ThreadPool Documentation

Release 0.1

Loic Reyreaud

Nov 26, 2018

Contents:

ThreadPool is a modern C++ header only library. It features perfect tasks forwarding and return value retrieval through futures and start/top mechanism.

You can find all code examples in the repository under `doc/src/examples`.

CHAPTER 1

Getting Started

1.1 Install

First, you need the `threadpool.hpp` header file. You can simply copy it into your source tree, or you can install it. To install run the following commands:

```
$ git clone git@github.com:reyreaud-l/threadpool.git
$ cd threadpool
$ mkdir build
$ cd build
$ cmake ..
$ make install
```

This will make the header available in your system.

1.2 First Pool

Let's see a simple use of the threadpool

```
#include "threadpool.hpp"

#include <iostream>

int main()
{
  ThreadPool::ThreadPool mypool(1);
  auto task = mypool.run([]() { std::cout << "Hello there!" << std::endl; });
  std::cout << "General Kenobi!" << std::endl;
  task.wait();
}
```

Note that we instantiate a task and wait for it to make sure the task is ran. With such a short program, it is possible that mypool will be deleted before a worker is woke up to pick the task and run it. Waiting for the result ensure the task is ran. It is possible to use this syntax if you don't care about the return value or don't want to wait for a task to end:

```
...
mypool.run([]() { std::cout << "Hello there!" << std::endl; });
...
```

This concludes a very basic example to set up the threadpool!

CHAPTER 2

Pool Control

2.1 Pool size

You can control the threadpool size when instantiating it with the constructors. Here are the different constructors you can use:

ThreadPool::ThreadPool::**ThreadPool**(std::size_t *pool_size*)
Constructs a ThreadPool.

Parameters

- `pool_size`: Number of threads to start.

ThreadPool::ThreadPool::**ThreadPool**(std::size_t *pool_size*, std::shared_ptr<Hooks> *hooks*)
Constructs a ThreadPool.

Parameters

- `pool_size`: Number of threads to start.
- `hooks`: Hooks to register in the pool.

The `pool_size` parameter will determine the number of workers(threads) the pool will start when instantiating.

2.2 Stopping the pool

You can stop the pool with:

void *ThreadPool*::*ThreadPool*::**stop**()
Stop the ThreadPool.

A stopped ThreadPool will discard any task dispatched to it. All workers will discard new tasks, but the threads will not exit.

You can check if the pool is stopped with:

bool *ThreadPool*::*ThreadPool*::**is_stopped**() **const**
Check the state of the threadpool.

Return True if the ThreadPool is stopped, false otherwise.

2.3 Checking the pool

You can check the current state of the workers with:

std::size_t *ThreadPool*::*ThreadPool*::**threads_available**() **const**
Check on the number of threads not currently working.

The number might be imprecise, as between the time the value is read and returned, a thread might become unavailable.

Return The number of threads currently waiting for a task.

std::size_t *ThreadPool*::*ThreadPool*::**threads_working**() **const**
Check on the number of threads currently working.

The number might be imprecise, as between the time the value is read and returned, a thread might finish a task and become available.

Return The number of threads currently working.

CHAPTER 3

Pool Performance

The pool performance can be measured using benchmarks.

Benchmark with neighbors work stealing enabled:

```
Running ./benchbin
Run on (4 X 3500 MHz CPU s)
CPU Caches:
  L1 Data 32K (x4)
  L1 Instruction 32K (x4)
  L2 Unified 256K (x4)
  L3 Unified 6144K (x1)
------------------------------------------------------------------
Benchmark                        Time           CPU Iterations
------------------------------------------------------------------
bm_work_tasks/8                  0 ms          0 ms      12826
bm_work_tasks/64                 0 ms          0 ms       7478
bm_work_tasks/512                1 ms          1 ms       1216
bm_work_tasks/4096               5 ms          4 ms        157
bm_work_tasks/32768             36 ms         36 ms         19
bm_work_tasks/262144           302 ms        301 ms          2
bm_work_tasks/2097152         2370 ms       2346 ms          1
bm_blocking_tasks/8              0 ms          0 ms      10341
bm_blocking_tasks/64             1 ms          0 ms       5379
bm_blocking_tasks/512            8 ms          1 ms        600
bm_blocking_tasks/4096          66 ms          4 ms        100
bm_blocking_tasks/32768        525 ms         30 ms         10
bm_blocking_tasks/262144      4160 ms        250 ms          1
```

Benchmark with work stealing across all workers enabled:

```
Running ./benchbin
Run on (4 X 3500 MHz CPU s)
CPU Caches:
  L1 Data 32K (x4)
  L1 Instruction 32K (x4)
```

(continues on next page)

