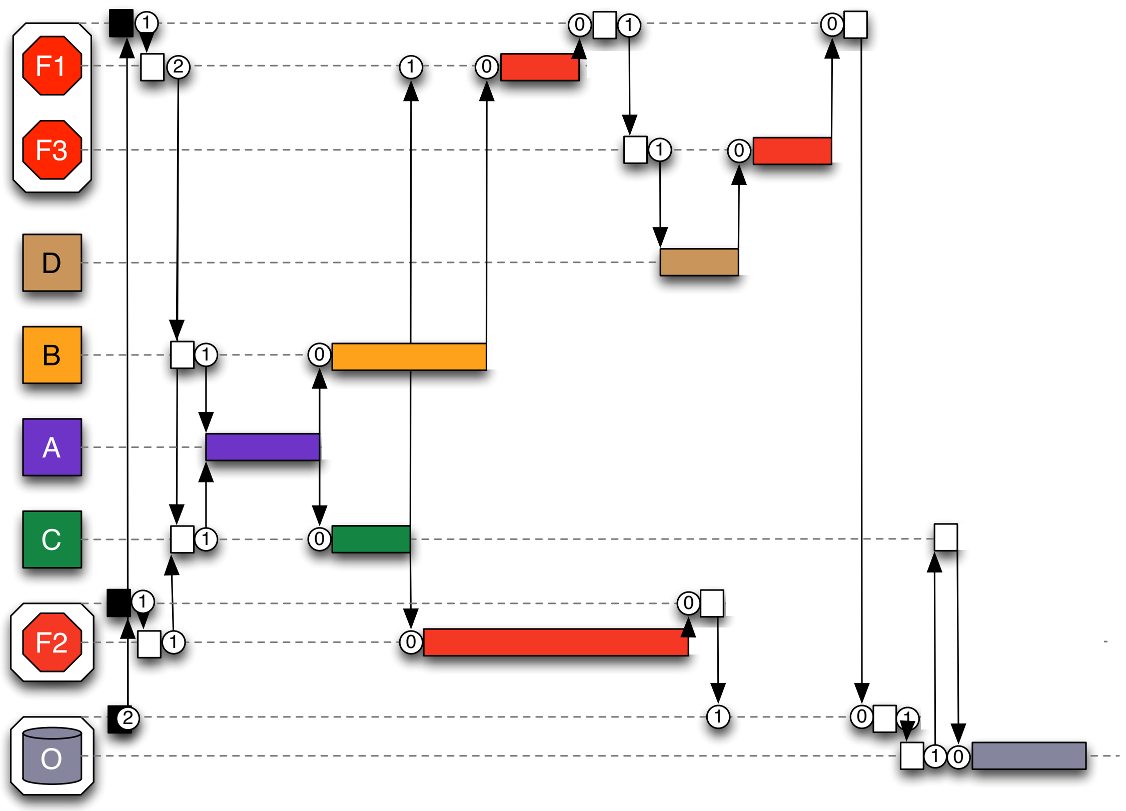

Figure 2. Example timing diagram. The vertical axis contains the different Modules which need to do work while the horizontal axis represents time. Each box represents some time used by a core to do work on behalf of a given Module. The circles with numbers represent outstanding dependencies. When the dependency goes to 0 the waiting task can do work.

Figure 2 uses the same example from Figure 1 and illustrates how the processing would progress over time. At the beginning of time, the two Paths and the EndPath, all shown as white octagons, start (represented by the black squares). The EndPath is dependent on the two Paths (shown by the circle with the number 2). The Paths each start their first respective Filters (F1 and F2) and are dependent on that Filter finishing. Filter F1 is dependent on B and C running while F2 is only dependent on C. B and C are both dependent on A so it is Producer A which is the first module to run. Once A finishes, the dependency for both B and C are satisfied so they run concurrently. C finishes early which allow F2 to start running (F1 is still waiting on B). When B finishes, F1 has all of its dependencies so F1 start to run concurrently with F2. F1 then finishes and since it accepts the event, the Path moves on to Filter F3. F3 depends on Producer D so D is started. The long running F2 finally finishes which also finishes that Path thereby reducing the EndPath's dependency to 1. With D now finished, F3 runs and once completed finishes the last Path. With both Paths now finished, the EndPath begins processing of the OutputModule O. O is dependent on C but since C is already done O can start processing immediately. Once O finishes, the whole process is started again with a new event.

