R-Friendly Multi-Threading in C++

Thomas Nagler
Technical University of Munich

Abstract

Calling multi-threaded C++ code from R has its perils. Since the R interpreter is single-threaded, one must not check for user interruptions or print to the R console from multiple threads. One can, however, synchronize with R from the main thread. The R package **RcppThread** (current version 0.5.0) contains a header only C++ library for thread safe communication with R that exploits this fact. It includes C++ classes for threads, a thread pool, and parallel loops that routinely synchronize with R. This article explains the package's functionality and gives examples of its usage. The the synchronization mechanism may also apply to other threading frameworks. Benchmarks suggest that, although synchronization causes overhead, the parallel abstractions of **RcppThread** are competitive with other popular libraries in typical scenarios encountered in statistical computing.

Keywords: R, C++, parallel, thread, concurrency.

1. Introduction

1.1. From single to multi-cores machines

For a long time, computers had only a single CPU and computer programs were a set of instructions that the CPU executed in sequential order. Accordingly, most programming languages that are still popular today (including R and C++) were designed with a single-processor model in mind. Computing power was growing at exponential rates for decades and there was no reason to change anything about that.

A paradigm shift came shortly after the turn of the millennium. Sutter (2005) warned that the "free lunch will soon be over": although the number of transistors on CPUs is continuing to grow exponentially, their clock speed is approaching physical limits. To keep increasing the computing power, manufacturers made a move towards multi-core machines. Today, virtually all desktop PCs, laptops, and even smart phones have multiple cores to deal with increasingly demanding software applications.

As time progresses, statistical tools and methods are becoming increasingly complex and demanding for the computer. For that reason, many R packages implement performance-critical tasks in a lower level language like C and Fortran. Interfacing with C++ has become especially popular in recent years thanks to the excellent **Rcpp** package (Eddelbuettel and François 2011; Eddelbuettel 2013), which is used by almost 1 500 ($\approx 10\%$ of total) packages on CRAN and counting.

1.2. Threads as programming abstraction for multi-core hardware

To get the most out of a modern computer, it is vital to utilize not only one, but many cores concurrently. This can be achieved by allowing multiple *threads* to be executed at the same time. A thread encapsulates a sequence of instructions that can be managed by a task scheduler, typically the operating system. A simple program has only a single thread of execution, that encapsulates all instructions in the program, the *main thread*. However, a thread can also *spawn* new threads that may run concurrently on multiple cores until all work is done and the threads are *joined*.

Many frameworks provide higher-level abstractions for running multi-threaded code in C++. The veteran among them is **OpenMP** (Dagum and Menon 1998). It provides preprocessor directives to mark code sections that run concurrently. More modern frameworks include **boost.thread** (Boost 2018), **Tinythread** (Geelnard 2012), and **Intel TBB** (Pheatt 2008). Since the advent of C++11, the standard library provides a built-in implementation, **std::thread**.

1.3. Calling multi-threaded code from R

Calling multi-threaded C++ code from R can be problematic because the R interpreter is single-threaded. To quote from the 'Writing R Extensions' manual: *Calling any of the R API from threaded code is 'for experts only'.* Using R's API from concurrent threads may crash the R session or cause other unexpected behavior. In particular, communication between C++ code and R is problematic. We can neither check for user interruptions during long computations nor should we print messages to the R console from any other than the main thread. It is possible to resolve this, but not without effort.

1.4. RcppThread and related packages

The R package **RcppThread** aims to relieve package developers of that burden. It contains C++ headers that provide:

- thread safe versions of `Rcpp::Rcout` and `Rcpp::checkUserInterrupt()`,
- parallel abstractions: thread, thread pool, and parallel `for` loops.

Besides the numerous packages for parallelism in R (see https://cran.r-project.org/web/views/HighPerformanceComputing.html), there are two packages that inspired **RcppThread** and provide similar functionality. **RcppProgress** (Forner 2018) allows to safely check for user interruptions when code is parallelized with **OpenMP** (but only then). Further, **RcppParallel** (Allaire, Francois, Ushey, Vandenbrouck, Geelnard, and Intel 2018) is an interface to many high-level parallel abstractions provided by **Intel TBB**, but does not allow for thread safe communication with R.

2. Thread safe communication with R

It is not safe to call R's C API from multiple threads. It is safe, however, to call it from the main thread. That's the idea behind **RcppThread**'s `checkUserInterrupt()` and `Rcout`. They behave almost like their **Rcpp** versions, but only communicate with R when called from the main thread.

2.1. Interrupting computations

It is fairly easy to make `checkUserInterrupt()` thread safe. We first check whether the function is called from the main thread, and only then we ask R whether there was a user interruption.

