A Java™ Fork-Join Calamity

Parallel processing with multi-core Java™ applications

Included in the new Java™ SE 7 release is a so-called lightweight Fork-Join framework. When you take a careful, comprehensive look at what this framework does and how it does it, then you will see that the framework is an inadequate academic experiment underpinning a research paper, not a lightweight framework.

This article explains why (5500 words) [yes, it's getting loooong]

Edward Harned (eh at coopsoft dot com)
Senior Developer, Cooperative Software Systems, Inc.
January, 2011 [updated March, 2015]

This article is part one of a three part series on parallel computing in Java™.

This part deals with the problems of an academic centric Fork/Join framework in a commercial development arena.
Part two deals with the devastating effects the Fork/Join framework has on parallel Bulk Data Operations.
Since those articles have become extremely length, there is a PDF consolidation
Part three deals with the failure to improve the performance of parallel operations.

What is the Lightweight Fork-Join framework?

The lightweight Fork/Join (F/J) framework is a structure that supports parallel programming where problems are recursively split into smaller parts, solved in parallel and recombined, as described in this pseudo code from A Java Fork/Join Framework:

```
  Result solve(Problem problem) {

    if (problem is small)
       directly solve problem
    else {
      split problem into independent parts
      fork new subtasks to solve each part
      join all subtasks
      compose result from subresults
   }
}
```

If only the real solution were this simple. In order to make this work, the framework

developer would have to build a parallel engine inside the Java™ Virtual Machine by ripping apart the current JDK source code, compilers, hotspot, various run times, and implementations — not a likely scenario. Hence, the only practical solution is to build an application server.

If only the lightweight F/J Framework were a blue-ribbon application server. Regrettably, the framework is not a professional application server. It is not even an API. The framework is the academic experiment for the Java Fork/Join Framework research paper. It is inadequate, not lightweight.

The F/J framework simply does not belong inside the JDK. The remainder of this article is a critique only as it applies to to the framework being part of Standard Edition Java™.

Critique

There are a baker's dozen major impediments for this framework being part of the JDK. The F/J framework is:

1. exceedingly complex
2. a design failure
3. lacking in industry professional attributes
4. deceptive
5. misapplying work-stealing
6. misusing parallelization
7. an academic exercise
8. inadequate in scope
9. a faulty task manager
10. inefficient
11. special purpose
12. slow and unscalable
13. not an API

Followed by a Conclusion and a Call to Action.

Exceedingly complex

> "Make things as simple as possible — but no simpler." Albert Einstein
> "Complicated is not a sign that you are clever. It is a sign that you failed." Gilad Bracha
> "There are two ways of constructing a software design.
> One way is to make it so simple that there are obviously no deficiencies.
> And the other way is to make it so complicated that there are no obvious deficiencies." C. A. R. Hoare

The F/J framework classes -

- have many, many levels of inheritance,
- nested classes on top of nested classes,
- instance variables used directly by other classes (known internally as "representation-level coupling among classes"),
- code from the Hackers Delight without comments about what it does,

- homegrown deques and queues instead of standard Java™ Classes,
- and so much more.

Programming techniques use bit switches (like assembler code) and direct memory manipulation (like C code.) The code looks more like an old C language program that was segmented into classes than an O-O structure.

The framework embeds within the calling client that also contains the user servicing code. This back and forth, one calling the other that calls the first, is the classic example of spaghetti code. For over forty years, programmers have known to stay away from entangling calls since the code becomes undecipherable.

The framework relies so heavily on the [sun.misc.]Unsafe class that the F/J framework is not a Java™ program. It is a pseudo-C program. The apparent necessity for writing in the pseudo-C language is to obtain a fair amount of speed by pre-optimizing the code since work-stealing is so inefficient (below.) Naturally, this adds another layer of complexity.

Why does this matter?

- The larger and more complex the software is, the more that can go wrong, the more fragile it becomes, and the shorter the time between failures.
- The code is so horrendously complex most experienced developers will tell you that it is going to fail just for that reason alone.
- It is the job of the JIT compilers to optimize code. No matter how smart and clever a developer may be, no individual can predict the runtime environment. Prematurely optimized code usually cannot be JIT optimized.
- Unsafe class hackery used for performance is sensitive to processor architecture, memory and other hardware dependencies. The code can perform differently between releases of the same architecture which may affect portability and even security.
- Junior Java™ programmers cannot understand the code much less find work-a-rounds when problems arise — what application maintenance is all about. The F/J framework is problematic to understand by anyone other than the original architects. It is doubtful the transient Concurrency Expert Group can maintain this immensely complex code over its lifetime.

This extremely complex code does not belong within the JDK.

A Design Failure

The framework is the academic experiment created for a research paper, designed to follow prior academic research and adhere to the principles of academia. It's primary uses are for fully-strict, compute-only, recursively decomposing processing of large aggregate data structures and in teaching students how to walk down the leaves of a balanced tree. It is not, and never will be, a general-purpose application development tool.

