科研认知与专题研究(2)课程报告

姓名　　杨哲涵

指导教师　续本达

1. 概述

自从 3 月 28 日起,邓雲峰,我和罗新阳在武益阳学长的指导下参与了清华自研电子学采数程序的 SRT 创新项目.

前期我了解和部分学习了 CERN 开发的 ROOT 实验物理数据分析框架,并且阅读了锦屏 1 吨原型的电子学部分的论文和代码,大概了解了目前 CAEN 电子学采数的工作流程.

4 月 26 日后,张老师和邓雲峰在锦屏现场部署安装了 64 路电子学系统,发现采数程序在输入信号 580Hz 以上时遇到了速度瓶颈,读出一段时间后会崩溃.为此,五一假期结束后,武益阳学长安排我探索多线程压缩写入的例程.我根据网上有关的资料,依靠武益阳,王宇逸学长的指导和帮助,尝试了 C++并行编程.我在测试中发现,通过并行压缩写入,至少可以将写入速度提升到单线程的 3 倍以上.在此之后,武益阳学长指导我将相关例程应用到上位机采数程序上.

2. 采数程序多线程压缩写入

2.1. `TBufferMerger`

自 ROOT 6.10 起, `TBufferMerger` 类被用于多线程并行写入文件的场景, 使用时每个要写入文件的进程从 `TBufferMerger` 中获取一个 `TBufferMergerFile` 作为其操作的文件, 随后就可以像正常操作 `TFile` 一样编写代码.

Writing ROOT Data in Parallel with TBufferMerger 提供了一个多线程并行写入的例子.我在接下来编写程序的时候参考了论文中的例程.

```
std::shared_ptr<TBufferMergerFile>
TBufferMerger::GetFile()
{
   R__LOCKGUARD(gROOTMutex);
   std::shared_ptr<TBufferMergerFile> f(new
TBufferMergerFile(*this));
   gROOT->GetListOfFiles()->Remove(f.get());
   fAttachedFiles.push_back(f);
   return f;
}

void
TBufferMerger::Merge(ROOT::TBufferMergerFile
*memfile)
{
   std::lock_guard q(fMergeMutex);
   memfile->WriteStreamerInfo();
   fMerger.AddFile(memfile);
   fMerger.PartialMerge(TFileMerger::kAll |
TFileMerger::kIncremental |
TFileMerger::kDelayWrite |
TFileMerger::kKeepCompression);
   fMerger.Reset();
}
```

每通过 `GetFile()`获取一个 `TBufferMergerFile`,都会被 `TBufferMerger` 的实例保留在私有成员变量 `fAttachedFiles` 中.并且,每次 `GetFile()`获得的 `TBufferMergerFile` 都是独立的,保证了线程安全.

因此,作为 ROOT 提供的一种较为便捷和底层的多线程写入文件的方法,`TBufferMerger` 可以优化计算密集型任务的速度.

2.2. 线程池并行写入

在了解了 `TBufferMerger` 的基本使用方法后,我尝试了使用 `BS_thread_pool` 线程池并行写入文件,写入速度相比单线程,至少是其 3 倍.可是,由于多线程执行顺序的不确定性,以及 `TBufferMerger` 合并过程用时的涨落,在实际应用中,记录的事例顺序会不同于原始采集到的顺序.

2.3. 解决保序问题

为了还原原先的事例的顺序,需要在 `TBufferMerger` 写完后再对事例进行排序,这一过程中不能再使用多线程对单一文件进行写入,但是开多个线程,每个线程内写入一个文件是可以的.

最终的优化方法流程如下:

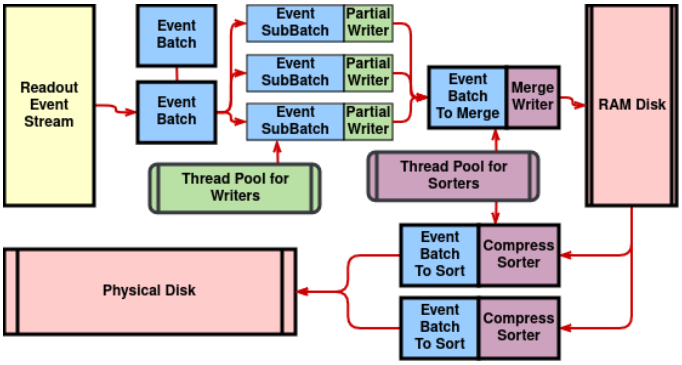

图 1　线程池并行写入流程

3. 结语

6 月 7 日,我和武益阳学长进行了结对编程,初步将采数程序的多线程压缩写入功能移植到了上位机采数程序 `Readout.C` 当中.由于锦屏服务器上的 ROOT 和 `cmake` 版本低于编译最低需求,还需升级依赖版本.实际采数测试将在未来的几天中进行.

在此我感谢续老师,武益阳,王宇逸,邓雲峰学长对我的指导和帮助,没有他们,我无法完成这次的任务.

此外感谢现代 C++的众多好用的特性,以及 ROOT 的强大功能,让我能够方便地编写代码.