Consider the following example with `std::thread`:

```
#include <RcppThread.h>
// [[Rcpp::export]]
void sleep()
{
    auto job = [] {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        RcppThread::checkUserInterrupt();
        std::this_thread::sleep_for(std::chrono::seconds(1));
    };
    std::thread t(job);
    t.join();
}
```

The first line includes the **RcppThread** header, which automatically includes the standard library headers required for `std::thread` and `std::chrono`. The second line triggers **Rcpp** to export the function to R . We define a function `sleep()`. In the function body, we declare a function `job()` that sleeps for one second, checks for a user interruption, and then sleeps for another second. We then create an `std::thread` with the new job and join it before the program exits.

If we call the above function from R, the program would complete as expected. But would we have used `Rcpp::checkUserInterrupt()` instead, the program would terminate unexpectedly and crash the R session.

If `RcppThread::checkUserInterrupt()` is called from the main thread and the user signaled an interruption, a `RcppThreadUserException` will be thrown. This translates to an error in R with the message

```
C++ call interrupted by the user.
```

A related function is `RcppThread::isInterrupted()` which does not throw an exception, but returns a boolean signaling the interruption status. This can be useful, if some additional cleanup is necessary or one wants to print diagnostics to the R console.

However, when the functions are called from a child thread, they do not actually check for an interruption. This can be problematic if it is *only* called from child threads. That does not happen with **OpenMP** or **Intel TBB**, but with lower level frameworks like **std::thread**, **TinyThread** or **boost.thread**.

Both functions accept a `bool` that allows to check conditionally on the state of the program. For example, in a loop over `i`, `RcppThread::checkUserInterrupt(i % 20)` will only check in every 20th iteration. Checking for interruptions is quite fast (usually microseconds), but there is a small overhead that can accumulate to something significant in loops with many iterations. Checking conditionally can mitigate this overhead.

There is a hidden detail worth mentioning. The two functions above are not completely useless when called from a child thread. They check for a global variable indicating whether the main thread has noticed an interruption. Hence, as soon the main thread witnesses an interruption, all child threads become aware.

In Section 3.2, we will discuss how to make sure that `isInterrupted()` is called from the main thread every now and then. For now, we are only able to write functions that are interruptable from the main thread, and safe to call from child threads.

2.2. Printing to the R console

A similar issue arises when multiple threads try to print to the R console simultaneously. Consider the following example:

```
#include <thread>
#include <Rcpp.h>
// [[Rcpp::export]]
void greet()
{
    auto job = [] () {
        for (size_t i = 0; i < 100; ++i)
                Rcpp::Rcout << "Hi!\n";
    };
    std::thread t1(job);
    std::thread t2(job);
    t1.join();
    t2.join();
}
```

We create a `job` function that prints the message `"Hi!"` to the R console 100 times. We spawn two threads that execute the job, and join them before program exits. We expect the function to print a stream of 200 messages saying `"Hi!"` in the R console. We can get lucky, but normally the two threads will try to say "Hi!" at least once at the same time. Again, the R session would terminate unexpectedly.

Now consider the following variant:

```
#include <RcppThread.h>
// [[Rcpp::export]]
void greet()
{
    auto job = [] () {
        for (size_t i = 0; i < 100; ++i)
            RcppThread::Rcout << "Hi!\n";
    };
    std::thread t1(job);
    std::thread t2(job);
    t1.join();
    t2.join();
```

```
    RcppThread::Rcout << "";
}
```

This function will print 200 messages in the R console as expected. But `RcppThread::Rcout` never prints to the console from child threads, so how does this work?

`RcppThread::Rcout` does not print to the R console directly. It first stores the message in global buffer that is protected by a lock. Then it checks whether it was called from the main thread. If this is not the case, it does nothing further. If it was called from the main thread, it releases all messages that are currently in the buffer. Notice that we print an empty message from the main thread in the last line of the program. This ensures that all messages are released from the buffer before the program exits.

3. An R-friendly thread class

As of C++11, the standard template library provides the class `std::thread` for executing code sections concurrently. The implementation and syntax are very similar to **boost.thread** and **TinyThread**. **RcppThread** 's `Thread` class is an R-friendly wrapper to `std::thread`.

Instances of class `Thread` behave almost like instances of `std::thread`. There is one important difference: Whenever child threads are running, the main thread periodically synchronizes with R. In particular, it checks for user interruptions and releases all messages passed to `RcppThread::Rcout`. When the user interrupts a threaded computation, any thread will stop as soon as it encounters `checkUserInterrupt()`.

3.1. Functionality

Let us start with an example:

```
#include <RcppThread.h>
// [[Rcpp::export]]
void pyjamaParty()
{
    using namespace RcppThread;
    auto job = [] (int id) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Rcout << id << " slept for one second" << std::endl;

        checkUserInterrupt();

        std::this_thread::sleep_for(std::chrono::seconds(1));
        Rcout << id << " slept for another second" << std::endl;
    };

    Thread t1(job, 1);
    Thread t2(job, 2);
    t1.join();
    t2.join();
}
```

We create a function `job` that takes a name as an argument and does the following: sleep for one second, send a message, check for a user interruption, go back to sleep, and send another message (in that order). We spawn two new `Thread`s with this job and join the threads before the program exits. Notice that the argument of the `job` function is passed to the `Thread` constructor. More generally, if a job function takes arguments, they must be passed to the constructor as a comma-separated list.