The framework is a failure as designed. Rather than addressing the difficulty

of an academic experiment structure in a commercial development setting, the architects are treating the symptoms. The proposed changes for JDK1.8 make this abundantly clear.

> Since the framework is totally void of any means to handle completion processing in asynchronous requests, general I/O processing, or other actions that may take a lot of time to execute, the framework is severely limited in usefulness.
>
> > The treatment — a new ForkJoinTask, CountedCompleter. Rather than a simple structure to hold intermediate results with a complete() method when computing finishes,
> >
> > - The architects introduce another layer of spaghetti since the Class requires a non-public hook into modified ForkJoinTask exception handling mechanics.
> > - The ForkJoinPool requires class-specific code (**instanceof** CountedCompleter checking) for the Class to function at all.
> > - And, since the framework cannot handle completion processing itself, using this Class requires a huge effort by application developers because the Class is "less intuitive to program." (political speak for "it is a nightmare to program")
>
> Since method join() is appropriate for use only when completion dependencies are acyclic (**D**irected **A**cyclic **G**raph), deadlocks are killing innovative development.
>
> > The treatment — the framework cannot support cyclic dependencies so developers have to use new, even more complex Classes and methods (Phaser, helpQuiesce(), complete(V), markForkJoinTask().)
>
> Since method join() requires a context switch to work properly (See below in Faulty Task Manager) and the framework doesn't do a context switch, the framework spews hundreds of "continuation threads" when recursion exceeds a few levels.
>
> > The treatment — an incredibly complex collection of classes and methods that try to follow the flow as if the joining thread really did a context switch. What happens is that half the worker threads stall with a wait(). (See below in Faulty Task Manager.)
>
> Since Task submissions go into the submission queue, the submission queue severely limits scaling and becomes a bottleneck when lots of clients submit lots of Tasks. (As can happen with Scala/Akka Actors.)
>
> > The treatment — use multiple submission queues and "loosely" associate submission queues with

submitting threads. Not only do we have a highly complex collection of deques designed for work-stealing, we also have a highly complex aggregation of queues just for submitting Tasks.

Since the framework is written in the pseudo-C language it only works well after being JIT compiled. The JDK1.8 parallel mode Concurrent Hash Map has severe performance problems the first time it is used: "... beware that the very first parallel task execution time can be terrible. Really terrible."

The treatment — More and more spaghetti code and use of the Unsafe.class. It won't be surprising if this framework eventually uses the Java Native Interface and supplies assembler modules for each platform on which it runs.

Since Java™ does not have a parallel engine (like Microsoft's® .NET Framework) and the ForkJoinPool embeds within the calling client that also contains the user servicing code, using parallel functions such as sort and filter in JDK1.8 requires passing the ForkJoinPool instance to all methods in a rather clumsy fashion.

The treatment — A static, final common pool variable in addition to the instance pool variable so those calls lacking a ForkJoinPool can use the common pool. While it makes for a prettier implementation, it almost doubles the complexity of the framework since methods have to determine if a common pool exists and have code for both conditions.

By catering to the fringe developers, the software becomes even more complex while loosing focus on the core purpose. It was horrendously complex before, now it is approaching unsoundness. Bad programming is simply a cover-up for bad design.

What is inevitable?

When you use a mountain bike in the Tour de France, the bike breaks down. You can patch it and patch it and patch it until one day it fails catastrophically.

This design failure does not belong within the JDK.

Lack of Industry Professional Attributes

Application programmers learn early on that:

- Programs fail, usually at the worst possible time, consequently error recovery is paramount
- Threads stall, again, usually at the worst possible time
- Software needs tuning (balancing of resources)
- Administrators sometimes need notification (also called alerting)

- Logs are a royal nuisance but necessary
- Interfaces into the structure to monitor performance and control functionality are critical
- The software should be useful across a wide range of computing needs
- A shared copy should be available to conserver resources

The F/J framework has none of these.

- There is no error detection/recovery/reporting (other than thrown exceptions)
- There is no stall detection and of course, no stall recovery (requires monitoring)
- There is no monitoring of the execution environment to ferret out anomalies
- There is only minuscule statistics gathering for performance analysis and tuning
- There is no alerting or logging (A rudimentary level of these is necessary for debugging and for error reporting. You need some way of knowing what happened.)
- Since the internal structure is so entangled with the calling client, there is almost no way to alter control variables
- There is no availability for general application usage. It is for compute intensive tasks only. When used for other than pure, short-lived computations, serious problems with task management arise (below)
- As currently written, the framework cannot be a remote object since there are entanglements of caller/service code.

An application service without industry professional attributes has no place in the JDK.

Deceptive

The F/J Framework is endeavoring to look easy to use but looks can be deceiving. When parallelism is too easy to utilize people will employ it without considering the consequences of their actions.

