

Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

October 10, 2011

Async Tasks in C++11: Not Quite There Yet

Posted by Bartosz Milewski under [C++](#), [Concurrency](#), [Multicore](#), [Multithreading](#), [Parallelism](#), [Programming](#)
[\[13\] Comments](#)

i

33 Votes

If you expected `std::async` to be just syntactic sugar over thread creation, you can stop reading right now, because that's what it is. If you expected more, read on.

Don't get me wrong, `std::async` combines several useful concurrency concepts into a nice package: It provides a `std::future` for the return value, and hides the `std::promise` side of the future. It also provides options to run a task synchronously. (See the Appendix for a short refresher.)

But *tasks* have a slightly different connotation in parallel programming: they are the basic blocks of task-based parallelism. And C++11 tasks fall short on that account.

Task-Based Parallelism

Tasks are an answer to performance and scalability problems associated with threads. Operating system threads are rather heavy-weight; it takes time and system resources to create a thread. If you have an algorithm that naturally decomposes into a large number of independent computations, a.k.a. tasks, you'll probably get your best performance not by creating a separate thread for each task, but by adjusting the number of threads depending on the amount of parallelism available on your particular system, e.g., the number of cores and their current loads. This can be done by hand, using thread pools and load balancing algorithms; or by using task-based systems.

In a task-based system, the programmer specifies what *can* be run in parallel but lets the system decide how much parallelism to actually use. The programmer splits the algorithm into tasks and the runtime assigns them to threads — often many tasks to a thread.

There are many implementations of task-based parallelism with varying language support. There's the Cilk language which pioneered this approach; there's the built in support in Haskell, F#, and Scala; and there are several C++ libraries, like Microsoft PPL or Intel TBB.

Unlike thread creation, task creation is supposed to be relatively inexpensive, letting the programmer explore low-level granularity parallelism and take advantage of multicore speedups.

At the center of task-based systems are work-stealing queues. When a thread creates tasks, they are initially queued on the processor (core) that runs the thread. But if there are other idle processors, they will try to steal tasks from other queues. The stolen tasks are then run in parallel.


Notice that tasks must be able to migrate between threads. What's more, efficient use of OS threads requires that tasks that are blocked, for instance waiting for I/O or waiting for other tasks to finish, should be taken off their threads, so that other tasks may reuse them.

C++11 Tasks

My expectation was that C++11 “tasks” that are created using `std::async` should be abstracted from threads, just as they are in task-based parallelism. When I started preparing a [video tutorial about tasks in C++](#), I wrote a simple program to demonstrate it. I created async tasks using the default launch policy and waited for them to complete. Each task slept for one second and then printed its thread ID.

```
int main()
{
    std::cout << "Main thread id: " << std::this_thread::get_id()
               << std::endl;
    std::vector<std::future> futures;
    for (int i = 0; i < 20; ++i)
    {
        auto fut = std::async([]
        {
            std::this_thread::sleep_for(std::chrono::seconds(1));
            std::cout << std::this_thread::get_id() << " ";
        });
        futures.push_back(std::move(fut));
    }
    std::for_each(futures.begin(), futures.end(), [](std::future & fut)
    {
        fut.wait();
    });
    std::cout << std::endl;
}
```

The results were surprising. The first six tasks executed in parallel, each in its own thread. But the rest of the tasks executed in the main thread one after another, separated by 1 second intervals. (Note: this behavior was fixed in v 1.7 of `Just::Thread` — read on).



A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Work\C++11\Release\C++11.exe". The command prompt displays the text "Main thread id: 1" followed by a sequence of numbers: "2 4 3 6 5 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1". The window has a standard Windows interface with minimize, maximize, and close buttons in the title bar.

The output of the task test

This approach to parallelism obviously doesn't scale very well.

Then I wrote another program that lists directories recursively, creating a separate async task for each subdirectory. But this time I explicitly requested launch policy `launch::async`, which guarantees that each task will start in a new thread. This program worked up to a point, but when I tried to list my whole disk, it failed by exhausting Windows' limited thread creation capability. Again, this approach doesn't scale very well.

What was even worse, when the program didn't fail, it performed better with `launch::deferred` policy, which forces all tasks to be executed serially, than with the `launch::async` policy. That's because thread creation in Windows is so expensive that it can easily nullify performance gains of parallelism (although Windows 7 supports user-level threads, which might bypass these problems).