The example from the previous section used `std::thread` was not interruptible. The reason is that `checkUserInterrupt()` was only called from child threads. This example is similar. However, the `Thread` objects synchronize with R and periodically check for user interruptions. If we call the function from R and interrupt the computation, we get the following.

```
> pyjamaParty()

1 slept for one second
2 slept for one second
Error in pyjamaParty() : C++ call interrupted by user
```

We make two observations: First, the example was in fact interrupted after the two threads were done with the first round of sleep. Second, although there was no `Rcout` statement in the main thread, the messages got send to the R console. The `Thread` instances took care of both checking for interruptions and releasing messages to the R console.

The two `.join()` statements are important in this example. Threads should always be joined before they are destructed. The `.join()` statements signal the main thread to wait until the jobs have finished. But instead of just waiting, the main thread starts synchronizing with R.

The class `Thread` also allows for all additional functionality (like swapping or detaching) provided by `std::thread`.

3.2. Implementation

The synchronization mechanism bears some interest because it can be implemented similarly for other threading frameworks than `std::thread`. The foundation is a concept called *future*. A future allows to continue with the program, until — at some later point in time – we explicitly request the result.

Let us first have a look at a slightly simplified version of the `Thread` class constructor.

```
template<class Function, class... Args> explicit
Thread(Function&& f, Args&&... args)
{
    auto f0 = [=] { f(args...); };
    auto task = std::packaged_task<void()>(f0);
    future_ = task.get_future();
    thread_ = std::thread(std::move(task));
}
```

The constructor is a variadic template that takes a function and an arbitrary number of additional arguments. The function `f` should be a callable object and the additional arguments

such that `f(args...)` is a valid call. The constructor creates a new function `f0` that evaluates `f`, passing it all additional arguments (if there are any). The new function `f0` is wrapped in a `std::packaged_task` that allows to access the result by a future. The future is stored in a class member `future_` and the task is run in an `std::thread`.

There is synchronization mechanism is in `join()`:

```
void join()
{
   auto timeout = std::chrono::milliseconds(250);
   while (future_.wait_for(timeout) != std::future_status::ready) {
       Rcout << "";
       if (isInterrupted())
           break;
   }
   thread_.join();
   Rcout << "";
   checkUserInterrupt();
}
```

The function runs a `while` loop that relies on the future. The condition of the loop let's the main thread sleeps until one of two events occur. One event is that a timeout of 250ms has been reached. After waking up, the thread releases all messages to the R console and checks for an interruption. If there was an interruption, the call to `isInterrupted()` will set the global flag for interruption to `true` (so child threads become aware) and exits the loop. Otherwise, the `while` loop continues and the main thread again waits for the two events. The second event is that the result of `f(args...)` is available. The `while` loop exits and the internal `std::thread` object is joined. We again release all messages in the buffer and call `checkUserInterrupt()`. The latter ensures that an exception is thrown if there was a user interruption.

The choice of 250ms for the timeout is somewhat arbitrary. It is short enough to avoid long waiting times for an interrupting user. At the same time, it is long enough such that any overhead from synchronization becomes negligible.

4. Parallel abstractions

When there are more than a few jobs to run, plain threads can be tedious. Every job requires spawning and joining a thread. This has a small, but non-negligible overhead. Even worse: if there are more threads than cores, the program may actually slow down. The **RcppThread** package provides two common abstractions to make life easier. Both synchronize with R using a similar mechanism as `Thread`.

4.1. An interruptible thread pool

A *thread pool* consists of a task queue and a fixed number of worker threads. Whenever the task queue contains jobs, a waiting worker fetches one task and does the work. Besides ease of use, the thread pool pattern has two benefits. Tasks are assigned to workers dynamically

such that all workers are busy until there are no tasks left. This is especially useful when some tasks take more time than others. The second benefit is that one can easily limit the number of threads running concurrently.

Basic usage

The class `ThreadPool` implements the thread pool pattern in a way that plays nicely with `checkUserInterrupt()` and `Rcout`. It's usage is fairly simple.

```
ThreadPool pool(3);
std::vector<int> x(100);
auto task = [&x] (unsigned int i) { x[i] = i; };
for (unsigned int i = 0; i < x.size(); ++i)
    pool.push(task, i);
pool.join();
```

The first line creates a `ThreadPool` object with three worker threads. If the argument is omitted, the pool will use as many worker threads as there are cores on the machine. A thread pool initialized with zero workers will do all work in the main thread. This makes it easy to let the user decide whether computations run in parallel.

