Vorbrodt's C++ Blog

PRACTICAL MODERN C++

Advanced thread pool

Below is my implementation of the thread pool described in this talk and a benchmark comparing it against my simple thread pool implementation. The advanced pool is 15x faster at scheduling and dispatching short random length work items on my 2018 MacBook Pro with i5 CPU and 4 logical cores. It uses a queue per worker thread and a work stealing dispatcher. It tries to enqueue the work items onto a queue that is not currently locked by a dispatch thread. It also tries to steal work from other unblocked queues. As always the complete implementation is available at GitHub.

Benchmark program:

```cpp
#include <iostream>
#include <chrono>
#include <cstdlib>
#include "pool.h"
using namespace std;
using namespace chrono;

const unsigned int COUNT = 10'000'000;
const unsigned int REPS = 10;

int main()
{
    srand(0);
    auto start = high_resolution_clock::now();
    {
        simple_thread_pool tp;
        for(int i = 0; i < COUNT; ++i)
            tp.enqueue_work([i]() {
                int x;
                int reps = REPS + (REPS * (rand() % 5));
                for(int n = 0; n < reps; ++n)
                    x = i + rand();
            });
    }
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);
    cout << "simple_thread_pool duration = " << duration.count() / 1000.f << " s" << endl;

    srand(0);
    start = high_resolution_clock::now();
    {
        thread_pool tp;
        for(int i = 0; i < COUNT; ++i)
            tp.enqueue_work([i]() {
                int x;
                int reps = REPS + (REPS * (rand() % 5));
                for(int n = 0; n < reps; ++n)
                    x = i + rand();
            });
    }
    end = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(end - start);
    cout << "thread_pool duration = " << duration.count() / 1000.f << " s" << endl;
}
```

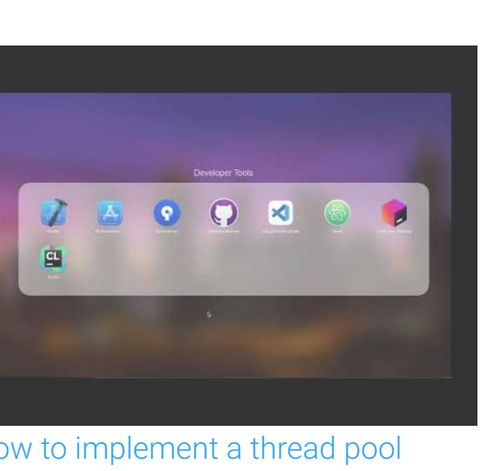
> * Apple CLANG -Ofast -march=native -std=c++17 -lc++
>
> simple_thread_pool duration = 30.337 s
> thread_pool duration = 1.625 s
>
> * LLVM -Ofast -march=native -std=c++17 -lc++
>
> simple_thread_pool duration = 25.785 s
> thread_pool duration = 1.615 s
>
> * G++ -Ofast -march=native -std=c++17 -lstdc++ *
>
> simple_thread_pool duration = 26.28 s
> thread_pool duration = 1.614 s
>
> *PROGRAM OUTPUT.*

thread_pool class:

```cpp
class thread_pool
{
public:
    thread_pool(unsigned int threads = std::thread::hardware_concurrency())
    : m_queues(threads), m_count(threads)
    {
        assert(threads != 0);
        auto worker = [&](unsigned int i)
        {
            while(true)
            {
                Proc f;
                for(unsigned int n = 0; n < m_count; ++n)
                    if(m_queues[(i + n) % m_count].try_pop(f)) break;
                if(!f && !m_queues[i].pop(f)) break;
                f();
            }
        };
        for(unsigned int i = 0; i < threads; ++i)
            m_threads.emplace_back(worker, i);
    }

    ~thread_pool() noexcept
    {
        for(auto& queue : m_queues)
            queue.done();
        for(auto& thread : m_threads)
            thread.join();
    }

    template<typename F, typename... Args>
    void enqueue_work(F&& f, Args&&... args)
    {
        auto work = [f,args...]() { f(args...); };
        unsigned int i = m_index++;
        for(unsigned int n = 0; n < m_count * K; ++n)
            if(m_queues[(i + n) % m_count].try_push(work)) return;
        m_queues[i % m_count].push(work);
    }

    template<typename F, typename... Args>
    auto enqueue_task(F&& f, Args&&... args) -> std::future<typename std::result_of<F(Args...)>::type>
    {
        using return_type = typename std::result_of<F(Args...)>::type;
        auto task = std::make_shared<std::packaged_task<return_type()>>(std::bind(std::forward<F>(
        std::future<return_type> res = task->get_future();

        auto work = [task](){ (*task)(); };
        unsigned int i = m_index++;
        for(unsigned int n = 0; n < m_count * K; ++n)
            if(m_queues[(i + n) % m_count].try_push(work)) return res;
        m_queues[i % m_count].push(work);

        return res;
    }

private:
    using Proc = std::function<void(void)>;
    using Queues = std::vector<simple_blocking_queue<Proc>>;
    Queues m_queues;

    using Threads = std::vector<std::thread>;
    Threads m_threads;

    const unsigned int m_count;
    std::atomic_uint m_index = 0;

    inline static const unsigned int K = 3;
};
```