3. Threading Model

For this work we chose the libdispatch framework which was developed by Apple Inc. Although libdispatch was built specifically for Apple Inc's operating systems, the code is also distributed as an open source product. The open source community has ported this code to Linux and Windows.

libdispatch is a task based queue system. A **task** is a C/C++ function plus a context, where a context is a pointer to any arbitrary user data structure. Tasks are placed into the back of thread safe queues. libdispatch pulls tasks from the front of the queues and executes them in a thread from its thread pool.

libdispatch supports two types of queues. The first type is a **global concurrent queue**. There is only one of these queues per process (on OS X and iOS there is only one concurrent queue for the entire operating system). As its name implies, the concurrent queue allows multiple tasks to be pulled from it and run simultaneously. The number of tasks run concurrently is dependent on the number of available free cores in the system. The second type of queue is a **private serial queue**. This queue is very lightweight (both in terms of memory and CPU) so the system can easily accommodate millions of queues per process. The serial queue guarantees that only one task from a given queue will be run at a time. Therefore one can use a serial queue to guarantee sequential behaviour without using system level thread synchronization primitives. Given that the queues are built using lock free constructs, the use of a serial queue can be faster than using a system lock when protecting a critical section or a concurrently accessed and updated data structure.

In addition, libdispatch offers an additional synchronization mechanism: task groups. A series of tasks can be associated with a 'dispatch_group_t' object. It is then possible to either tell the system to have the present thread wait until all tasks of that group have finished or, better yet, to associate a new task to add to a specified queue once all the grouped tasks have finished. Multiple tasks can be associated to one task group. Task groups can also be handled manually by calling 'dispatch_group_enter' and 'dispatch_group_leave' directly. Once as many 'leaves' have occurred as 'enters' the group will send its notification. The existence of task groups is critical to the way the test framework is implemented.

3. Framework Implementation

The test framework[1] processes *N* `Event` instances simultaneously, where *N* is configurable in the JSON formatted configuration file read by the program. Each `Event` instance is used to contain information from one triggered beam crossing at a time, although an `Event` instance and its associated data structures are reused repeatedly. In a real application, the number of `Events` used simultaneously would depend on the total amount of memory available on the node as well as the number of cores available. NOTE: in this paper we will distinguish between the C++ class `Event` and the abstract concept of an HEP event by capitalization and choice of font.

The base class for the `Modules` contains a list of all data products the module instance will request from the `Event`. This is used to do parallel prefetching of the data before the `Module` instance is run. The implementation also allows `Modules` to request data from the `Event` synchronously, however that ability was not used in the measurements described in this paper. This implementation has `Module` instances shared between all `Event` instances. Sharing was chosen over cloning `Module` instances for each `Event` instance since the goal was to minimize the total amount of memory used.

[1] Source code available at http://cdcvs.fnal.gov/projects/toy-mt-framework/

However, if later studies show sharing leads to poor parallelization (due to synchronization issues) it would be easy to change the implementation to also allow cloning.

The per `Event` meta-data needed for each `Module` instance is contained in a `ModuleWrapper` instance. There is one `ModuleWrapper` instance per `Module` per `Event`. The `ModuleWrapper` holds a serial queue used to guarantee a module is run only once per event. It also holds a task group which is used to notify when the data products needed by the `Module` have all been prefetched so the `Module′s` running task can then be placed in the serial queue held by the `ModuleWrapper`. Each of the `Module` types have additional per event meta-data they must store and that information is held by a class that inherits from `ModuleWrapper`.

The extra per `Event` meta-data needed by a `Producer` is held by the `ProducerWrapper`. The `ProducerWrapper` has a task group which is used to notify other `Modules` when a `Producer` has made its data product for that event. The `ProducerWrapper` also holds a Boolean which states if the `Producer` has already been run for that particular event.

The extra per `Event` meta-data needed by a `Filter` is held by the `FilterWrapper`. This meta-data is made up of two Booleans. The first Boolean states whether the `Filter` has already been run for this event while the second Boolean records if the event passed the `Filter′s` criteria. Since the same instance of a `Filter` is allowed to be on multiple `Paths` we use a `FilterOnPathWrapper` to handle the task of calling the `FilterWrapper`. Then once the `Filter` has completed the `FilterOnPathWrapper` decides if the `Path` should end or if we should advance to the next `Filter` (or more accurately the next `FilterOnPathWrapper`). To do its job, the `FilterOnPathWrapper` holds the index of the `Filter` with respect to the start of the `Path` as well as which `Path` and `FilterWrapper` it is operating on.