The second line instantiates a vector `x` that is to be filled in parallel. The task function takes an index argument `i` and assigns it to the `i`th element of `x`. The thread pool knows about `x` because the lambda function captures its address (expressed through `[&x]`). We push 100 tasks to the thread pool, each along with a different index. Then the thread pool is joined. Again, the `join()` statement is important. First and foremost, it causes the main thread to halt until all tasks are done. But similar to `Thread::join()`, the thread pool starts to periodically synchronize with R. Only after all work is done, the worker threads inside the pool are joined.

In the example, multiple threads write to the same object concurrently. Generally, this is dangerous. In our example, however, we know that the threads are accessing different memory locations, because each task comes with a unique index. Code that writes to an address that is accessed from another thread concurrently needs extra synchronization (for example using a *mutex*). Read operations are generally thread safe as long as nobody is writing at the same time.

The thread pool is interruptible without any explicit call to `checkUserInterrupt()`. Before a worker does a job, it always checks for a user interruption.

Tasks returning a value

In some use cases, it is more convenient to let the tasks assigned to the pool return a value. The function `pushReturn()` returns an `std::future` to the result of the computations. After all jobs are pushed, we can call `get()` on the future object to retrieve the results:

```
ThreadPool pool;
auto task = [] (int i) {
        double result;
        // some work
```

```
        return result;
};
std::vector<std::future<double>> futures(10);
std::vector<double> results(10);
for (unsigned int i = 0; i < 10; ++i)
    futures[i] = pool.pushReturn(task, i);
for (unsigned int i = 0; i < 10; ++i)
    results[i] = futures[i].get();
pool.join();
```

Using the same thread pool multiple times

It is also possible to wait for a set of tasks to be done and re-use the thread pool by calling `pool.wait()`. The call to `wait()` synchronizes with R while waiting for the current jobs to finish, but does not join the worker threads. When all tasks are done, we can start pushing new jobs to the pool.

4.2. Parallel for loops

Index-based parallel for loops

The previous example used the thread pool to implement a very common parallel pattern: a parallel `for` loop. The single-threaded version is much simpler.

```
std::vector<int> x(100);
for (unsigned int i = 0; i < x.size(); ++i)
    x[i] = i;
```

Since this pattern is so common, **RcppThread** provides a wrapper `parallelFor` that encapsulates the boiler plate from the thread pool example. A parallel version of the above example can be expressed similarly.

```
std::vector<int> x(100);
parallelFor(0, x.size(), [&x] (unsigned int i) {
    x[i] = i;
});
```

Although the two version look quite similar, there are differences between the single- and multi-threaded version. The single-threaded version instantiates the loop counter in the `for` declaration. The multi-threaded version, passes the start and end indices and a lambda function that captures `&x` and takes the loop counter as an argument. The latter version is a bit less flexible regarding the loop's break condition and increment operation. Additionally, the multi-threaded version may need additional synchronization if the same memory address is written in multiple iterations.

Parallel for-each loops

Another common pattern is the for-each loop. It loops over all elements in a container and

applies the same operation to each element. A single-threaded example of such a loop is the following.

```
std::vector<int> x(100, 1);
for (auto& xx : x)
    xx *= 2;
```

The `auto` loop runs over all elements of `x` an multiplies them by two. The parallel version is similar:

```
std::vector<int> x(100, 1);
parallelForEach(x, [] (double& xx) {
    xx *= 2;
});
```

Both `parallelFor` and `parallelForEach` use a `ThreadPool` object under the hood. As such, they are interruptible by default and periodically synchronize with R.

Fine tuning the scheduling system

The two functions `parallelFor` and `parallelForEach` essentially create a thread pool, push the tasks, join the pool, and exit. By default, there are as many worker threads in the pool as there are cores on the machine. The number of workers can be specified manually, however. The following example runs the loop from the previous example with only two workers (indicated by the `2` in the last line).

```
parallelForEach(x, [] (double& xx) {
    xx *= 2;
}, 2);
```

The syntax for `parallelFor` is similar.