There is a physical cost associated with dynamically decomposing the work with this framework. These include multiple threads for executing the tasks as well as "continuation threads", volatile variables, atomic instructions, significant memory usage, and CPU intensive operations to name just a few. The physical cost clearly becomes apparent in a production environment.

Can people easily figure out the cost between parallelism with this framework and other techniques? Don't think so.

- Sometimes it is better to use thread pools than decomposition.
- Sometimes it is better not to use threads when concurrency issues are paramount.
- Sometimes it is better to restrict dynamically how many threads participate in decomposition depending on the current load on the system. (There is a better chance of isolating the long running request by keeping it in a narrow window.)

- Sometimes the generation of excessive "continuation threads" can wreak havoc on other processing in the machine.

So, how do you know what this framework is doing so you can make intelligent decisions? In a nutshell, you don't.

> Having a way to monitor and control a framework makes for an intelligent product. An intelligent product makes debugging and performance analysis easier. Performance analysis makes choosing parallelism or another method easier.
>
> This framework is without the means to gather statistics, to alter control variables (bounds limits, timings, threading) and to profile the overall executing image. Therefore, cost analysis is haphazard at best.

While making it easy to write parallel algorithms is a nice feature to have, being able to evaluating the cost is an essential component for finding an effective solution.

A dumb framework has no place within the JDK.

Misapplying Work-stealing

Work-stealing is the second part of a multi-threading configuration where each thread has a private queue from which it fetches work. Many developers are familiar with part 2:

> Underutilized threads take the initiative; they tend to steal work (tasks) from other threads.

Part 1:

> When a thread creates new work (sub-tasks), those sub-tasks go back into the thread's own queue.

Part 1 is the problem when using general-purpose Java™ applications with the F/J framework. To understand why, a little background is necessary.

What is work-stealing good for?

> Operating systems and closed/cluster (requiring a compiler and separate runtime) application Task Managers like Cilk. There, new sub-tasks have an affinity for the environment from which they came. They may share storage, handles, caches, etc. so it is logical to put each new sub-task back into the creator's queue. The creator "owns" all those tasks.
>
> When a processor has no work and it can find a task on another processor's queue, it is practical for that processor to "steal" that task. Executing another processor's task is better than being idle. That is good.

What is work-stealing not good for?

> General-purpose, SMP Java™ applications running the F/J

framework. There, the framework creates an initial task and puts it into a submission queue. Since the framework is incapable of placing the task into the queue of an underutilized thread, all threads have to wake up and blindly go looking for work somewhere.

When a task creates sub-tasks, those new sub-tasks go back into the thread's own queue. However, the new sub-tasks do not have an affinity (handles, caches, etc.) with their creator. Java™ applications cannot assume anything about caches or operating system/hardware properties. Therefore, the creating thread does not "own" the new sub-tasks. There is no merit putting them back into the creator's queue especially if those sub-tasks execute very quickly, do significant work, uses I/O, or are not fully-strict (i.e. not processing a **D**irected **A**cyclic **G**raph.)

When a thread has no work then it blindly goes looking around for an available task. However, this is not stealing, as threads do not "own" tasks. Since the load is unbalanced -

- threads can waste a significant amount of computing time just looking for work and
- thread starvation is a real problem when the computation is modest.

What is the alternative? Work-sharing is a superior technique for general-purpose Java™ applications.

In a work-sharing environment, the scheduler tries to spread the work among possessors in hopes of distributing the work to underutilized processors and to keep the load balanced.

- When a new task starts or an existing task creates a sub-task, the scheduler puts it into the queue with the least number of pending tasks. There is no separate submission queue.
- When a thread has no work in its own queue, then the thread tries to select a task from another thread's queue with the highest pending work. However, this is not stealing since threads do not "own" tasks.
- Load balancing is efficient since it is better for a thread to process tasks from its own queue then to go looking for work somewhere else. Therefore, the overhead of scheduling tasks is acceptable.

A botched framework has no place within the JDK.

Misusing Parallelization

Fork/Join is not parallel processing.
Fork/Join is just one aspect of parallel processing.
Dynamic decomposition is just one facet of Fork/Join.
Recursive decomposition is a derivative of dynamic decomposition.

Dynamic decomposition has a very narrow performance window such as:

- If you need to sum an array of one hundred integers then you would do it sequentially; one million integers, then it pays to decompose the work with Fork/Join.
- If you have *N* processors and 8(*N*) concurrent requests, then using one thread per request is often more efficient for throughput. The break-even point for decomposition is somewhere just greater than 8(*N*), depending on the application. The logic here is simple. If you have *N* processors available and you split your work accordingly but there are hundreds of other tasks ahead of you, then what's the point of splitting?