Related

Simple thread pool
I know the topic of thread pools has been beaten to death on the internet, nevertheless I wanted to present to you my implementation which uses only standard C++ components :)
February 12, 2019
In "Martin"



How to implement a thread pool
April 15, 2021
In "Martin"

Better timer class
It bothered me that my previous simple timer implementation fired off a new thread for each timeout and interval. I knew things could be done better, but didn't yet know how. Well
February 25, 2019
In "Martin"

Posted on February 27, 2019 by Martin

BETTER CODE: CONCURRENCY PARALLEL STL

14 Replies to "Advanced thread pool"

Pingback: Simple thread pool – Vorbrodt's C++ Blog

Oliver Schönrock says:
December 1, 2019 at 10:55 pm

Thanks for writing this. Just built your pool from git:

both are the same speed for me?

simple_thread_pool 100 1 38.2594 s
379.447 ms 377.661 ms 382.583 ms
11.8086 ms 7.64767 ms 17.8581 ms

thread_pool 100 1 38.7423 s
387.369 ms 386.488 ms 387.932 ms
3.53217 ms 2.48396 ms 4.95731 ms

That's just running you built in tests.

Is is that the "work" is too predictable to benefit from "task stealing etc".

Loading...

Reply

Martin Vorbrodt says:
December 2, 2019 at 12:43 am

Are those release builds or debug? As the tasks get longer the benefit of the better pool will diminish btw…

Loading...

Reply

1. **Oliver Schönrock** says:
December 2, 2019 at 6:00 am

I checked with verbose and it was all compiled with -O3. Lots of ransom tiny tasks are tricky to schedule that's true. So I played with REPS etc but couldn't create a difference.

I had to add a PREFER PTHREADS to the cmake list to make it compile on my ubuntu 19.04 system.

Btw why is there no "official" implementation for this sort of thing? Something in the STL or at least Boost or some other well maintained library.

std:async is totally broken on linux at least. It runs with "deferred" (which is not parallel at all) or launches one thread per task and blows up. So a thread pool is totally needed. Am I missing something?

Lots of talk about "better futures" (eg the Sean Parent futures library, or the new stuff in c++20), but where the "standard thread pool implementation"?

Loading...

Reply

2. **Oliver Schönrock** says:
December 2, 2019 at 6:21 am

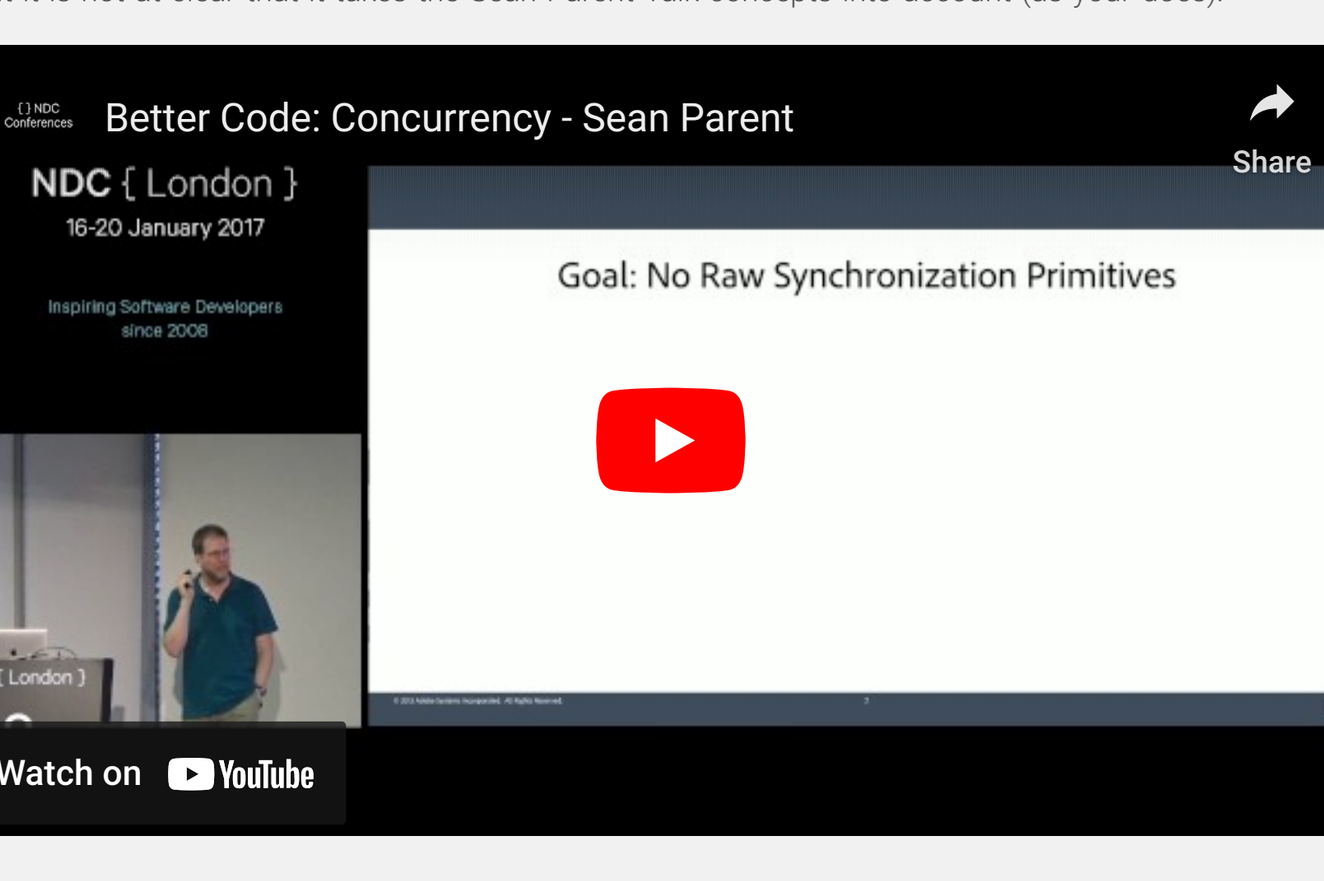
This guy compared a bunch of random github projects:

