A Java Fork/Join Blunder

Ed Harned
eh at coopsoft dot com

The F/J framework is a faulty enterprise from the beginning. The basic design is Divide-and Conquer using dyadic recursive decomposition. Simply put, the framework supports tasks that decompose or fork into two tasks, that decompose into two tasks, that decompose… When the decomposing or forking stops, the bottom tasks return a result up the chain. The forking tasks retrieve the results of the forked tasks with an intermediate join()[1]. Hence, Fork/Join. This is a beautiful design in theory. In the reality of JavaSE it doesn't work well.

It doesn't work well because it is the wrong tool for the job. The F/J framework is the underlying software experiment for the 2000 research paper, "A Java Fork/Join Framework."[2] That experimental software is not, has never been, and will never be the foundation for a general-purpose application framework. Using such a tool for application development is like using a pocketknife to chisel a granite sculpture. There is just so, so much wrong with the F/J framework as a general-purpose, commercial application development tool that the author wrote two articles[3], with seventeen (17) points, to illustrate the calamity. This paper is a consolidation of those articles explaining why the F/J framework is the wrong tool for the job.

There are four major faults with the F/J framework:

1. The use of Deques/Submission queues
2. The use of an intermediate join()
3. The use of academic research standards instead of application development standards
4. The use of the CountedCompleter class

1. The use of Deques/Submission queues

The first design fault with the F/J framework is the use of Deques/Submission queues. Deques/Submission-Queues are a feature primarily for

1. Applications that run on clusters of computers (Cilk for one.)
2. Operating systems that balance the load between CPU's.
3. A number of other environments irrelevant to this discussion.

While deques are efficient in limiting contention (there are many academic research papers on work-stealing and deques), there is no hint of how new processes (tasks) actually get into the deques.

[1] An intermediate join() waits for the fork() to complete and should not be confused with a Thread.join() where the later waits for another Thread to finish.
[2] http://gee.cs.oswego.edu/dl/papers/fj.pdf
[3] http://coopsoft.com/ar/CalamityArticle.html
http://coopsoft.com/ar/Calamity2Article.html

> An interesting note is that the architect lists many of the academic papers in the internal documentation of the ForkJoinPool class. The framework appears to be more of an academic exercise than an application development project. Here is a link to the article that details this fault and the list of those papers: http://www.coopsoft.com/ar/CalamityArticle.html#academic

In cluster environments, the network communication cost for processors to load balance queues across the cluster is very high so using a deque for each processor is preferable to FIFO queues. Since deques do not support non-owner task insertion, the use of a submission queue is necessary.

In operating systems, the new process (task, job) is a major event. The operating system does not have to contend with many new processes/sec. (The JVM is a process. No one creates thousands of JVM/sec.) Even the new sub-process (thread) coming from the process does not come in thousands/sec. The operating system can insert a new task into any deque at any time as it essentially controls all access. Since the sub-process has an affinity for its creator, the logical place to put it is back in the spawning process' deque.

Deques are a superb asset for cluster environments and operating systems.

In applications services, new requests (tasks) aren't any different from forked tasks. The new request task, like the forked task, has no affinity for its creator and it doesn't matter where a task goes to await execution[4]. Applications must contend with thousands/millions of requests/sec needing to get into the service. A program with threads constantly querying the submission queue is grossly inefficient. One appalling attempt to resolve this wastefulness is the F/J framework's reuse of the submitting thread as a worker thread. http://coopsoft.com/ar/Calamity2Article.html#submit

The common-pool submission queue is a LIFO queue. When a CompletableFuture asynchronous task attempts to complete all its dependent tasks in the completion chain, it can cause a call-stack overflow.
http://www.coopsoft.com/ar/Calamity2Article.html#common

Java runs on shared memory computers. There is no network communication cost between threads (as there is with clusters) to load balance FIFO queues. Scheduling a task, initial or forked, into the queue with the lowest pending workload is both simple and low overhead. Each work thread can spend most of its time fetching work from its queue and executing that work instead of looking around for work in another queue. When a thread runs out of work in its own queue, then the thread can look in other queues to pilfer tasks. With no submission queue, there are none of the aforementioned problems.