My first reaction was to blame Anthony Williams for badly implementing the `Just::Thread` library I was using. When he assured me that it was Standard compliant, I turned to Herb Sutter and Hans Boehm for confirmation and they sided with Anthony. It turns out that there are serious problems that prevented C++11 from standardizing task-based concurrency.

The Problems

The foundation of task-based parallelism is the ability for tasks to share threads and to migrate between threads. This sharing and migration must be transparent.

The requirements for the default-launch tasks are the following:

- The runtime can either run such task asynchronously or synchronously
- When it's run synchronously, it should be run in the context of the parent thread
- When it's run asynchronously, it should behave as if it were run on a separate thread

Strictly speaking, a task could always call `this_thread::get_id()` and fool any attempt at thread sharing or migration by discovering the ID of the current thread. In general, the namespace `std::this_thread`, which also contains sleep functions, is thread-bound.

But let's suppose that we only require that asynchronous tasks behave as if they were run on separate threads, except when they call functions in the `this thread` namespace. There are still several problems.

Thread-Local Variables

C++11 introduced a new `thread_local` storage qualifier. A thread-local variable is separately initialized and destroyed in every thread. It must not survive thread termination. This requirement complicates thread sharing.

In our exchange, Hans Boehm clarified the termination requirement for tasks: Thread-local variables must be fully destroyed before the owning thread returns from calling `get` or `wait` on a future produced by the corresponding `std::async`; or before the destructor of that future returns, whichever comes first.

This actually leaves some wiggle room: A thread could be reused if the runtime guarantees that thread-local variables of terminating tasks are correctly destroyed. Unfortunately, this might be impossible if the programmer calls OS-specific API, like Windows' `TlsAlloc`. Anthony also pointed out that it's not clear how to deal with `DLL_THREAD_DETACH` handlers in DLLs, when switching to task granularity.

Locks

There's another aspect of C++11 concurrency that is tied to threads — locking. The `std::mutex` object is thread aware. It requires that `unlock` is called from the same thread as `lock`. Why should this be a problem?

I haven't mentioned yet that task migration might be necessary in the middle of execution, but that is what most task-based systems do. It's an optimization in the case when you're dealing with blocking tasks.

There are two major blocking scenarios: external and internal. External blocking happens when a task calls an OS function (directly or indirectly) that may block, for instance waiting for I/O. My directory listing program did a lot of that. Internal blocking, which is potentially easier to intercept, happens when tasks are blocked on futures. My program did a lot of that too, when waiting for the results of tasks that were listing subdirectories of the current directory.

A blocked task doesn't use processor resources, but it does occupy a thread. That thread could be reused to run another task. But that requires a clean way of taking a task off a thread and then restoring its state once the call unblocks. Now, if the task takes a lock on a mutex before blocking, it cannot be migrated to another thread. The unlocking wouldn't work from another thread.

Herb Sutter observed that, if we tried to restore the task to its original thread, we might get into a spurious deadlock, when the original thread is occupied by another task waiting for the same mutex.

The other problem with locks is in the use of a `recursive_mutex`. A thread may lock such a mutex multiple times before calling `unlock` (also multiple times). But if a second thread tries to lock a mutex that's owned by the current thread, it will block. As long as tasks run in separate threads, this works. But if they share the same thread, they may successfully acquire the same mutex and cause data corruption.

Imagine the following scenario. Task A wants to modify a shared data structure and takes a lock on its recursive mutex. It then blocks on some OS call (probably not a good idea in general, but it may happen). The task gets taken off of the current thread, and task B starts executing. It takes a lock on the same mutex — successfully, as it is executing in the same thread, and reads or modifies a data structure that was in the middle of being modified by task A. A disaster unfolds.

Notice that this is not a problem if tasks are run serially in the same thread, as it happens with the `launch::deferred` policy, because each task runs to completion before allowing another task to run.