There is more: the `parallelFor` and `parallelForEach` functions bundle a set of tasks into batches. This can speed up code significantly when the loop consists of many short-running iterations. Synchronization between worker threads in the pool causes overhead that is reduced by packaging tasks into batches. At the same time, we benefit from dynamic scheduling whenever there are more batches than tasks. **RcppThread** relies on heuristics to determine an appropriate batch size automatically. Sometimes, the performance of the loops can be improved by a more careful control over the batch size. The two functions take a fourth argument that allows to set the number of batches. The following code runs the loop with two workers in 20 batches.

```
parallelForEach(x, [] (double& xx) {
    xx *= 2;
}, 2, 20);
```

Calling for loops from a thread pool

The functions create and join a thread pool every time they are called. To reduce overhead, the functions can also be called as methods of a thread pool instance.

```
ThreadPool pool;
pool.parallelForEach(x, [] (double& xx) {
    xx = 2 * xx;
});
pool.wait();
pool.parallelFor(0, x.size(), [&x] (int i) {
    x[i] *= 2;
});
pool.join();
```

Nested for loops

Nested loops appear naturally when operating on multi-dimensional arrays. One can also nest the parallel for loops mentioned above. Although not necessary, it is more efficient to use the same thread pool for both loops.

```
ThreadPool pool;
std::vector<std::vector<double>> x(100);
for (auto &xx : x)
    xx = std::vector<double>(100, 1.0);
pool.parallelFor(0, x.size(), [&] (int i) {
    pool.parallelFor(0, x[i].size(), [&, i] (int j) {
        x[i][j] *= 2;
    });
});
pool.join();
```

The syntax for nested `parallelForEach` is similar.

A few warnings: It is usually more efficient to run only the outer loop in parallel. To minimize overhead, one should parallelize at the highest level possible. Furthermore, if both inner and outer loops run in parallel, we do not know the execution order of tasks. We must not parallelize nested loops in which order matters. Captures of lambda functions (or other callable objects replacing the loop body) require extra care: since the outer loop index `i` is temporary, it must be copied.

5. Using the RcppThread package in other projects

The **RcppThread** package contains a header-only C++ library that only requires a C++11 compatible compiler. This is only a mild restriction, because the standard has long been implemented by most common compilers. To use the package in other R projects, users only need to include the **RcppThread** headers and enable C++11 functionality. In the following, we briefly explain how this can be achieved.

5.1. Using RcppThread in inline C++ snippets

The `cppFunction()` and `sourceCpp()` functions of the **Rcpp** package provide a convenient

way to quickly implement a C++ function inline from R . The following is a minimal example of **RcppThread** used with `cppFunction()`:

```
func <- Rcpp::cppFunction('void func() { /* actual code here */ }',
                          depends = "RcppThread", plugins = "cpp11")
```

The first argument of `cppFunction()` is a C++ snippet defining a function. The `depends = "RcppThread"` argument takes care that the relevant **RcppThread** headers are included. Finally, `plugins = "cpp11"` tells the compiler to enable C++11 functionality. After running this line in the R console, `func()` can be called as a regular R function.

The same can be achieved using `sourceCpp()`:

```
Rcpp::sourceCpp(code =
    '// [[Rcpp::plugins(cpp11)]]
    // [[Rcpp::depends(RcppThread)]]
    #include "RcppThread.h"
    // [[Rcpp::export]]
    void func() { /* actual code here */}'
)
```

Inside the `code` block, the first line enables C++11, the second tells the compiler where to look for **RcppThread** headers, which are included in the third line. The fourth and fifth lines define the function and request it to be exported to R . The `sourceCpp()` is a bit more verbose, but offers additional flexibility, since other functions or classes can be defined in the same code block.

5.2. Using RcppThread in another R package

Using **RcppThread** in other R packages is similarly easy:

1. Add the the line `CXX_STD = CXX11` to the `src/Makevars(.win)` files of your package.
2. Add `RcppThread` to the `LinkingTo` field in the `DESCRIPTION`.
3. Include the headers with `#include "RcppThread.h"`.

5.3. Using RcppThread to port existing C++ code

For packages porting existing C++ libraries to R , **RcppThread** provides two preprocessor macros for convenience. The respective `#define` statements need to be placed before including the **RcppThread** headers.

- `#define RCPPTHREAD_OVERRIDE_COUT 1`: replaces all instances of `std::cout` by `RcppThread::Rcout`.
- `#define RCPPTHREAD_OVERRIDE_THREAD 1`: replaces all instances of `std::thread` by `RcppThread::Thread`.

6. Benchmarks

Parallel computation is primarily about speed, so it's a good idea to measure. In a first step, we want to quantify the overhead by **RcppThread** 's synchronization with R . The second part compares the performance of the `ThreadPool` and `parallelFor` abstractions against competitor implementations using **RcppParallel** and **OpenMP**.

Results of computing benchmarks depend strongly on the hardware, especially for concurrent programs. The following results in this section were recorded on a i5-6600K CPU with four cores at 3.5 GHz, 6MB cache size, and 16GB 3GHz DDR4 RAM. The code for the benchmarks is available at `https://gist.github.com/tnagler/369d6e36e6e6b375bac0a9c57840aca7`, so readers can run test them on their own machine.

6.1. Synchronization overhead

Create, join, and destruct threads

As explained in Section 3.2, `RcppThread::Thread` encapsulates a `std::thread` object, but exploits a `std::future` for additional synchronization. To quantify the overhead, our first example simply creates a number of thread objects, and then joins and destroys them.