[4] Locality of reference might be a good reason to put the forked task back into the creator's deque especially when processing an in-memory array. However, application services never know what type of structure needs decomposing or how that structure will split and tailoring a service to one specific configuration is bad practice.

Deques/Submission-Queues are futile. Supporting a Deques/Submission-Queue design, with its horrendous complexity and impairments, just to adhere to the original academic research paper is amateur (see also footnote 6.)

2. The use of an intermediate join()

The second design fault with the F/J framework is the use of an intermediate join().

The framework design comes from MIT's Cilk (now the property of Intel). Cilk uses a container for the applications. That is, it has a compiler (or preprocessor) and a separate run-time for the applications. This is a closed design since applications must run inside the container. Since Cilk controls the environment, Cilk can use a pseudo context-switch to support an intermediate join(). In a similar manor, JavaEE uses a container so it can manage threads waiting and switching. If you want to use an intermediate join() then you need a container.

JavaSE is open, no closed container. There is no way a JavaSE application can do a context-switch, or a pseudo context-switch, or anything similar. The intermediate join() in the F/J framework doesn't work because it requires a context-switch.
http://www.coopsoft.com/ar/CalamityArticle.html#faulty

The architect strongly suggests restrictions for using the framework since structural deficiencies, of which the intermediate join() is one, can have unintended consequences.
http://www.coopsoft.com/ar/CalamityArticle.html#special

The architect tried compensating for the lack of a context-switch, required by this intermediate join(), with "continuation threads" in Java7 and "continuation fetching" in Java8. Both those techniques had so many negatives that they're not worth using in many applications.

For some applications, the negatives don't have a chance to materialize and they "appear" to be success stories.

- applications that execute very quickly
- applications that do not use join() with each fork() (asynchronous tasks)
- applications that follow the restrictions put forth by the architect (above)

Nevertheless, a closer look at these successes reveals there is another story underneath.

For applications that are simple and have short paths (like sum()) the problems aren't apparent but when you compare them to a plain, single threading applications then the worms come out of the woodwork:
http://www.coopsoft.com/ar/CalamityArticle.html#inefficient

The asynchronous mode is popular with other languages that use the JVM. Essentially the framework is nothing more than a queue with a pool of threads. Without the join() logic tasks can just as easily use queue.offer(new_task) than new_task.fork(). It essentially does the same thing. One AKKA developer had an application that only submitted new requests to the framework without decomposition or any work. He was ecstatic that it processed millions of request/sec. He would have had better turn-around just executing in his own thread.

Applications using primitive arrays in submissions such as sort, merge, filter, and scan are perfect for the F/J framework since the array can easily process as a DAG. They can follow the rules without any deviation. The "continuation threads" in Java7 and "continuation fetching" in Java8 appear irrelevant since the computing is so fast. However, when you run the application under a load with several other applications that use threads (your normal server environment) then the competition for resources becomes evident. The F/J framework works well in a research atmosphere but nosedives in a production setting.

> The dyadic recursive division technique produces excessive tasks. http://www.coopsoft.com/ar/CalamityArticle.html#slow In that example there are 32 work tasks and 31 extra tasks that threads must address. Now multiply the number of extra tasks by the number of concurrent submissions and you end up with a slow-down rather than a speed-up since useless work clogs each thread pipeline.

It is often hard to see the stall caused by an intermediate join() in Java8 when the thread runs out of elements in its own queue. Multithreading/multitasking developers create extreme programs that can take minutes to run so they can visualize problems in a profiler. The article, http://coopsoft.com/ar/Calamity2Article.html#references, provides downloadable extreme programs that demonstrate the effect stalling threads have on the program. Now imagine how much faster a program would be without stalling threads. That would be a success.

Join() without a context switch is a failure. Plain and simple.

3. The use of academic research standards instead of application development standards

The third design fault with the F/J framework is the use of academic research standards (i.e. the F/J framework design is from the software experiment of an academic research paper) instead of application development standards.