Recursive decomposition has an even narrower performance window. In addition to the above, it only works well:

- on balanced tree structures (DAG),
- where there are no cyclic dependencies,
- where the computation duration is neither too short nor too long
- where there is no blocking

Why would anyone use the recursively decomposing F/J framework, with its very narrow measure, as the engine for parallel programming support (lambda, bulk operations, JEP103) in JDK1.8? The JDK1.8 engineers say they don't have anything else and since the F/J framework is already inside the JDK, they use what they have.

Why isn't there anything else? Isn't Oracle® powerful enough to create a parallel engine for Java™? Microsoft® did just that for C# with its .NET Framework for parallel programming.

By having the F/J framework inside the JDK, Java™ kernel engineers are misusing the F/J framework rather then developing the proper tools for parallel development.

What is foreseeable?

> Oversubscription happens when a parallel operation uses a parallel operation, which uses a parallel operation … With the misuse of resources by this framework (see below in "continuation threads") the system can be flooded with waiting threads and context switches.
>
> Load imbalance happens when many different applications models use the framework concurrently. Some tasks with I/O and those tasks using shared data can block the faster compute only tasks. Rather than the basic Fork/Join decomposition application, there could be a plethora of uses some having nothing to do with recursive decomposition of aggregate data structures.
>
> Support for true parallelization becomes impossible with a recursive decomposing framework. A true parallel engine provides support for parallelizing regions of code where every statement in a region may execute concurrently.

```
Parallel.Invoke(() => ComputeMinimum(),
                () => ComputeMedian(),
                () => ComputeMaximum());
```

> And support for common parallel functions such as **Parallel**.**for**() and **Parallel**.**forEach**() becomes hopeless since the F/J framework needs to warm up before it can be useful. (The code only works well after compilation; essentially the same problem as the JDK1.8 parallel mode Concurrent Hash Map)

What is the answer?

> Get this inappropriate framework out of the JDK and force Oracle® to develop a proper parallel engine.

The F/J framework is not a parallel engine and does not belong within the JDK.

Academic exercise

The work-stealing algorithm using deques is only of academic interest. There are myriad papers devoted to work-stealing on the web, but none of them are for general-purpose application programming.

Assuming locality of reference by placing sub-tasks back into the deque from which they came and assuming that the same thread will process the sub-task is totally without merit. It may look good on paper but the Java™ compiler, JIT compiler, and garbage collector rearrange memory. Additionally, Java™ applications sometimes run under two virtual machines (mainframes run Linux™ under VM and Linux™ runs the JVM) and there is no guarantee of mapping of Java™ threads to O/S threads (see JRockit and others.)

The extolled work-stolen count is of theoretical interest only. To be of industry usage the count must include whether the owner thread was running or blocking, the queue length at the time of filch, and the time it takes to process a task.

The F/J framework internal method comments site numerous scientific papers as the basis for their existence. Although the code adheres to the highest academic standards, those credentials are not foremost in building a fast, scalable, developer-friendly application service. Moreover, the "Implementation Overview" in ForkJoinPool describes complexity that is unintelligible by mere mortals breaking the time honored dictum that program documentation is for programmers to help them find work-a-rounds when bugs arise.

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." Martin Fowler

Why is this a concern?

- The focus is on prior academic research and adherence to scholarly principles, not on wide-area usefulness and maintainability.
- The design must follow the textbook rather than being innovative.
    - The work-stealing technique, while scholarly, is for very restricted practice (strict, fully-strict, terminally-strict computations) and requires compiler/runtime support (closed environment) to work well.
    - There are no alternatives to work-stealing such as scatter-gather since alternatives would not conform to the curriculum.
- The true audience is academia rather than the general programming community since the F/J framework is a modification of the original academic experiment.

The F/J framework is only an academic venture and does not belong within the JDK.

Inadequate in Scope

The over utilization of the Unsafe class precludes ever being able to downgrade the F/J framework to run on JavaME.

The F/J framework cannot run in a significant portion of the JavaEE environment. EJB and Servlet applications usually call an RMI Server when using threading frameworks. However, the F/J framework cannot function with RMI, IIOP, POA, or other networking technologies.

The entanglement of caller/server code forestalls the F/J framework from running as a server.

The F/J framework is only designed to work in a trivial segment of the computing world:

- on JavaSE only,
- using humongous data structures,
- on high-performance work-stations (16+ CPU's without hyper-threading)
- doing compute-only work,
- with no task I/O, no inter-task concurrency and no networking.

and only for aggregate operations on collections of elements that have a regular structure. That is, you must be able to express things in terms of apply, reduce, filter, map, cumulate, sort, uniquify, paired mappings, and so on — no general purpose application programming here.

The F/J framework is only designed to work for one request at a time.
A hypothetical example:
Let’s say there are multiple F/J worker threads and
there are many user threads submitting requests:

- A worker thread picks up the first request and eventually the other worker threads pick up the forked tasks from the first request.
- Other submitted requests wait in turn for a worker thread to empty its own deque, find no work in other worker deques, find a waiting

request in the submission queue, and start forking.