To simplify the implementation of the test framework, `OutputModules` were not added. Instead, the use of an `OutputModule` was modelled by a `Filter` which always passes an `Event`. Similarly, no distinct `EndPaths` were created and instead we studied configuration which had no `Paths` , i.e. only `EndPaths`. For those cases, a `Path` can be used as a substitute to an `EndPath`.

The implementation uses two recurring patterns. The first pattern is as one task ends, that task creates and queues a new task or tasks. The new task is the next action in the workflow. This is done either because the multiple tasks can be run in parallel or the new task uses a different queue and therefore has different concurrency requirements compared to the original task. The second pattern is using a task group where the task group is managed manually via ‘dispatch_group_enter’ and ‘dispatch_group_leave’. This is done since the first task associated with the task group may itself launch other tasks and it is the entire task hierarchy which must be waited upon. Manually controlling the task group is needed if multiple task groups are waiting for the same tasks.

We will use a simple example to illustrate how an event is processed in the test framework. The example will use one `Path` containing one `Filter` (F) and two `Producers` (P1 and P2). The processing of an event happens through the following steps.

- A task containing a pointer to an `Event` is added to the global concurrent queue.
- When run, the task resets the `Event` and all Wrappers associated with the `Event`. At the end the task creates a new task containing a pointer to an `Event` and adds the task to the `Source`’s serial queue.

- The task fetches the new event info from the `Source` and updates the `Event`. Since the task is from the `Source`'s serial queue, no other thread will be advancing the `Source` to the next event in the file at the same time.
- The `Source` task then creates a new task to start the `Paths`. The `Path` management task is queued to the global concurrent queue.
- When the `Path` management task is run, it resets all `Paths` and creates a new task for each `Path` and places those tasks in the global concurrent queue. The global concurrent queue is used since multiple `Paths` are allowed to run simultaneously. In our example, only one task is made since we only have one `Path`. In addition, it calls `dispatch_group_enter` on its own 'wait for paths' task group. As its final step, the `Path` manager sets a notification on its 'wait for paths' task group to add a 'do next event' task to the global concurrent queue once all `Paths` have finished processing this event.
- The `Path` task sees that we are at the beginning of the Path so it tells the `FilterOnPathWrapper` associated to `Filter` F to start processing.
- The `FilterOnPathWrapper` sees that the `FilterWrapper` hasn't yet been run for this event and data needs to be prefetched. So for each data product it calls a method of `Event` to get the data asynchronously. As part of the call it passes in its own prefetch task group as well as the identifier saying which data item is being requested.
- Once the calls to the `Event` have finished, the `FilterOnPathWrapper` sets up an asynchronous notification on its prefetch task group to add a 'do work' task to the `FilterWrapper`'s serial queue once the two prefetches have finished.
- The calls to the `Event` to get data do the following work.
    - The `Event` retrieves which `ProducerWrapper` is associated with the requested data.
    - The `Event` checks to see if the `ProducerWrapper` has already made the data. In this case it has not.
    - The `Event` calls `dispatch_group_enter` on the prefetch task group which was passed in as an argument. This will make the task group wait until the data has been obtained.
    - The `Event` then tells the `ProducerWrapper` to produce its data asynchronously. In addition, it gets the 'data done' task group from the `ProducerWrapper`.
    - The `Event` then sets up a notification on the 'data done' task group which will add a callback task to the global concurrent queue. The only thing this callback task does is call `dispatch_group_leave` on the prefetch task group, therefore informing the `FilterWrapper` that this one data item it was waiting on has now been obtained.
- The calls to the `ProducerWrapper` to produce its data asynchronously do the following work.
    - First it checks to see if the `Producer` was already run for this event. In this case it has not been.
    - Second it checks if data needs to be prefetched. In this case `Producers` P1 and P2 do not depend on any other data so no prefetch has to be done.
    - Third it calls `dispatch_group_enter` on its own 'data done' task group.
    - Finally, it sets a notification task on its prefetch task group which will add a 'do work' task to the `ProducerWrapper`'s serial queue. Since there are no prefetches, this notification goes off immediately.
- At this point in the sequence, the only tasks in any queues are the 'do work' tasks associated with the two `ProducerWrappers`. Since there are no other tasks in their respective queues, both tasks are run concurrently. These tasks