The Fork/Join pool constructs without specifying the maximum number of threads. Instead, a program specifies a desired parallelism level either with a system property or by a flawed[5] default. This is the number of active threads that should run at the same time. That abstract, academic theory sounds good on paper – don't worry about physical

[5] Runtime.getRuntime().availableProcessors() does not always return the number of hardware threads. See this JavaSpecialists newsletter http://www.javaspecialists.eu/archive/Issue220.html

elements just focus on the logical concept[6]. However, like other aspects of this framework, what looks good in a research environment often flops in a production setting. What emerges is that the framework sometimes replaces worker threads that are blocking with internal spare threads. When the framework exhausts those spare worker threads, the framework generates new compensation threads to achieve the desired level of parallelism[7]. (e.g. ForkJoinPool.awaitJoin() used in nested parallel streams (below)) There is absolutely no limit on thread creation[8]. Since there is no guarantee the new compensation thread won't block, the framework can flood the system with hundreds/thousands of compensation threads until the entire box stalls or the JVM runs out of memory. Academic theory flouts sound application programming policy.

The problem persists in classes that use the F/J framework underneath with no way to limit compensation threads. Phaser.arriveAndAwaitAdvance(), CompletableFuture.get() to name two. The article, http://coopsoft.com/ar/Calamity2Article.html#references, provides downloadable programs that demonstrate the excessive compensation thread problem.

Generating excessive threads was such a problem with the intermediate join() in Java7 "continuation threads" the architect abandoned the practice and in Java8 resorted to "continuation fetching." http://www.coopsoft.com/ar/CalamityArticle.html#faulty

In a typical business application environment there may be multiple processes (the JVM is a process) resident in memory. Some of those processes may be multi-user applications with each application using numerous threads (e.g. a box with multiple RMI/Restlet servers.) Conservation of resources is prudent, limiting excessive thread creation is mandatory.

Administrators cannot balance resources without knowing the maximum thread count for an application. Filling out an application impact statement for a server environment by answering the question as to max threads with "maybe 8, or maybe 800 or maybe 8000 or no way to tell" is unacceptable in a professional, commercial enterprise.

Creating compensation threads should be an option. The default should be no compensation threads. If an errant application stalls with waits, then fix the application. If

[6] The concept is rooted in the original research paper where the theory is that F/J tasks are instances of a lightweight class, not instances of a thread. The focus is always on a lightweight execution framework rather than heavyweight threads. While the notion of "lightweight threads" works in an actor context (Scala, etc.), native Java doesn't support such a model.

[7] Throwing threads at the problem only creates more problems since threads are a problem in themselves. http://coopsoft.com/ar/J2SEArticle.html explains the thread problem and the absolute necessity of controlling threads. This framework lacks proper thread control.

[8] The JDK1.8_40 release limits spare threads in the common pool (default 256.) The unrestricted thread creation is causing a huge stink in the application community. However, the spare threads cap logic is the cap only for classes that use the F/J framework common pool.

waits are unavoidable then prudently[9] use the ForkJoinPool.managedBlock() Interface. Creating a massive number of threads makes the errant application's behavior problematic for every other application currently executing since Java threads, along with their native threads and memory consumption, compete with other threads and memory requirements universally. The total number of work threads, not just the active threads, should be part of that option.

> The lack of options is another sore point with this framework. Professional application development frameworks, especially those for managing system-wide parallelism, have copious preferences for controlling functionality usually with a configuration file and startup options. The F/J framework has no preferences and only three constructor options/system properties: parallelism level, thread factory, exception handler. (The framework can inadvertently override, without any notification to the user, both the thread factory and the exception handler when the framework uses submitting threads as worker threads (below).)

The opposite of the capricious compensation thread problem is the worker-thread stall problem. This is when the framework does not create compensation threads.

- A Stream where the lambda blocks – …stream().parallel().map(…) -> { – stalls the worker thread until the blocking finishes. When the stall is short, it appears irrelevant but appearances are deceiving. Since Streams uses the common ForkJoinPool, other submissions pile up behind the blocking application making other users wait unnecessarily.
- In a join()[10], the framework marks the joining task as logically waiting and the worker thread continues fetching and executing tasks from the deque. When the worker thread gets to the bottom of the deque it cannot continue and issues a wait(). Fully half the worker threads can stall when the recursion level becomes long. The article provides a downloadable example of this fault. (MultiRecurSubmit.java) http://coopsoft.com/ar/CalamityArticle.html#references
- The framework cannot detect a run-a-way thread (where the program is stuck in a never-ending loop.) Without detection, there is no solution, not even putting the offending request out of its misery. When all threads stall, the JVM is finished.