(continued from previous page)

```
  L2 Unified 256K (x4)
  L3 Unified 6144K (x1)
------------------------------------------------------------------
Benchmark                        Time           CPU Iterations
------------------------------------------------------------------
bm_work_tasks/8                   0 ms          0 ms       12669
bm_work_tasks/64                  0 ms          0 ms        7430
bm_work_tasks/512                 1 ms          1 ms        1146
bm_work_tasks/4096                5 ms          5 ms         153
bm_work_tasks/32768              37 ms         36 ms          19
bm_work_tasks/262144            311 ms        308 ms           2
bm_work_tasks/2097152          2459 ms       2443 ms           1
bm_blocking_tasks/8               0 ms          0 ms       11270
bm_blocking_tasks/64              1 ms          0 ms        5836
bm_blocking_tasks/512             8 ms          1 ms         600
bm_blocking_tasks/4096           65 ms          4 ms         100
bm_blocking_tasks/32768         516 ms         30 ms          10
bm_blocking_tasks/262144       4125 ms        244 ms           1
```

Benchmark with work no work stealing:

```
Running ./benchbin
Run on (4 X 3500 MHz CPU s)
CPU Caches:
  L1 Data 32K (x4)
  L1 Instruction 32K (x4)
  L2 Unified 256K (x4)
  L3 Unified 6144K (x1)
------------------------------------------------------------------
Benchmark                        Time           CPU Iterations
------------------------------------------------------------------
bm_work_tasks/8                   0 ms          0 ms       12989
bm_work_tasks/64                  0 ms          0 ms        6765
bm_work_tasks/512                 1 ms          1 ms        1181
bm_work_tasks/4096                5 ms          5 ms         157
bm_work_tasks/32768              36 ms         36 ms          19
bm_work_tasks/262144            307 ms        305 ms           2
bm_work_tasks/2097152          2407 ms       2393 ms           1
bm_blocking_tasks/8               0 ms          0 ms        8766
bm_blocking_tasks/64              1 ms          0 ms        4970
bm_blocking_tasks/512             9 ms          1 ms         600
bm_blocking_tasks/4096           67 ms          4 ms         100
bm_blocking_tasks/32768         529 ms         30 ms          10
bm_blocking_tasks/262144       4224 ms        242 ms           1
```

CHAPTER 4

Pool Hooks

4.1 Writing hooks

Hooks are written by creating a class or a struct which inherits from the class `ThreadPool::Hooks`. You can then override whichever hooks you need.

4.1.1 Tasks Hooks

virtual void `ThreadPool::Hooks::`**pre_task_hook**`()`
Hook called before picking a task.

This hook will be called by a worker before a task is executed. The worker will not have anything locked when calling the hook. The worker will call in a "working" state. That means that if the hook takes too long, the worker will hold on the task execution and not run it.

virtual void `ThreadPool::Hooks::`**post_task_hook**`()`
Hook called after a task is done.

This hook will be called by a worker after a task is done. The worker will not have anything locked when calling the hook. The worker will call in a "working" state. That means that if the hook takes too long, the worker will hold and not pick a task until the hook is completed.

4.1.2 Workers Hooks

virtual void `ThreadPool::Hooks::`**on_worker_add**`()`
Hook called when a worker is added for a single task.

This hook will be called by the main thread (the thread making the call to run). It is called only when the ThreadPool automatically scales to add one more worker. The initials workers created by the ThreadPool will not notify this hook.

virtual void `ThreadPool::Hooks::`**on_worker_die**`()`
Hook called when a worker dies.

This hook will be called by the thread the ThreadPool is detroyed with. All workers will notify this hook.

4.2 Registering hooks

Hooks can be used to be notified when actions happens in the pool. The hooks are registered in the pool using the function:

void *ThreadPool*::*ThreadPool*::**register_hooks**(std::shared_ptr<Hooks> *hooks*)
Register a ThreadPool::Hooks class.

Parameters

- `hooks`: The class to be registered

4.3 Example

Here is a simple example to use hooks with the Pool.

```
#include <iostream>
#include <memory>

#include "threadpool.hpp"

struct TestHooks : public ThreadPool::Hooks
{
  void pre_task_hook() final
  {
    std::cout << "pre_task_hook\n";
  }

  void post_task_hook() final
  {
    std::cout << "post_task_hook\n";
  }
};

int main(void)
{
  ThreadPool::ThreadPool pool(1);
  std::shared_ptr<TestHooks> hooks(new TestHooks());
  pool.register_hooks(hooks);

  auto res = pool.run([]() { return 0; });
  res.wait();
}
```

CHAPTER 5

ThreadPool Doxygen

5.1 Public ThreadPool

class ThreadPool

ThreadPool implement a multiple queues/multiple workers threadpool.

When created, the pool will start the workers(threads) immediatly. The threads will only terminate when the pool is destroyed.

This class implements a one queue per worker strategy to dispatch work.

Public Functions

ThreadPool()

Constructs a ThreadPool.

The number of workers will be deduced from hardware.

ThreadPool(std::size_t *pool_size*)

Constructs a ThreadPool.

Parameters

- `pool_size`: Number of threads to start.

ThreadPool(std::shared_ptr<Hooks> *hooks*)

Constructs a ThreadPool.

Parameters

- `hooks`: Hooks to register in the pool.

ThreadPool(std::size_t *pool_size*, std::shared_ptr<Hooks> *hooks*)

Constructs a ThreadPool.

Parameters

- `pool_size`: Number of threads to start.
- `hooks`: Hooks to register in the pool.