- Other threads may do the same. That is, they pick up one request at a time, finish that request, and help finish other requests before looking for a new request.
- As more and more requests fill up the submission queue they back up. This may result in a significant backlog since there is no way to time any request, no way to cancel synchronous requests, and no way to head queue the hot request. (Work-Stealing as implemented with FIFO/LIFO deques is not conductive to inserting requests directly into a worker deque.)
- Therefore, for multiple submissions there is little, if any, benefit to the F/J framework. Using one thread per request (like a thread pool) without forking is often more efficient for throughput. Don't believe it? Try it yourself. (You can download the source code for a F/J vs. Thread Pool demo below.)

This lack of scope only qualifies the F/J framework for a seat in a classroom, not as the core of Java™ parallel computing.

A Faulty Task Manager

The Fork-Join technique splits the work into fragments and joins the results together. A subordinate practice that allows an intermediate join() for each fork() can only work successfully in a controlled environment.

A fatal flaw with the F/J Framework is that it is trying to manage an intermediate join() in a task outside of a controlled environment. Only the Operating System (O/S) or a pseudo O/S can manage an intermediate join() in a task. (Whether they're called tasks or processes is all the same thing.)

Briefly: When a task requires a resource or service (memory, file handle, join()) it goes to the O/S, the ultimate controlled environment. If the request will put the running task in a waiting state, the O/S switches the task context to the suspended queue and switches in a new task context from the active queue for the processor to execute.

An Application Task Manager (pseudo O/S) like Cilk or jCilk can only work when the application goes through **it** (using a compiler and runtime that creates a controlled environment) to request resources or services like join(). That way the manager knows to switch out/in on pseudo processors it controls.

Basic application frameworks simply have no way of knowing the application execution environment therefore, there is no way on this great, green earth an application framework can do a context switch to support an intermediate join().

The F/J Framework's JDK1.7 answer to an intermediate join() without a context switch is "continuation threads." That is, the framework creates additional, temporary threads to continue fetching application tasks from the deques and submission queue while the joining thread enters a wait state. That can result in a huge number of threads for each request. (StackOverflow has one report of over 700 continuation threads needed to complete one call.) — An unmitigated failure.

The F/J Framework's JDK1.8 answer to an intermediate join() without a context switch is "continuation fetching." That is, the framework marks the joining task as logically waiting and the thread continues fetching and executing tasks from the deque. That can result in a stall (thread in a wait state) when the recursion level becomes long. The proof is in the profiler. (You can download the source code for MultiRecurSubmit.java demo below.)

> Additionally, there is the problem of errors/exceptions in subsequently fetched tasks.
>
> - Since the stack holds the entire flow, can the F/J Framework handle stack-overflow problems?
> - Can the F/J Framework find the stack entries for the first joining task, or the second, or the third for error recovery?
> - Can the F/J Framework back out and recover other associated tasks in other deques?
>
> The answer is **no** to all of the above.
>
> Let's not forget the asynchronous use of outside resources by tasks. Often a resource will associate a request with the thread that made the request:
>
> ```
> Thread caller = Thread.currentThread();
> ```
>
> When the first tasks does a fork-join while the asynchronous call proceeds, the same thread will execute the forked task. If that new task fails,
>
> - what happens to the asynchronous request to the outside resource since they're both associated with the same thread?
> - What happens to handles, and registers, and memory since there is no clean break (context switch) between tasks?
>
> The answer is **uncertainty** to all of the above.
>
> The F/J Framework's current handling of tasks for JDK1.8 is an unmitigated failure. Soon to come is the Lambda-fication of the framework that should prove interesting, but unsuccessful.

The intermediate join() isn't the only source of excessive thread creation with this framework.

> Introduced with JDK1.7 is the Phaser Class. To quote the JavaDoc:
>
> > "Phasers may be tiered (i.e., constructed in tree structures) to reduce contention. Phasers with large numbers of parties that would otherwise experience heavy synchronization

> contention costs may instead be set up so that groups of sub-phasers share a common parent."

> Not mentioned in the JavaDoc is that when using a large number of parties the framework creates "compensation threads" to continue fetching application tasks from the deques and submission queue while each original thread waits for sub-phasers to arrive. These "compensation threads" can be so excessive they make the "continuation threads" problem, above, seem meek. See for yourself. (You can download the source code for TieredPhaser.java demo below.)

Introduced with JDK1.8 is the CompletableFuture Class. To quote the JavaDoc:

> "A Future that may be explicitly completed (setting its value and status), and may include dependent functions and actions that trigger upon its completion."