Both the compensation thread problem and the worker-thread stall problem are the direct result of the framework not being able to detect an application stall[11]. Without stall detection there is no stall recovery (other than throwing threads at the problem.)

Incredibly, there is no way to time a synchronous request (e.g. invoke(task, timeout)), and, no way to cancel a synchronous request. Any synchronous task that stalls, cripples

[9] Prudent usage means keeping track of how many threads are blocking and restricting new compensation threads when blocking threads become excessive. This is essential for classes that use the framework underneath so users don't get a nasty surprise.
[10] Other methods that call join() – ForkJoinPool.awaitJoin() .quietlyJoin(); ForkJoinTask.doJoin() .get() .invokeAll()
[11] http://coopsoft.com/ar/StalledArticle.html

the request forever. While no timing/cancelling may be acceptable in an academic setting, it is unsuitable for professional, commercial application development.

Furthermore, since there is no notification to Anyone About Anything, no one is aware of the stalling application. If the stalling becomes enduring the only remedy is to terminate the JVM. A real business friendly solution.

The F/J framework does not consider the impact these practices might have on other applications. In the commercial software development community, we call this “not playing nice with others.”

The overall dogma of this framework seems to be, “me first, by myself.”

- The policy of creating copious threads without regard for others, AKA solving my problem by making my problem your problem.
- The policy of disregarding simplicity in favor of implementing academic theories. http://www.coopsoft.com/ar/CalamityArticle.html#complex
- The policy of coding in what the architect calls VM’ese rather than plain Java.
- The policy of ostentatious internal comments and JavaDoc. http://www.coopsoft.com/ar/CalamityArticle.html#academic
- The policy of speed-above-all-else by not providing statistics gathering for threads/deques, options for controlling functionality and other industry standard attributes. http://www.coopsoft.com/ar/CalamityArticle.html#attrb
- The policy of focusing chiefly on massive, in-memory DAG processing (scientific/academic usage) instead of general-purpose application usage.
- The policy of not separating a caller from the external processing so that the framework can only function in a local JVM.
- The policy of not providing stall detection for application tasks. http://coopsoft.com/ar/StalledArticle.html
- The policy of not providing any way to time or cancel synchronous requests.
- The policy of using the calling thread as a worker thread. http://coopsoft.com/ar/Calamity2Article.html#submit

While that doctrine works in a research/academic environment, it is inconsiderate of the commercial application community in practice.

The F/J framework is an oxymoron. Created without the benefit of commercial application development experience but claiming to be an application development framework.

4. The use of the CountedCompleter class

The forth design fault with the F/J framework is the use of the CountedCompleter class.

It seems apparent that the architect gave up trying to support the intermediate join() for the parallel version of Streams in Java8 with the introduction of the CountedCompleter class. http://coopsoft.com/ar/Calamity2Article.html#counted

In a CountedCompleter the first task forks all other tasks up front and uses a manual procedure to invoke a callback method (onComplete()) to gather the results. This is Data Parallelism without the dyadic recursive division. However, there is a major problem with the CountedCompleter class – there is no structure to support it.

> The CountedCompleter class is similar to another Data Parallelism technique called scatter-gather. The scatter-gather technique is a more natural programming model that allows spawning an unbounded number of parallel tasks simultaneously, rather than creating a cascade of binary forks and corresponding joins. The scatter-gather method scatters tasks to every queue in the session to await execution. A thread fetches the task from its queue, calls the compute() method and when the method completes the thread saves (gathers) the return object in the request's repository. After the last task's compute() method finishes, the thread calls the last task's complete() method making available the collection of gathered objects from the request's repository.

Rather than returning to the drawing board to create a structure that supports the new decomposition algorithm the architect butchered the already horrendously complex program to support a different method of recursive decomposition using the new class.