~ThreadPool()
Stops the pool and clean all workers.

template <typename Function, typename... *Args*>
auto **run**(Function &&*f*, Args&&... *args*)
Run a task in the SingleQueue.

When a task is ran in the SingleQueue, the callable object will be packaged in a packaged_task and put in the inner task_queue. A waiting worker will pick the task and execute it. If no workers are available, the task will remain in the queue until a worker picks it up.

Return Returns a future containing the result of the task.

void **stop**()
Stop the ThreadPool.

A stopped ThreadPool will discard any task dispatched to it. All workers will discard new tasks, but the threads will not exit.

void **register_hooks**(std::shared_ptr<Hooks> *hooks*)
Register a ThreadPool::Hooks class.

Parameters

- `hooks`: The class to be registered

bool **is_stopped**() **const**
Check the state of the threadpool.

Return True if the ThreadPool is stopped, false otherwise.

std::size_t **threads_available**() **const**
Check on the number of threads not currently working.

The number might be imprecise, as between the time the value is read and returned, a thread might become unavailable.

Return The number of threads currently waiting for a task.

std::size_t **threads_working**() **const**
Check on the number of threads currently working.

The number might be imprecise, as between the time the value is read and returned, a thread might finish a task and become available.

Return The number of threads currently working.

5.2 Private ThreadPool

class ThreadPool
ThreadPool implement a multiple queues/multiple workers threadpool.

When created, the pool will start the workers(threads) immediatly. The threads will only terminate when the pool is destroyed.

This class implements a one queue per worker strategy to dispatch work.

Private Functions

void **start_pool**()
Starts the pool when the pool is constructed.

It will starts _pool_size threads.

void **clean**()
Clean the pool and join threads of dead workers.

This method may be called at any time by any thread putting a job in the queue. This function acquires a lock on the pool vector.

void **terminate**()
Joins all threads in the pool.

Should only be called from destructor. This method will stop all the worker and join the corresponding thread.

std::size_t **get_dispatch_worker**()
Find the worker for which to dispatch the tasks.

Return The index in the worker array to which a task should be dispatch.

template <**typename** TaskType>
void **dispatch_work**(**const** std::size_t *idx*, TaskType *task*)
Dispatch a task to a given worker.

Parameters

- `idx`: Index of the worker to dispatch the work at
- `task`: Task to dispatch into the worker

void **check_spawn_single_worker**()
Check if the pool can spawn more workers, and spawn one for a single task.

It will check the current number of spawned threads and if it can spawn or not a new thread. If a thread can be spawned, one is created.

Private Members

std::vector<std::pair<std::thread, std::unique_ptr<*Worker*>>> **pool**
Vector of thread, the actual thread pool.

Emplacing in this vector construct and launch a thread.

std::mutex **pool_lock**
Mutex regulating acces to the pool.

std::atomic<std::size_t> **waiting_threads**
Number of waiting threads in the pool.

std::atomic<std::size_t> **working_threads**
Number of threads executing a task in the pool.

std::atomic<bool> **stopped**
Boolean representing if the pool is stopped.

std::shared_ptr<Hooks> **hooks**
Struct containing all hooks the threadpool will call.

const std::size_t **pool_size**
Size of the pool.

struct Worker
Inner worker class. Capture the ThreadPool when built.

The only job of this class is to run tasks. It will use the captured ThreadPool to interact with it.

Private Members

ThreadPool ***pool**
Captured ThreadPool that the worker works for.

CHAPTER 6

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

6.1 Unreleased

6.2 [3.0.0] - 2018-11-26

6.2.1 Added

- Added overloading constructor which deducts number of available thread from the hardware.
- All code now lies in the `ThreadPool` namespace.
- Add an option to either download dependencies or use system wide install.
- Added work stealing in the ThreadPool to increase performance.
- Add multiple macros to select whether or not you wish to enable work stealing.

6.2.2 Changed

- ThreadPool implementation is now using multiple queues with multiple workers.
- Remove hook copy overload (it was wrong)
- Changed dependency management (gtest/gbench) to be downloaded at compile time and not at configure time.

6.2.3 Removed

- The ThreadPool is now a fixed size pool. I removed the ability to adapt the number of threads at runtime.

6.3 [2.0.0] - 2018-09-21

6.3.1 Added

- Code examples from the doc are now buildable standalone programs.
- Tasks hooks.
- Workers hooks.
- Documentation generation with breath and sphinx.
- Deploy documentation on read the doc.
- Uninstall target in CMake.
- Changelog section in documentation.

6.3.2 Changed

- CI now check linux/osx with multiples compiler.
- Documentation is now hosted on read the doc. The documentation now also includes doxygen using breathe sphinx plugin.

6.4 [1.0.0] - 2018-06-26

6.4.1 Added

- First release (!yay).
- Fixed/Variable pool size.
- Fetch result of task with futures.
- Fetch number of waiting/working workers.

6.5 List of releases

- Unreleased
- 3.0.0
- 2.0.0
- 1.0.0

CHAPTER 7

Indices and tables

- genindex

Index

T