    - run their respective `Producers`;
    - take the created data and add it to the `Event`;
    - call `dispatch_group_leave` on its own ‘data done‘ task group. This triggers the notification setup in the `Event` calls which leads to the task that call `dispatch_group_leave` on the `FilterWrapper`’s prefetch task group.
- Once both `Producers` have been run, the `FilterWrapper`’s prefetch task group will send its notification. That notification will add the ‘do work’ task of the `FilterOnPathWrapper` on the `FilterWrapper`’s serial queue. Since no other task is running on that queue, the ‘do work’ task is run immediately.
- The ‘do work’ task has the `FilterOnPathWrapper` ask the `FilterWrapper` to run the `Filter`.
- The `FilterWrapper` checks that the `Filter` hasn’t already been run on this event (in this case it has not) and then tells the `Filter` to run.
- Once the `Filter` is done, the `FilterWrapper` caches the decision made by the `Filter`.
- The `FilterOnPathWrapper` takes the `Filter`’s decision and passes it, along with the `Filter`’s index on the `Path` to the `Path`’s method which determines what next to do.
- The `Path`’s ‘what next to do’ method sees that the index it was given is for the last `Filter` on the `Path` and therefore the `Path`’s work is done. The `Path` then calls back to `Path` management that it has completed its work. This callback calls `dispatch_group_leave` on the ‘wait for paths’ task group. Since all `Paths` are finished, this adds a ‘do next event’ task to the global concurrent queue which causes the entire processes to happen over for the next available event.

The use of serial queues to manage when the work of a `Module` is to be done provides a convenient mechanism for controlling thread safety. The test framework supports three types of thread safety for a `Module`: fully re-entrant, one-event-at-a-time and thread-unsafe. In the fully re-entrant case, the implementation of the `Module` is such that the same `Module` can be run on multiple events simultaneously. In that case we assign a unique serial queue to each `ModuleWrapper` instance. For the one-event-at-a-time case, a `Module` instance can be run at the same time as different `Module` instance but does not support being called simultaneously itself. In this case, all `ModuleWrappers` which work with that `Module` instance share the same serial queue. Finally, a thread-unsafe `Module` cannot be run at the same time as any other thread-unsafe `Module`. In this case, `ModuleWrapper`’s associate with a thread-unsafe `Module` share the same serial queue.

3. Measurements

We want to know the performance benefits possible from the use of a fine-grained threaded framework as well as any scaling limits inherit in the use of libdispatch. To achieve those goals we decided to configure the test framework in such a way as to mimic the timing characteristics of CMS’ reconstruction program running on ‘high-pileup’ data.

The characteristics of the reconstruction were obtained by extracting the inter-module dependencies as well as measuring the per module per event timing of the program. The inter-module dependencies, that is for each module finding which other modules make the data it needs, are stored automatically by the full CMS processing framework and are easily obtained from the resultant output file. The per module per event timing was recorded for 100 events of a reconstruction job using CMS’ most recent software version (CMSSW_5_2_0) and data recorded during the ‘high pileup’ LHC runs recorded at the end of 2011. This ‘high pileup’ data averages around 30 interactions per recorded event so is a reasonable approximation for the data to be taken during the 2012 LHC run. The reconstruction job consists of 489 modules and processing for each event is triggered by two OutputModules who are each on their own EndPath. No event filtering is applied.

We developed two special module classes for the measurements. Both modules take a list of modules to which they were dependent upon and a list of the amount of time they should wait for each of the 100 events seen in the data. The two module classes differ on how they wait. The first module waits by doing a numerical integration where the number of integration steps per second has been calibrated to the machine being used for the study. This way of waiting utilizes an entire core and approximates the CPU utilization of the actual reconstruction program. The second module type waits by calling usleep. While sleeping, a module does not use a core and therefore additional work can be scheduled. This allows one to explore how the program scales for a system which would have a very large number of cores. Measurements made using configurations for which use of the two module types should show the same performance have shown that the two modules agree to within 1%.

Besides the multi-threaded test framework, we also developed an optimized single-threaded framework. The single-threaded framework supports the same concepts as the multi-threaded framework and reads the same configuration files. In this way we can compare the relative timing between the single-threaded and multi-threaded framework to determine any inherent overhead caused by using libdispatch.