The speed using `RcppThread::Thread` (dashed) is compared against `std::thread` (solid) in Figure 1. We observe that `RcppThread::Thread` is roughly two times slower than `std::thread`. Both lines show a kink at four threads. This corresponds to the four physical cores on the benchmark machine. If there are more threads than cores, we pay an additional fee for 'context switching' (jumping between threads).

The marginal cost of a single `RcppThread::Thread` is around 10μs when there are less than four threads, and roughly 30μs otherwise. Although even 30μs sounds cheap, running more threads than cores will slow down also all other computations. Thread pools or parallel loops use a fixed number of threads and should be preferred over naked threads when possible.

Checking for user interruptions

Figure 2 shows the time it takes to call `checkUserInterrupt()` either from the main or a child thread. Checking for user interruptions is rather cheap: one check costs around 100ns from the main thread, and 5ns from a child thread. The latter is cheaper because there is no direct synchronization with R. If called from a child thread, `checkUserInterrupt()` only queries a global flag. Hence, we can generously sprinkle parallelized code sections with such checks without paying much in performance.

6.2. Comparison to other parallel libraries

Empty jobs

To start, we measure the performance for running empty jobs. The solid line in Figure 3 indicates the time required to run a single threaded loop with `jobs` iterations doing nothing. The other lines show the performance of various parallel abstractions: **RcppThread** 's `ThreadPool` and `parallelFor`, and parallel for loops based on **OpenMP** and **RcppParallel**.

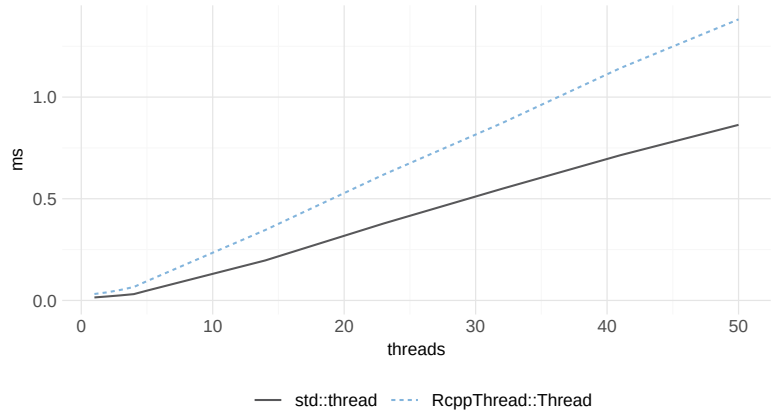

Figure 1: Time required for creating, joining, and destroying thread objects of class `std::thread` and `RcppThread::Thread`.

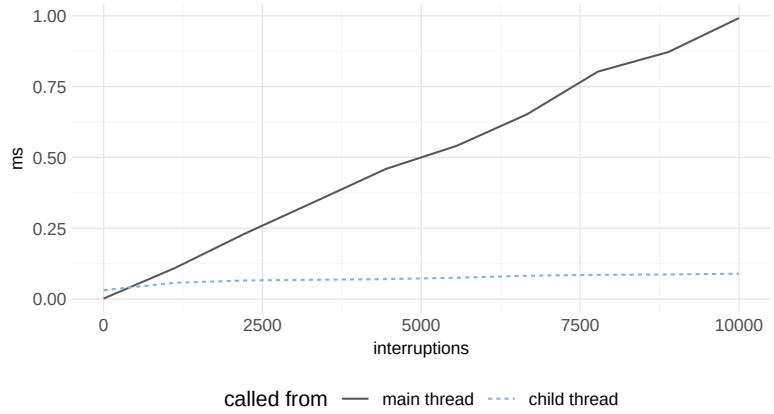

Figure 2: Time required to check for user interruptions from either the main or a child thread.

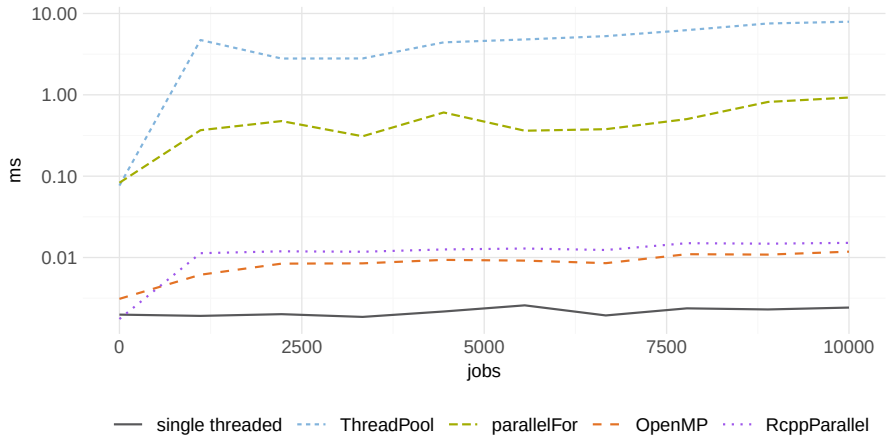

Figure 3: Time required for submitting empty jobs to different parallelism frameworks.