https://github.com/Fdhvdu/ThreadPool/tree/master/comparison

And This one won (his own one). And it's modern cod, etc

https://github.com/Fdhvdu/ThreadPool

But it is not at clear that it takes the Sean Parent Talk concepts into account (as your does):



https://github.com/Fdhvdu/ThreadPool/blob/master/README.md
"Future work

add a non-block version of CThreadPool::add
work stealing"

???

It's not that (!) complicated, but no "formal" solutions? Everyone hacking around on their own?

Loading...

Reply

Sedit T says:
January 28, 2020 at 8:05 pm

Nice TP,
Funny, I have never seen this but ended up producing a very similar pool likely due to the fact we both has inspiration from Parents talk. Only difference is the way I chose to handle the Task. I make a task that stores the arguments in a Tuple and uses a polymorphic Invoke so that I could store pointers to the base class allowing me to store functions with varying arguments. I found it to perform better than the solution similar to the one above which I tried first.

Still in construction because I am working on making it a parent stealing queue. The one you have above fails when you attempt to make recursive calls to the TP. Try a Merge sort to see what I mean.

https://github.com/xSeditx/Creature-Engine/blob/master/CreatureEngine/Core/Threading/Threadpool.h
https://github.com/xSeditx/Creature-Engine/blob/master/CreatureEngine/Core/Threading/Threadpool.cpp

Loading...

Reply

Sedit T says:
January 28, 2020 at 8:06 pm

Funny, I have never seen this but ended up producing a very similar pool likely due to the fact we both has inspiration from Parents talk. Only difference is the way I chose to handle the Task. I make a task that stores the arguments in a Tuple and uses a polymorphic Invoke so that I could store pointers to the base class allowing me to store functions with varying arguments. I found it to perform better than the solution similar to the one above which I tried first.

Still in construction because I am working on making it a parent stealing queue. The one you have above fails when you attempt to make recursive calls to the TP. Try a Merge sort to see what I mean.

https://github.com/xSeditx/Creature-Engine/blob/master/CreatureEngine/Core/Threading/Threadpool.h
https://github.com/xSeditx/Creature-Engine/blob/master/CreatureEngine/Core/Threading/Threadpool.cpp

Loading...