The system used to do the measurements was a virtual machine running on an Intel(R) Xeon(R) CPU E5620. The machine has 16 physical cores (4 cores/socket with 4 sockets) with each core running at 2.40GHz. The virtual machine runs Scientific Linux 6 and was configured to use all 16 cores but only allowed access to 16GB of the system's 47 GB of RAM.

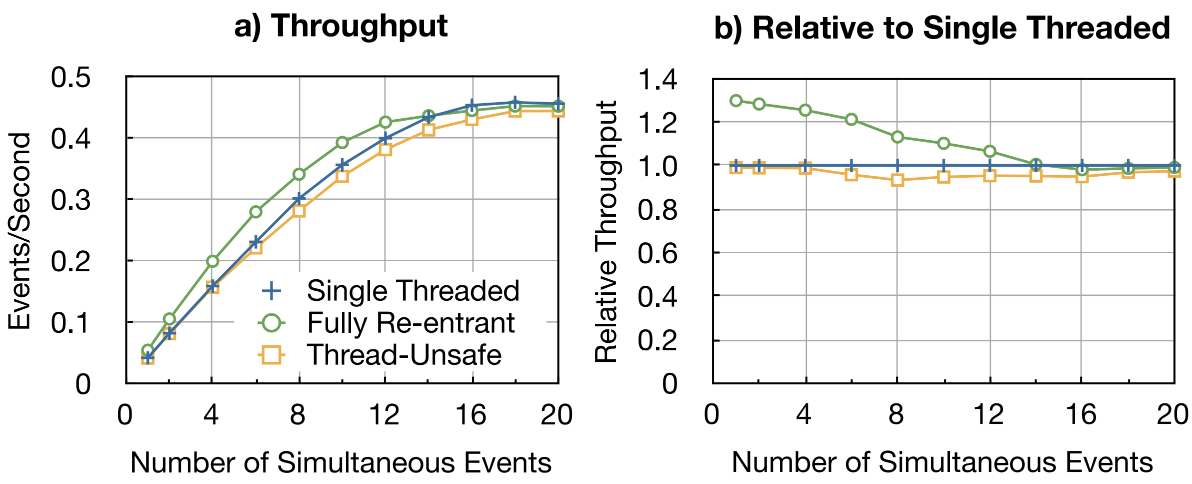

Figure 3. Throughput a) and throughput relative to single thread throughput b) measurements for 16 core machine with Producers doing numeric integration.

Figure 3 shows the throughput when using the Producers which do numeric integration while waiting their appropriate amount of time. Figure 3a shows the raw throughput and figure 3b shows the throughput relative to the single-threaded throughput. The three curves show three different configurations. The blue cross curve was for the single-threaded framework. The yellow-orange boxes are for the case where all modules say they are thread-unsafe and therefore the framework can only run one module at a time. The final green circle curve is for the case where the modules say they are fully re-entrant. In these measurements, we systematically increased either the total number of processes run simultaneously (for the single-threaded and thread-unsafe cases) or the total number of events run in parallel in one process (for the fully re-entrant case). As one should expect, all of these cases level out their throughput once we are processing as many events simultaneously as there are cores on the machine. In addition we found that at worse the thread-unsafe case is only 5% slower than the optimized single-threaded framework. This implies the overhead of libdispatch is small for the reconstruction job. Comparing the fully re-entrant case to the single threaded case at 16 simultaneous events we see only a 2% drop in throughput. However, as we decrease the number of simultaneous events being processed in a job we see the throughput increase relative to the single-threaded case since more than one core can work on an event at the same time. The maximum speedup was 30%. The reason for this limit is discussed later in this paper.