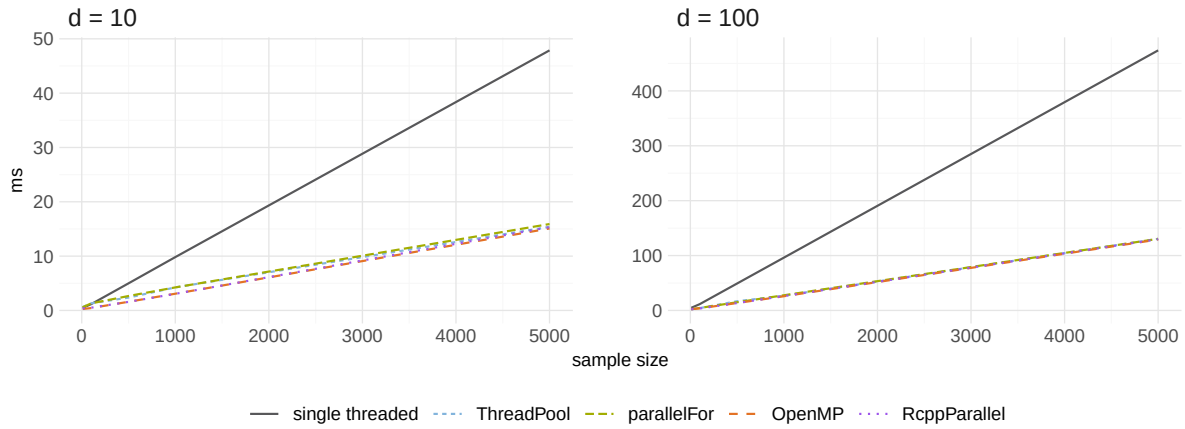

Figure 4: Time required for computing the kernel density estimate in parallel for d variables.

The abstractions provided by **RcppThread** are much slower than their competitors (note the log scale on the y axis). This has two reasons. The **RcppThread** functions are weighted with infrastructure for synchronization with R. In contrast, the competitor libraries are highly optimized for high-throughput scenarios by avoiding memory locks as much as possible.

We also observe that the parallel for loops are are much faster than the thread pool. Since the thread pool accepts new jobs at any time, it must handle any job as a separate instance. Parallel for loops know up front how much work there is and bundle jobs into a smaller number of batches. This technique reduces the necessary amount of synchronization between threads.

The single threaded version was the fastest, by far. Of course, we cannot expect any gain from parallelism when there is nothing to do. When jobs are that light, the overhead vastly outweighs the benefits of concurrent computation.

Computing kernel density estimates

Let us consider a scenario that is more realistic for statistical applications. Suppose we observe data from several variables and want to compute a kernel density estimate (KDE) for each variable. This is a common task in exploratory data analysis or nonparametric inference (e.g., the naive Bayes classifier) and is easy to parallelize. For simplicity, the estimator is evaluated on 500 grid points.

Figure 4 shows the performance for $d = 10$ (left panel), $d = 100$ variables (right panel), and increasing sample size. For $d = 10$ and moderate sample size the two **RcppThread** functions are about 10% slower than their competitors, but catch up for large samples. The shift is essentially the overhead we measured in the previous benchmark. For $d = 100$, the overhead of **RcppThread** is negligible and all methods are on par. Generally, all parallel methods are approximately 4x faster than the single threaded version.

Computing Kendall's correlation matrix

Now suppose we want to compute a matrix of pair-wise Kendall's correlation coefficients. Kendall's τ is a popular rank-based measure of association. In contrast to Pearson's cor-