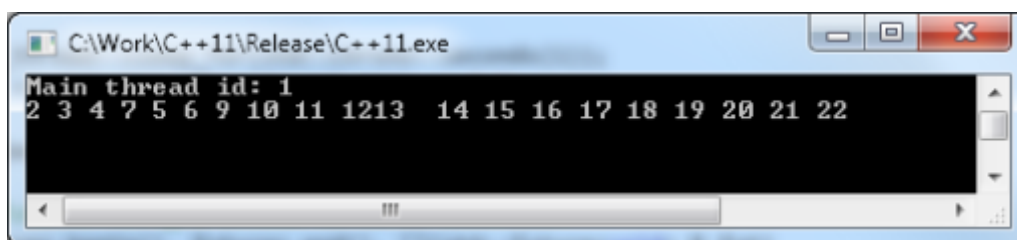
Finally, such migration of running tasks would also wreaks havoc with thread-local variables.

Possible Solutions

Optimizing the Default Launch Policy

The easiest part was to change the implementation of the default policy, to defer the decision whether to run a given task asynchronously or as deferred. Anthony was quick to notice this, and almost immediately released a fix — version 1.7 of `Just::Thread`.

The idea is simple, you schedule N tasks asynchronously — N being some runtime number dependent on the number of available cores — and put the rest on a queue. When any of the queued tasks is forced (by the call to `get` or `wait` on its future), it is executed synchronously in the context of the forcing thread — as if the `launch::deferred` policy were used. Otherwise, as soon as one of the asynchronous tasks finishes, the next task from the queue is scheduled to run asynchronously. Here's the output of the same test program after the changes in `Just::Thread`:



The output of the test with the new library

This time each task ran in a separate thread, but because of the default launch policy, they ran in small batches that could effectively exploit the parallelism of a multicore machine. Still, without thread reuse, the runtime had to create 22 OS threads. The hope is that the operating system caches thread resources so that the creation of the second batch of threads is substantially cheaper than the first one.

(I suggest running this test when evaluating any implementation of a task library.)

Thread Reuse

The next step in improving task performance would be to use a thread pool and reuse threads instead of creating them from scratch. Because of the problem with thread-local variables, it might be impossible to implement thread reuse without some help from the language runtime. The task library would need hooks into every creation of a `thread_local` variable, so it can destroy them at task exit.

That still leaves the problem of tasks calling APIs like `TlsAlloc` directly. An attractive option (for library writers) would be to ignore the problem — after all the language provides a portable way of dealing with thread-local storage.

Task Migration

We would like to be able to remove a blocked task from a thread in order to run another task on it. This is not easy because of thread-locals and locks.

The problem with `thread_local` variables is that they should really be task-local. Or at least they should behave “as if” they were task-local. So when two tasks are sharing the same thread, there has to be some mechanism for “context switching” between them. The context would have to include the state of all thread-local variables.

Migrating a task that is holding a lock could only be done if locks were task-bound rather than thread-bound. Interestingly, there is a provision in the Standard for this kind of behavior. The definition of Lockable in (30.2.5) talks about “execution agents” that could be threads, but could also be something else. This comment is of particular interest:

[Note: Implementations or users may introduce other kinds of agents such as processes or thread-pool tasks. —end note]

However, the Standard Library `mutex` is bound to threads, not tasks. The intention of (30.2.5) is that, if you create your own separate task library with your own task-local variables and mutexes, you will still be able to use the standard utilities such as `lock_guard` or condition variables. But the implementation of `std::async` tasks must work with `thread_local` and `std::mutex`.

Deadlocks

Here’s a potential scenario where two tasks could deadlock if their threads are reused while they are blocked:

1. Task A runs on thread T1, takes the mutex M1, and makes a blocking call
2. The runtime takes A off T1 (saves its state, etc.) and puts it in a Blocked queue
3. Task B starts executing on the same thread, T1, and tries to take M1, which is locked by A

4. In order to unlock M1, task A would have to run on T1 — the same thread the lock was taken on — but T1 is now occupied by B, and A can't make progress

The only way to resolve this deadlock is to take B off the thread. So that's what a task migration system must do — guarantee that any blocked task is taken off its thread.

In general, any spurious deadlock would involve a bunch of blocked tasks. If all of them are blocked on locks, this is an actual deadlock which would happen anyway. If there is at least one task that can make progress when its blocking call returns, it can always be assigned back to its thread, either because the task running on it completes, or because it's blocked and will be taken off of it.

Of course if we allow lock migration, as in associating locks with tasks rather than threads, the problem disappears on its own.