> Not mentioned in the JavaDoc is that when using a large number of dependent functions with a get() method, the framework creates "compensation threads" to continue fetching application tasks from the deques and submission queue. Once again, these "compensation threads" can be so excessive they make the "continuation threads" problem, above, seem meek. See for yourself. (You can download the source code for MultiCompletables.java demo below.)

The problem is tasks needing to wait. When a task needs to wait for any resource, the framework must detach the task from the thread. This framework cannot detach the task from the thread so it tries other means again and again and again.

Patching the F/J Framework repeatedly is like playing whack-a-mole. There always comes a time when no amount of patching will work since it never addresses the fundamental flaw —

Any wait without a context switch is an absolute, total, complete failure.

Is there an alternative? Certainly.

Using a separate structure to hold intermediate results rather then using an intermediate join() is a simple alternative and it has commendable benefits:

- It doesn't require the horrendous complexity of Task Management.

- It can easily support cyclic dependencies.
- It can easily handle synchronous and asynchronous completion processing.
- It provides
  - a way to track the request throughout it's lifetime to aid in error detection/recovery.
  - the ability to time and cancel requests.
  - the capability to gather statistics for tuning.

The faulty F/J Framework has no place within the JDK.

Inefficient

Assume a java.util.concurrent.RecursiveTask that sums a long array in the compute() method.

```
Sum left  = new Sum(array, low, mid);
Sum right = new Sum(array, mid, high);

left.fork();
long rightAns = right.compute();
long leftAns  = left.join();

return leftAns + rightAns;
```

At level 1, we split the array creating two new Tasks for the next level. We issue a fork() for the left and compute() for the right. The fork() adds a Task to the same worker thread deque. The compute() places an item on the stack and continues execution.

In the second-level compute(), we do the same as above.

…

Eventually we get to the bottom where compute() actually sums the array returning a value for the last item on the stack. That last item can then issue a join() for the last fork() it did. What we've accomplished is creating Tasks for subsequent fetching by the forking worker thread and Tasks that other worker threads may steal.

Since all forked Tasks go into the same worker thread deque, the stealing worker threads will fight each other at the top of the deque over the forked Tasks. Not only is this woefully inefficient it also involves contention between the stealing worker threads. Contention is exactly what work-stealing is supposed to avoid. But in reality, contention is what the work-stealing algorithm produces. What we really have is a single queue with a pool of threads fighting over elements. The larger the array, the more apparent this becomes.

> In A Java Fork/Join Framework, section 4.5 Task Locality, "[the framework] is optimized for the case where worker threads

> locally consume the vast majority of the tasks they create. … As seen in the figure, in most programs, the relative number of stolen tasks is at most a few percent." Even the original research paper shows work-stealing is inefficient.
>
> This framework can never function efficiently in a high-performance environment. A hypothetical example:
>
> > Let's say there are 10 worker threads, the forking generates 100 Tasks, and it takes 1 millisecond to process the actual computation. Therefore, it would take 100 milliseconds to process the Tasks sequentially.
> >
> > Using a load balancing algorithm that evenly splits the work among all the worker threads, the time to complete the work is about 10 milliseconds.
> >
> > Using work-stealing, if one worker thread does 80% of the Tasks and other worker threads do the rest, then the time to complete the work is about 80.3 milliseconds.
> >
> > Then there is the problem of starvation. A few worker threads become saturated with work leaving other stealing worker threads starving.
> >
> > For a student doing calculations on a laptop — so what. For an enterprise server doing transaction processing — oops.

Is there a better way? Absolutely.

> It's called load balancing with the scatter-gather method (below.)

This convoluted, inefficient framework has no business in core Java.

Special Purpose

Brian Goetz wrote, Stick a Fork In It, introducing the ParallelArray classes and the Fork/Join framework. He recognized the limitations of these classes for "aggregate data operations" only. As Brian suggested, the F/J framework is exclusively for number crunching.

The F/J framework has none of the attributes of a professional, multi-use application server (see above.) That's why it should not be used for other than its original purpose. But there is no way on this earth developers will adhere to unreasonable restrictions. (Some don't even follow reasonable rules.)

Recommended restrictions:

- must be plain (between 100 and 10,000 basic computational steps in the compute method),
- compute intensive code only,

- no blocking,
- no I/O,
- no synchronization

Do you wonder why > 100, < 10k computational steps?

> 100 has to do with the work stealing problem. All forked Tasks go into the same deque making other threads search for work. When the threads encounter contention they back off and look somewhere else. Since there is no work anywhere else they try the same deque again, and again, and again until the forking thread finally finishes the work all by itself. You can see the proof by downloading the source code for Class LongSum.java below. Hence, run slow or there will be no parallelism.

> < 10k has to do with the join() problem. Since the F/J framework cannot do pure Task Management (see Faulty Task Manager, above) with Tasks actually waiting independently of threads when they call join(), the framework has to create "continuation threads" to avoid a halt. There can only be a limited time before it all falls apart. Hence, run fast or die.