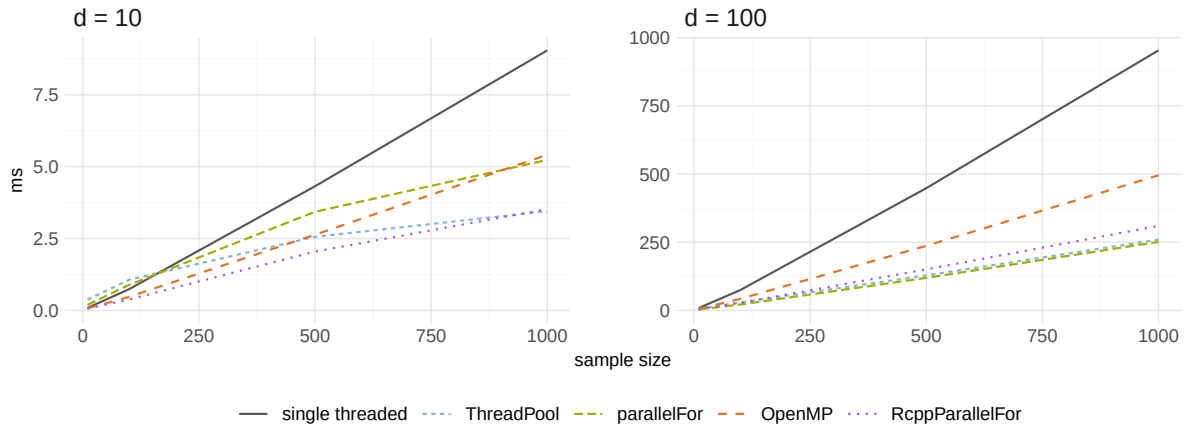

Figure 5: Time required for computing Kendall's correlation matrix in parallel for d variables. TODO sample size

relation, it measures monotonic (not just linear) dependence, but is computationally more complex. For example, R's implementation in `cor()` scales quadratically in the sample size n. Knight (1966) proposed an efficient algorithm based on a merge sort that scales $n \log n$ (as implemented in Nagler 2018). As a downside, the correlation matrix can no longer be computed with matrix algebra; each coefficient $\tau_{i,j}$ must be considered separately. There are $\binom{d}{2}$ unique pairs of variables (i, j), $1 \leq i < j \leq d$. The coefficients are computed in a nested loop over i and j, where we only parallelize the outer loop over i.

This problem is quite different from the KDE benchmark. First, the problem scales quadratically in the dimension d. And more importantly, the jobs are unbalanced: for each i, there are only $d - i$ iterations in the inner loop. Hence, iterations with small i take longer than iterations with large i. The larger the dimension d, the larger the imbalance.

For $d = 10$, we observe that none of the parallel methods achieve a 4x speed up. The reason is that the tasks are still rather small. Even for $n = 1\,000$, each iteration of the inner loop takes only a fraction of a millisecond. For such jobs, the parallelization overhead becomes visible. `parallelFor` is slowest among all methods. For sample sizes smaller than 500, it is hardly faster than the single threaded loop. Also OpenMP achieves less than a 2x improvement. Only `ThreadPool` and `RcppParallel` achieve an approximate 3x speed up. Their scheduling appears to better compensate the imbalance of the problem.

For $d = 100$, the picture is quite different. The **RcppThread** functions are faster than their competitors: roughly twice as fast as **OpenMP** and 10% faster than **RcppParallel**. Furthermore, it gives an approximate 4x speed up, indicating an optimal use of resources.

6.3. Conclusions

We conclude that the parallel abstractions provided by **RcppThread** cause notable overhead when concurrent tasks are small. For many applications in statistical computing, however, this overhead becomes negligible. In the future, the implementation **RcppThread** may benefit from additional optimizations. In particular, a lock free implementation of the task queue may allow to reduce the overhead on small tasks. In any case, the main advantage is automatic and safe synchronization with R, i.e., usability and not speed.

References

Allaire J, Francois R, Ushey K, Vandenbrouck G, Geelnard M, Intel (2018). *RcppParallel: Parallel Programming Tools for 'Rcpp'*. R package version 4.4.1, URL https://CRAN.R-project.org/package=RcppParallel.

Boost (2018). "Boost C++ Libraries." http://www.boost.org/.

Dagum L, Menon R (1998). "OpenMP: an Industry Standard API for Shared-Memory Programming." *IEEE computational science and engineering*, **5**(1), 46–55.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Springer, New York. doi:10.1007/978-1-4614-6868-4. ISBN 978-1-4614-6867-7.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08. URL http://www.jstatsoft.org/v40/i08/.

Forner K (2018). *RcppProgress: An Interruptible Progress Bar with OpenMP Support for C++ in R Packages*. R package version 0.4.1, URL https://CRAN.R-project.org/package=RcppProgress.

Geelnard M (2012). *TinyThread*. C++ library version 1.1, URL http://tinythreadpp.bitsnbites.eu/.

Knight WR (1966). "A Computer Method for Calculating Kendall's tau With Ungrouped Data." *Journal of the American Statistical Association*, **61**(314), 436–439.

Nagler T (2018). *wdm: Weighted Dependence Measures*. C++ library version 0.1.1, URL https://github.com/tnagler/wdm.

Pheatt C (2008). "Intel® Threading Building Blocks." *Journal of Computing Sciences in Colleges*, **23**(4), 298–298.

Sutter H (2005). "The Concurrency Revolution." *C/C++ Users Journal*, **23**(3).

Affiliation:

Thomas Nagler
Technische Universität München
Zentrum Mathematik
Lehrstuhl für Mathematische Statistik
Boltzmannstraße 3, 85748 Garching
E-mail: mail@tnagler.com
URL: http://tnagler.com