Conclusion

What I learned from this exercise was that `std::async` with default launch policy can be made usable. However its strong association with threads makes it virtually impossible to implement full-blown task-based parallelism. A task-based system could be implemented as a library but it would have to come with severe restrictions on the use of `thread_local` variables and standard mutexes. Such a system would have to implement its own version of task-local variables and mutexes.

I'm grateful to Anthony Williams, Hans Boehm, Herb Sutter, and Artur Laksberg for enlightening discussions.

Appendix: `async` Refresher

Here's some typical code that uses `std::async` to start a task:

```

auto ftr = std::async([](bool flag)->bool
{
    if (flag)
        throw std::exception("Hi!");
    return flag;
}, true); // <- pass true to lambda
// do some work in parallel...
try
{
    bool flag = ftr.get(); // may re-throw exception
}
catch(std::exception & e)
{
    std::cout << e.what() << std::endl;
}

```

The code calls `std::async` with a lambda (anonymous function) that takes a Boolean flag and returns a Boolean. The lambda can either throw an exception or return the flag back. The second argument to `async` (`true`, in this case) is passed to the lambda when it is executed.

The value or the exception is passed back to the parent code when it calls the `get` method on the future returned by `async`. The call to `async` may create a new thread, or defer the execution of the function until the call to `get` is made.

The same code may be implemented directly using `std::thread`, `std::promise`, and `std::future` but, among other things, it requires modifications to the thread function (here, to the lambda):

```

std::promise prms;
auto th = std::thread([](std::promise<bool> & prms, bool flag)
{
    if (flag)
        prms.set_exception(std::make_exception_ptr(std::exception("Hi!")));
    else
        prms.set_value(flag);
}, std::ref(prms), true);
// do some work
th.join();
auto ftr = prms.get_future();
try
{
    bool flag = ftr.get();
}
catch(std::exception & e)
{
    std::cout << e.what() << std::endl;
}

```


13 Responses to “Async Tasks in C++11: Not Quite There Yet”

1. [Async Tasks in C++11: Not Quite There Yet](#) | [Real Coding!!](#) Says:

October 10, 2011 at 3:32 pm

[...] Read the whole article [...]

2. [The Future of C++ Concurrency and Parallelism](#) « [Bartosz Milewski's Programming Cafe](#) Says:

May 11, 2012 at 12:48 pm

[...] point I had a long email exchange about it with Anthony Williams and Hans Boehm that resulted in a blog post. I thought the things were settled until I was alerted to the fact that Microsoft's [...]

3. [Tom](#) Says:



October 10, 2012 at 1:42 pm

Doesn't libdispatch solve most of the problems listed here? I think it is quite an elegant library. Pity that Windows support is not quite there yet.

4. [\(Not\) using std::thread](#) | [Andrzej's C++ blog](#) Says:

November 14, 2012 at 3:38 pm

[...] You may be surprised to find out that f1 and f2 are not executed in parallel: the execution of f2 will not start until f1 is finished! See this paper by Herb Sutter for explanation. Also, Bartosz Milewski describes some invalid expectations of async in his article Async Tasks in C++11: Not Quite There Yet. [...]

5. [lurscher](#) Says:



December 8, 2012 at 10:05 am

[...]The problem with thread_local variables is that they should really be task-local. Or at least they should behave “as if” they were task-local. So when two tasks are sharing the same thread, there has to be some mechanism for “context switching” between them. The context would have to include the state of all thread-local variables.[...]

Have you seen http://www.boost.org/doc/libs/1_52_0/libs/context/doc/html/index.html? user-space context swapping in the same thread. I think there is another boost vault library called fiber in the works that tries to address these shortcomings, basically giving a user-space yield mechanism

6. [Bartosz Milewski](#) Says:



December 8, 2012 at 5:24 pm

Ultimately this problem has to be solved at the language level, since thread local storage is now supported by the language. The Standards Committee is well aware of that.

7. [Doug Gale](#) Says:



August 4, 2013 at 2:50 pm

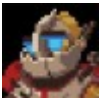
On Windows, `std::async` should be a thin wrapper around `QueueUserWorkItem`. TLS is a complete non-issue because the caller of non-deferred `async` expects/insists that the work runs in another thread (and that it is unpredictable which thread). Completion can be handled by (atomically) lazily initializing an event to wait for (from a pool of event handles), or the event and wait bypassed with atomic operations (if the work completes before anyone waits). I've implemented such a thing and it was easy and had ultra-low overhead. Windows has a great thread pool and it's a shame that C++11 uses it so poorly.