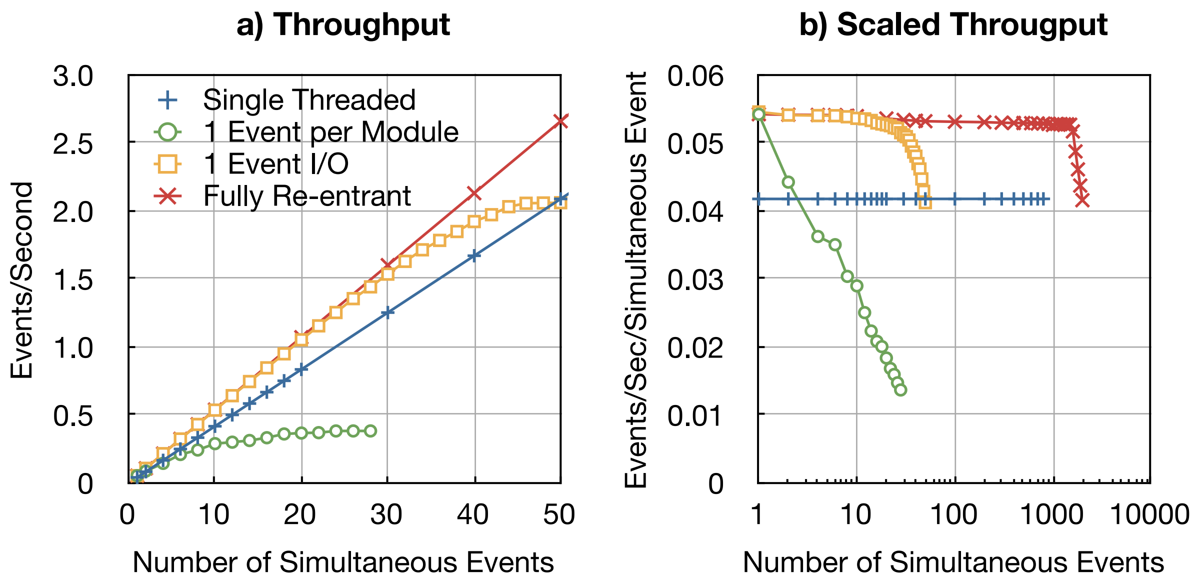

Figure 4. Throughput a) and throughput divided by number of simultaneous events b) measurements for configurations with Producers using usleep.

Figure 4 shows the throughput for configurations using Producers which utilize usleep to wait the appropriate amount of time. As explained earlier, these measurements let us see how well the technology is likely to scale with very large numbers of cores. Figure 4a shows the standard throughput plot while figure 4b shows throughput which has been scaled based on the number of simultaneous events being processed. In figure 4b, a flat horizontal line corresponds to perfect scaling since adding an additional event to process has no affect on how quickly any other event gets processed. The reason scaled throughput was chosen for the y axis of figure 4b rather than the relative throughput plotted in figure 3b is because the single threaded measurements (blue crosses) had to stop at 800 simultaneous events because at that point we ran out of memory on the virtual machine while we could still run the multi-threaded framework well beyond that point. The green circle curve shows the case where modules can only process one event at a time. That case causes the events to form a pipeline where the first event is being processed by the Nth module, the second event by the (N-1)th module and so forth. As you can see, pipelining does not scale for this case and runs slower than the single-threaded case once we exceed two simultaneous events. The reason is all the events get stuck waiting to get a chance to run on the slowest module or behind a very complex event which in general takes longer to process. The yellow-orange box curve shows the case where all the modules that produce data are re-entrant but where both the reading and writing of the data can only process one event at a time. In that case we see the system diverges from perfect scaling at about 25 simultaneous events and then flattens out its throughput at 44 simultaneous events. This implies a need for a thread safe I/O layer sooner rather than later since 32 core machines are already available on the market.

Finally, we see the fully re-entrant case (red x's) maintains its 30% speed up relative to the single-threaded case all the way up to 1600 simultaneous events. The reason the fully re-entrant case stops scaling at 1600 is that is where the program hits the system limit of 2048 threads allowed in one job. Even at 2000 simultaneous events the multi-threaded framework only requires 1.1GB of resident memory and 1/8th of the available CPU.