Reply

Howard Rosenorn says:
March 5, 2020 at 10:36 pm

I didn't see much difference between the two thread pools. Although it was a fun way to kill an evening. 🙂

AMD Phenom(tm) II X4 965 Processor
4 Cores
Arch Linux

simple_thread_pool duration = 35.443 s
thread_pool duration = 32.128 s

simple (100,000 tasks, 100 reps) 1164.47 ms (10,000,000)
advanced (100,000 tasks, 100 reps) 823.953 ms (10,000,000)

simple (100,000 tasks, 200 reps) 2015.13 ms (20,000,000)
advanced (100,000 tasks, 200 reps) 2469.78 ms (20,000,000)

simple (100,000 tasks, 300 reps) 3775.12 ms (30,000,000)
advanced (100,000 tasks, 300 reps) 3748.74 ms (30,000,000)

simple (100,000 tasks, 400 reps) 4824.86 ms (40,000,000)
advanced (100,000 tasks, 400 reps) 4554.73 ms (40,000,000)

simple (100,000 tasks, 500 reps) 5721.67 ms (50,000,000)
advanced (100,000 tasks, 500 reps) 5684.69 ms (50,000,000)

simple (100,000 tasks, 600 reps) 6955.63 ms (60,000,000)
advanced (100,000 tasks, 600 reps) 6838.74 ms (60,000,000)

simple (100,000 tasks, 700 reps) 7943.49 ms (70,000,000)
advanced (100,000 tasks, 700 reps) 7716.07 ms (70,000,000)

simple (100,000 tasks, 800 reps) 8803.46 ms (80,000,000)
advanced (100,000 tasks, 800 reps) 9205.39 ms (80,000,000)

simple (100,000 tasks, 900 reps) 10698.1 ms (90,000,000)
advanced (100,000 tasks, 900 reps) 10614.8 ms (90,000,000)

simple (100,000 tasks, 1,000 reps) 11327.7 ms (100,000,000)
advanced (100,000 tasks, 1,000 reps) 11496 ms (100,000,000)

I ran the test longer then this but the results were pretty consistant.

Loading...

Reply

Martin Vorbrodt says:
March 5, 2020 at 10:39 pm

You are correct in the sense that as the duration of each individual task goes up the performance difference of the pools comes closer. Try tasks with 1 to 10 reps instead, you may find bigger performance gains from the advanced one.

Loading...

Reply

Hadi says:
August 30, 2020 at 7:05 pm

Hello Martin,
I have difficulty to enqueue the jobs via my function (which has an smart pointer as argument). Would u please check my post:
https://stackoverflow.com/questions/63662155/problem-in-calling-enqueue-work-vorbrodt-thread-pool

Loading...

Reply

Martin Vorbrodt says:
August 31, 2020 at 12:09 pm

I posted a reply to your stack overflow question…

Loading...

Reply

Anonymous says:
September 7, 2020 at 7:17 am

Hi Martin,

why do you prefer the following signature as opposed to a simpler signature not including f's arguments (they can be bound to the 'work' function externally)? This signature suggests perfect-forwarding, but I don't think that really happens internally. Am I right?

template
void enqueue_work(F&& f, Args&&… args)
{
auto work = f,args… { f(args…); };
unsigned int i = m_index++;
for(unsigned int n = 0; n < m_count * K; ++n)
if(m_queues[(i + n) % m_count].try_push(work)) return;
m_queues[i % m_count].push(work);
}

Loading...

Reply

Claudio says:
September 7, 2020 at 7:18 am

Hi Martin, (re-posting as non-anonymous)

why do you prefer the following signature as opposed to a simpler signature not including f's arguments (they can be bound to the 'work' function externally)? This signature suggests perfect-forwarding, but I don't think that really happens internally. Am I right?

template
void enqueue_work(F&& f, Args&&… args)
{
auto work = f,args… { f(args…); };
unsigned int i = m_index++;
for(unsigned int n = 0; n < m_count * K; ++n)
if(m_queues[(i + n) % m_count].try_push(work)) return;
m_queues[i % m_count].push(work);
}

Loading...

Reply

Martin Vorbrodt says:
September 12, 2020 at 7:03 am

I prefer this signature because it allows BOTH: binding arguments to a function in place and externally. the proc and args are forwarded correctly (moves) but then a copy has to be made when they are put on a queue unfortunately. I could solve it with shared pointers I suppose…

Loading...

Reply

Leave a Reply

Enter your comment here...

ME, ELSEWHERE

MY CORNER

Archives
About Me

KOBI'S CORNER

San Diego C++ Meetup
Kobi's Posts
About Kobi

C++ ONLINE

Top 50 C++ Blogs
- Awesome C++
- Belay the C++
- CppCast
- C++ Core Guidelines
- C++ Reference
- C++ Stories
- C++ Team Blog
- Fluent { C++ }
- foonathan::blog()
- ISO C++ Blog
- Meeting C++
- Modernes C++
- Sutter's Mill

TOP POSTS

Function alias vs Function pointer alias
Blocking queue
Memory barriers and thread synchronization
Protocol Buffers: or how to serialize data
SSO of std::string

SUBSCRIBE VIA EMAIL

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Email Address

Subscribe

Join 2,119 other subscribers

ADMIN

Log in
Entries feed
Comments feed
WordPress.org

STATS

226,228 views

top