As for the other restrictions — any actual thread blocking, without the framework's knowledge, would probably stall the entire framework. Since there is no way to view the internals, to cancel synchronous requests, and absolutely no way to time any request, the only alternative to deal with a stall is to kill the JVM. A real business friendly solution.

Misuse of the framework has already started:
Java Tip: When to use ForkJoinPool vs ExecutorService - JavaWorld

Soon people will find uses for this framework no one ever imagined. It's the first Law of application software development which follows the Law of unintended consequences.

The F/J framework does not belong in core Java.

Slow and Unscalable

Speed

> Speed is relevant to other Fork-Join products. In this case, speed compared to the TymeacDSE project. A single summation call puts the TymeacDSE product 3-4 times faster than the F/J Framework. (You can download the source code below.)

> Since the products cannot directly compare — F/J framework has limited scope, TymeacDSE is a full feature application server — the comparison must include the internal structures.

> > F/J Framework uses an inefficient (above) textbook work-stealing paradigm that makes threads go looking for work and unnecessarily splits the work into useless tasks.

Let's say there is an array of 1M elements and the sequential threshold is 32K, then it takes 32 tasks to sum the array in parallel. However, using the code example above, the method generates an extra 31 tasks in six levels just to recursively split the array into two parts and join(). That is almost double the toil for threads that could otherwise be doing useful work.

Threads cannot pick up new requests until they completely finish all prior work. (That is, they have to exhaust all the deques of work before they can go to the submission queue.) Since there is no separate structure to hold intermediate results, each split (fork()) must wait (join()) until subsequent splits complete, loosely resembling this:

- fork()
- join()

TymeacDSE uses a load balancing, scatter-gather algorithm that feeds work to threads immediately. There is no separate submission queue and there are no split/wait tasks. Thus, the 1M array example above takes simply 32 tasks to sum the array in parallel. TymeacDSE uses a separate structure for all calls both to track the work and to hold intermediate results. Therefore, TymeacDSE allows this:

- fork() as many times as necessary to spread the work amongst all processors, returning an intermediate result to the server for each computed value.
- When complete, finish the work using all the intermediate results. Which may comprise forking the results array.

What this means for applications like sorting and map/reduce is that TymeacDSE can use the full collection of processors to sort or map the data and then use the full set of processors to merge or reduce the intermediate results. Using all the processors without waiting (join()) makes TymeacDSE fast.

Unscalable

Generally, scaling implies that as you add more processors/threads to the equation, you decrease the time to completion. The F/J Framework structure precludes scaling.

- The entanglement of client call/server processing,
- the spare threads necessary to help with the join() waiting problem (above),

- as well as the work stealing code
  - (threads need to serially search for work among all deques,
  - threads consume the vast majority of the tasks they create so adding more processors/threads has no effect on parallelism)

only works well on a small number of processors. This design can never scale to hundreds or thousands of processors. The overhead and thread starvation would nullify the benefits of additional processors.

TymeacDSE scales to thousands of processors. It's simply a matter of design. When the architect separates the client caller, the user processing, and the server management then anything is possible. Building an index over the management structure of threads/queues is easy when there are no code entanglements.

Speed and scalability are the lynchpins of parallelization. A slow and unscalable application service does not belong within the JDK.

Not an API

The F/J Framework is part of the SE7 API Specification but the package is deficient in "characteristics of a good API" (Joshua Bloch). The package is actually an independent entity masquerading as an API.

None of the base classes in the F/J framework package can live alone nor do the base classes have any usage outside the framework. The main pool class extends AbstractExecutorService, which makes the framework an application server — an independent entity.

Other classes in the JDK extend the Executor Service (ThreadPoolExecutor, for one) but they cannot live alone. Those classes are components in user-built application services; they are not a service in themselves.

The F/J framework is a stand-alone entity. Most base support methods are final, to prevent overriding. The base package classes are a server unto themselves, they are not individual API's for programmers to extend and build another service.

A clear example of independence is when non-Java™ languages such as Scala, Akka, Clojure, X10, Fortress, and others use the framework without extension.

An independent server does not belong within the JDK.