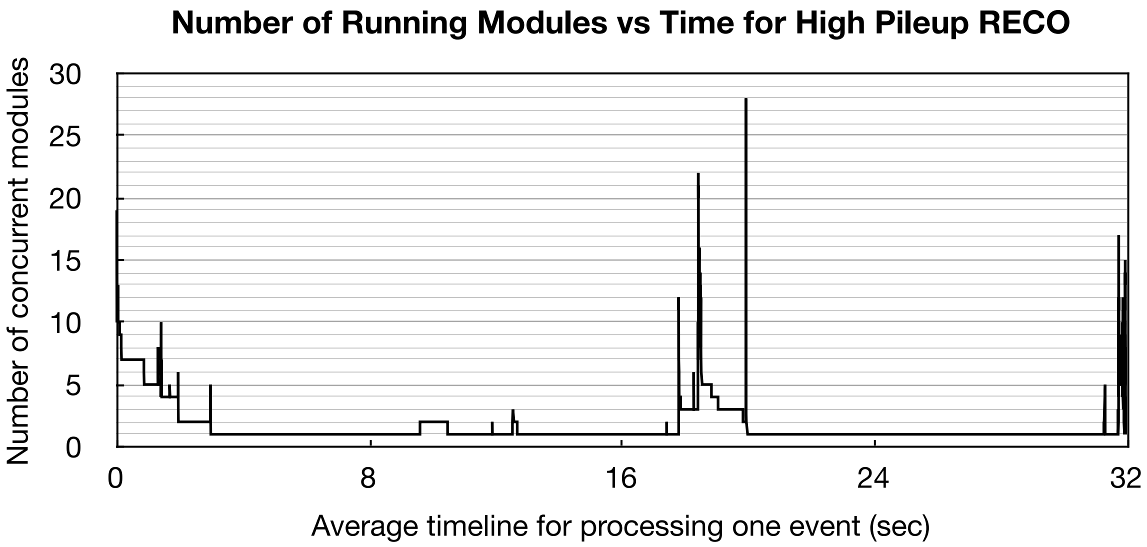

Figure 5. Plot of the number of modules which are running simultaneously at any given moment in time during the processing of a typical high-pileup event.

Figure 4 illustrates why the amount of concurrency is limited to only 30%. The reason for the limitation is due to the dependencies between modules. In figure 4 we see over the time it takes to process one typical event how many modules are running concurrently. At the beginning we have a high level of parallelism for a short time because the modules doing the work are all independent of each other. Then we reach the first long period where only one or two modules can run concurrently. This long period is when the modules associated with track finding and fitting are running. Once the tracks have been found, we get a second short period of high parallelism based on the modules which were waiting for the tracks. After that we have our second long period where only one module is running. That lone module is finding photons which were converted into electron-positron pairs inside the tracking volume. The event processing then ends with a final short burst of parallel activity. It is the two long periods of one or two module concurrency which limits the average concurrency achievable in this setting. All is not lost, however. If we are able to parallelize the tasks internal to these long running modules the average concurrency would be greatly improved. Research has begun on how we can parallelize within these modules[6].

Previously[3] we calculated the amount of expected parallelism for a multi-threaded framework which parallelizes processing of modules. Updating that calculation to using the new module dependencies and timing we estimate a speedup of 29%. Since the calculation does not factor in any overhead for managing the parallelism we conclude that such overhead is minimal for this reconstruction case.

4. Conclusion

As we have shown, libdispatch can be used to create a fine grained threaded HEP framework. The technology scales well to thousands of concurrent events. Using serial task queues allows us to support modules with different levels of thread safety and therefore provides a migration path where we can update modules one at a time. In addition we believe the use of a queuing model is simpler to explain to physicists than thread primitives, such as locks, and therefore easier to use correctly. However, we see that parallelizing modules only gave a modest 30% speed improvement due to the dependencies between modules. We are therefore actively researching how to change the algorithms internal to our track reconstruction to exploit parallelism within the modules themselves.

Although libdispatch has proven to be an excellent technology, we plan to study other technologies, specifically OpenMP[7] and Intel Threading Building Blocks[8], before finalizing our choice by the end of this summer. Once chosen, we will re-engineer CMS' full framework to use threads.

5. References

[1] http://www.jcmit.com/memoryprice.htm

[2] http://www.intel.com/technology/itj/2007/v11i3/3-bandwidth/7-conclusion.htm

[3] Jones C D, Elmer P, Sexton-Kennedy L, Green C and Baldooci A 2011 Multi-core aware applications in CMS *J. Phys.: Conf. Ser.* **331** 042012 doi:10.1088/1742-6596/331/4/042012

[4] http://developer.apple.com/library/ios/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html

[5] Jones C D, Paterno M, Kowalkowski J, Sexton-Kennedy L and Tanenbaum W 2006 *Proc. CHEP 2006* vol 1, ed S Banerjee (India: Macmillan) pp 248-251

[6] Piparo D, Innocente V and Hauth T 2012 Developement and Evaluation of Vectorised and Multi-Core Event Reconstruction Algorithms within the CMS Software Framework *Proceedings of CHEP 2012*

[7] http://www.openmp.org

[8] http://threadingbuildingblocks.org

Acknowledgements

Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.