It appears the architect is not giving up the Cilk-like paradigm but is just replacing the intermediate join(). The resulting F/J classes are unequivocally ugly. The CountedCompler class is exceedingly difficult for application programmers to use. The extreme complexity and lack of respect for Object-Oriented programming standards is unprofessional.

Just like attempting to make a land based vehicle operate on water without a redesign, there are complications with this new class. Each solution to each new problem is merely playing whack-a-mole.

One of the biggest glitches derives from the number of tasks that streams can generate.

> In a scatter-gather system, each thread calls the task's compute() method and the compute() returns its result. The thread can save the results in the request's repository and call the next task.
>
> In this Cilk-imitation-without-join program, each thread calls compute() which calls compute()… until a call-stack overflow occurs.
>
> The current solution is to group tasks into a small collection, process that collection sequentially and then do the same for the next group of tasks in a technique called paraquential. The article describes this portmanteau word.

http://coopsoft.com/ar/Calamity2Article.html#para

Another major problem is that tasks hold their own results, which frequently causes Out Of Memory Errors.

In a scatter-gather system, each task returns its results to the calling thread. The thread can save the results in the request's repository letting the garbage collector free the task memory. Execute and be gone.

In this Cilk-imitation-without-join program, the memory increases linearly since each task holds its own results and the forking task keeps a live reference to the task so it can retrieve the result later. Execute and live on.

When all tasks complete, the manually invoked onComplete() callback method sequentially calls get() on each task eventually freeing each task's memory for garbage collection. However, memory exhaustion is common since task memory piles up until all computing finishes.

Again, paraquential processing is the whack-a-mole solution. The program sequentially groups tasks into a small collections. When each collection completes, the manually invoked onComplete() callback method sequentially calls get() on each task eventually freeing each task's memory for garbage collection. Sprints of malloc/free saves the OOME but at the cost of parallelization.

Another problem is defective nested parallel processing support. That is:

```
range(0,outerLoop).parallel().forEach(i -> {
    range(0,innerLoop).parallel().forEach(i -> {
```

In a scatter-gather system, the main thread can start a new asynchronous request for each outer loop process.

Each outer loop asynchronous request can fork as many inner loop tasks as necessary and continue helping with the forked tasks.

All the forked task results for each asynchronous request go into the request's repository until the last task finishes in the asynchronous request.

In the complete stage of each asynchronous request the thread can process the results collection and callback to the main thread where the outcome can add to a store.

The main thread can process the outcome of each asynchronous request as it completes or wait for all asynchronous requests to complete. This same scenario works for any depth of nesting.

In this Cilk-imitation-without-join program, the outer loop threads stall with awaitJoin() [the join problem again] requiring "continuation threads." Those additional threads can number as many as ten times the parallelism level. Therefore, if you set the parallelism level to eight, you may end up with eighty threads. It is also 2-3 times slower than sequential processing of the inner loop. The article provides a downloadable example of this fault.
http://coopsoft.com/ar/Calamity2Article.html#references

JDK1.8_40 partially fixed this problem. However when going to three levels of nesting the excessive threads problem returns vigorously.
http://coopsoft.com/ar/Calamity3Article.html

And wouldn't you know it, the "fix" caused so much of a problem that it was rolled back in future releases – JDK-8080623: CPU overhead in FJ due to spinning in awaitWork
https://bugs.openjdk.java.net/browse/JDK-8080623

There is a proof-of-concept on GitHub that compares sequential, parallel, and scatter-gather proving that nested, parallel processing is feasible.
https://github.com/edharned/nestedTest

Using the CountedCompleter class to emulate scatter-gather is a debacle.

Conclusion

- Deques/Submission Queues are a deplorable choice for a parallel engine.
- Recursive decomposition outside of a controlled environment is a failure.
- Using the software experiment for an academic research paper as the parallel engine for core Java is delusional deportment.
- Emulating a scatter-gather discipline from inside a recursive decomposing entity is a fool's errand.

Every week people find another problem with this faulty framework. There will come a time when treating the symptoms will no longer work and replacing it with software, designed from the beginning as a general-purpose, commercial application development tool, is the only option. Going forward means rethinking the problem. As Albert Einstein once said, "We cannot solve our problems with the same thinking we used when we created them."