Conclusion

The F/J framework is lacking as an API

- It is deficient in "characteristics of a good API" (Joshua Bloch)
- Even with closures support (JDK8) the amount of extra classes (486

classes and interfaces in the "extra" package) necessary for this feature makes it difficult to use, creates bloat, and adds to the entanglements of caller/servicing (spaghetti) code

The F/J framework is severely inadequate as an application service

- It is only available as an embedded server, no remote object access such as RMI
- It is only recommended for aggregate data operations.
- It is only recommended for users with very large data structures (humongous arrays)
- It is only useful for machines with profuse processors (16+ recommended)
- It is only useful for one request at a time
- It is only useful for strictly limited processing (recommended)
    - must be plain (between 100 and 10,000 basic computational steps in the compute method)
    - compute intensive code only
    - no blocking
    - no I/O
    - no synchronization
- It has no logging or alerting
- It has no error/stall recovery
- It has no ability to display/alter the current execution status
- It relies on academic correctness rather than maintainability
- It lacks scope
- Its multi-tasking is faulty
- Its work-stealing is misapplied
- It is deceptive
- It is inefficient
- It cannot not scale to hundreds of processors
- It is complex to the point of being incomprehensible
- There is no user manual, just JavaDoc

The minuscule benefits of the F/J framework do not outweigh

- the bloat this experiment adds to the Java™ run time for non fork-join users
- the restrictions for developers looking to parallelize their software especially with JDK1.8
- the difficulty in finding bugs with all the embedded classes (ParallelLongArray has seven levels of inheritance)
- the affront to the user community for extending core Java™ with an inferior product (If the architects wanted a Cilk-like product as an extension to Java™, then they should have built one and then made a light version.)

The F/J framework is an inadequate academic experiment underpinning a research paper, not a lightweight framework. It does not belong in the JDK.

A Call to Action

What can we do about this calamity?

Solution # 1
What we all want is a parallel engine for any purpose integrated into the JVM, supporting timed and interruptible calls, that is decently performing, and that is simple to use like this:

```
if (problem is small)
    seq.doWork();
else {
    parallel.doWork();
}
```

We don't have one and are unlikely to see one in the near future since it requires Java compiler/runtime alteration to support an internal, controlled multithreading environment.

Solution # 2
For second best, we want a limited parallel engine within the JDK. It should be a hidden part of the Special Purpose Implementations of the Collections Framework. It is for the single purpose of parallelizing aggregate data structures (like sorting.) Hidden is important so programmers will not try to use it on their own and the engine can support the appearance of fully-strict while being open to other techniques. Usage is only by supplied classes (such as the ParallelArray classes mentioned by Brian Goetz in his Stick a Fork In It article.)

Solution # 3
Additionally, we want parallel application services outside the JDK supporting the full range of Fork/Join. They are for any purpose. They are decently performing, they support timed and interruptible calls, they are simple to set up and understand, and they run embedded as well as remotely to conserve resources. They cannot be as clean looking as the first choice but they should be straightforward such as this:

```
if (problem is small)
    seq.doWork();
else {
    setUpParametersForCall();
    parallel.doWork();
}
```

What we have now is attempting to look like solution # 1 to users but in doing so it introduces complexity that only dooms it to failure.

It is time for the commercial application community to tell Oracle® to stop

promoting this calamity as the answer to parallelizing applications and to implement proper solutions.

References

Download the source code for the article here.

Download the PDF consolidation

A Java Fork/Join Framework — Doug Lea

Part two of this series, A Java Parallel Calamity
http://coopsoft.com/ar/Calamity2Article.html

Part three of this series, A Java Parallel Failure
http://coopsoft.com/ar/Calamity3Article.html

JDK1.8 Concurrent Hash Map on Concurrency Interest List
http://cs.oswego.edu/pipermail/concurrency-interest/2012-August/009711.html

JDK1.8 Java Extension Proposal 103
http://openjdk.java.net/jeps/103

How To Design a Good API and Why It Maters — Joshua Bloch

When to use ForkJoinPool vs ExecutorService — JavaWorld
http://www.javaworld.com/javaworld/jw-10-2011/111004-jtip-recursion-in-java-7.html

700 continuation threads —
http://stackoverflow.com/questions/10797568/what-determines-the-number-of-threads-a-java-forkjoinpool-creates

Java theory and practice: Stick a fork in it, Part 1
http://www.ibm.com/developerworks/java/library/j-jtp11137.html

The Cilk-Plus Site — http://software.intel.com/en-us/articles/intel-cilk-plus/

The Java based jCilk Site — http://supertech.csail.mit.edu/jCilkImp.html

The JRockit Site —
http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html

TymeacDSE article — A Java Fork-Join Conqueror

Work stealing papers:

www.cs.rice.edu/~vs3/PDF/ppopp.09/p45-michael.pdf
www.eecis.udel.edu/~cavazos/cisc879-spring2008/Brice.pdf
home.ifi.uio.no/paalh/publications/files/mucocos2010.pdf

Oracle® Promotion

Fork/Join (The Java Tutorials)
Java 7 Fork/Join Framework Initial Look, and Resources

Preparing to Compute Using the Fork/Join Framework, Part 1
Fork and Join: Java Can Excel at Painless Parallel Programming Too!

About the Author

Edward Harned is a software developer with over thirty years industry experience. He first led projects as an employee in major industries and then worked as an independent consultant. Today, Ed is a senior developer at Cooperative Software Systems, Inc., where, for the last fourteen years, he has used Java™ programming to bring fork-join solutions to a wide range of tasks.