8. [Edward C++Hands](#) | [Bartosz Milewski's Programming Cafe](#) Says:

September 19, 2013 at 12:41 pm

[...] was also a misguided and confusing attempt at providing support for task-based parallelism with `async` tasks and non-composable futures (both seriously considered for deprecation in C++14). Thread-local [...]

9. Robbie Says:



August 26, 2014 at 9:04 pm

This might be a naïve question, but wouldn't a pointer to some context in the heap be enough to emulate Task Local variables? We know when a task has finished executing and it would be reasonable to assume that the end user is responsible for cleaning up the result of a future.

I've recently ran into issues with deadlock in fixed size threadpools. I've been playing around with a Conservative Cached Thread Pool, where when all the tasks cannot make progress a new worker thread is created. Creating a heuristic of when to create or destroy a thread is taking some thought though.

10. hej Says:



December 2, 2014 at 1:17 pm

"What was even worse, when the program didn't fail, it performed better with `launch::deferred` policy, which forces all tasks to be executed serially, than with the `launch::async` policy. That's because thread creation in Windows is so expensive that it can easily nullify performance gains of parallelism..."

Hardly, I guess it's more fun to bash windows but that obviously has to do with your drive not responding well to multitasking. This shouldn't come as a surprise, and in the real world you would rather spend much energy serializing your accesses rather than trying to access it in parallel. Yes, I get that it's just there for illustrative purposes but regardless it really takes away much from the, otherwise interesting, article that such a blatantly obvious result was misinterpreted.

11. [What Limitation of std::async is Stroustrup Referring To?](#) | [Solutions for enthusiast and professional programmers](#) Says:

December 30, 2014 at 8:23 am

[...] was a lot of discussion how powerful this feature should be. In particular, I found the article Async Tasks in C++11: Not Quite There Yet by Bartosz Milewski a very good summary of the problems that have to be considered when implementing [...]

12. [Jeniffer Lenirs](#) Says:



September 19, 2016 at 10:40 am

I saw a lot of discussion about this post, it helps me to get all of my answers. C programmers are everywhere, they should use it. It's very helpful post for every programmer. Thanks for sharing. 😊



13. Arne Vogel Says:

June 12, 2018 at 4:55 am

@hej “Hardly, I guess it’s more fun to bash windows” – Oh no, sorry to challenge your reverence for MS, but according to publicly available benchmark results, it took about 10ms to create a thread on Windows, whereas it only took 10µs on Linux. That is a 1000x (in words, *one thousand times*) difference. Pointing out the brokenness of certain Windows features has nothing to do with bashing, it is simple, time-tested, fact-based criticism. (The machines are not the same, but how much do you think performance increased in 3 years?)

<https://stackoverflow.com/questions/13125105/why-so-much-difference-in-performance-between-thread-and-task>

<https://stackoverflow.com/questions/3929774/how-much-overhead-is-there-when-creating-a-thread#27764581>

BTW, there exist user-space solutions that make even Linux appear sluggish, e.g. HPX thread creation averaged at about 250ns in this benchmark, or 40x faster than Linux native (40,000x faster than Windows native): <http://salishan.ahsc-nm.org/uploads/4/9/7/0/49704495/2016-sterling.pdf>

TL;DR: Thread creation on Windows is not just slow, it is *ridiculously* slow.

What actually brought me here was however to point out that `std::async` is (IMO) even less useful than explained here, because `libstdc++`’s implementation does not *ever* destroy the task before the associated future is waited on. And the lib is not even wrong in doing so because the standard does not specify the task’s lifetime (could theoretically be unlimited?).

<https://stackoverflow.com/questions/47202990/how-does-stdfuture-affects-the-lifetime-of-an-associated-stdpackaged-task/47219655#47219655>

So the task is only destroyed upon `future.get()`. But it gets even worse than that! If you do a `future.share()`, the task lives until the last `shared_future` is destroyed. Which means that using a task owning any resources, and proper RAII, is more than contraindicated, for `libstdc++` at least.

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

[Blog at WordPress.com.](#)