当然还有使用到的开源 C++库:

- BS_thread_pool
- spdlog
- argparse

4. 附录

以下是我编写的多线程并行写入的例程:

```
/* Concurrent.C */
#include <future>
#include <queue>
#include <csignal>
#include <random>
#include <mutex>
#include <memory>

#include "ROOT/TBufferMerger.hxx"
#include "RtypesCore.h"
#include "TFile.h"
#include "TTree.h"
#include "TTreeReader.h"
#include "TTreeReaderValue.h"
#include "TBranch.h"
#include "TROOT.h"

#include "spdlog/spdlog.h"
#include "argparse/argparse.hpp"
#include "BS_thread_pool.hpp"

std::atomic<bool> should_shutdown = false;

std::mutex one_file_writer_pool_mutex;

void graceful_shutdown(int signum)
{
    if (should_shutdown.load())
        return;
    should_shutdown.store(true);
    spdlog::info("receive shutdown"); // deny
any new task and wait for all tasks to finish
}

/// @brief simply write entries from multiple
threads to a single file without compression
class EventBatchWriter
{
private:
    ROOT::TBufferMerger merger;
    std::queue<std::future<void>> results;
    /// @brief must acquire mutex while
submiting tasks to pool since the pool is
shared among all writers
    std::shared_ptr<BS::thread_pool> pool;
    /// @brief limit for number of entries per
file
    size_t size_limit = 100000;
    size_t size = 0;

    void wait_for_all_results()
    {
        std::signal(SIGINT, graceful_shutdown);
        while (!results.empty())
            update_result(1);
    }
    void update_result(size_t t)
    {
        std::signal(SIGINT, graceful_shutdown);
        if
(results.front().wait_for(std::chrono::milliseconds(t)
== std::future_status::ready)
        {
            results.pop();
            spdlog::debug("pop a result from
{}.root and {} left", filename,
results.size());
        }
    };
    void save_sequence(size_t cnt, size_t
index)
    {
        std::signal(SIGINT, graceful_shutdown);
        auto f = merger.GetFile();
        TTree t("test", "Test");
        std::vector<unsigned int>
waveform(1000);
        t.Branch("Waveform", &waveform);
        for (size_t n = 0; n < cnt; ++n)
        {
            for (size_t j = 0; j < 1000; j++)
                waveform[j] = (index + n) *
1000 + j;
            t.Fill();
        }
        f->Write();
        spdlog::debug("save entry {}-{} to
{}.root", index, index + cnt, filename);
    }

public:
    /// @brief root filename without extension
    std::string filename;
    std::string base_filename;
    EventBatchWriter(const std::string
&output_file_name, const std::string &temp_dir,
std::shared_ptr<BS::thread_pool> p, size_t l) :
merger(ROOT::TBufferMerger(TString(temp_dir +
"/" + output_file_name + ".root"), "recreate",
ROOT::RCompressionSetting::ELevel::kUncompressed)),
pool(p), size_limit(l), filename(temp_dir + "/"
```

```
+ output_file_name),
base_filename(output_file_name)
    {
        spdlog::info("create writer {}.root",
output_file_name);
    }
    bool is_too_large()
    {
        return size >= size_limit;
    }
    void submit_task(size_t cnt, size_t index)
    {
        const std::lock_guard<std::mutex>
lock(one_file_writer_pool_mutex);
        results.emplace(pool-
>submit_task([this, cnt, index]()

{ save_sequence(cnt, index); }));
        size += cnt;
    }
    ~EventBatchWriter()
    {
        wait_for_all_results();
        spdlog::info("finish writing {}.root to
disk", filename);
    }
};

void into_sorter(EventBatchWriter *writer)
{
    std::signal(SIGINT, graceful_shutdown);
    std::string filename = writer->filename;
    std::string final_filename = "s_" + writer-
>base_filename;
    // manually calling destructor so that the
IO writing be done and the file is prepared
before sorting
    delete writer;
    TFile *raw_file =
TFile::Open(TString(filename + ".root"));
    spdlog::info("open {}.root for reading
uncompressed data", filename);
    TTreeReader r("test", raw_file);
    TTreeReaderValue<std::vector<unsigned int>>
r_waveform(r, "Waveform");
    std::vector<unsigned int> w_waveform;
    TFile *sorted_file =
TFile::Open(TString(final_filename + ".root"),
"recreate"); // to be sorted and compressed
    spdlog::info("open {}.root for writing
compressed data", final_filename);
    TTree t("test", "Test");
    t.Branch("Waveform", &w_waveform);
    std::vector<std::vector<unsigned int>>
waves;
    while (r.Next())
        waves.emplace_back(*r_waveform);
    std::sort(waves.begin(), waves.end(), []
(std::vector<unsigned int> &a,
std::vector<unsigned int> &b)
              { return a[0] < b[0]; }); //
ascending order
    for (size_t i = 0, N = waves.size(); i < N;
++i)
    {
        w_waveform = waves[i];
        t.Fill();
    }
    sorted_file->Write();
    delete raw_file;
    delete sorted_file;
    spdlog::info("finish writing compressed
data to {}.root", filename);
}

int main(int argc, char **argv)
{
    std::signal(SIGINT, graceful_shutdown);
    argparse::ArgumentParser
program("Concurrent", "0.1.0",
argparse::default_arguments::all, true);
    program.add_description("multithread raw
writing with retarded compression demo");
    std::string output_file_name =
"concurrent";
    program.add_argument("-o", "--
output").help("assign output
filename").store_into(output_file_name);
    size_t entries_cnt = 1000;
    program.add_argument("-n").help("assign
test entires size").store_into(entries_cnt);
    size_t per_thread_entries_cnt = 1000;
    program.add_argument("-p").help("assign
test per thread entires
size").store_into(per_thread_entries_cnt);
    size_t w_threadpool_size = 4;
    program.add_argument("-w").help("assign
writer thread pool
size").store_into(w_threadpool_size);
    size_t s_threadpool_size = 8;
    program.add_argument("-s").help("assign
sorter thread pool
size").store_into(s_threadpool_size);
    int log_level = spdlog::level::debug;
    program.add_argument("-l").help("assign log
level").store_into(log_level);
    std::string temp_dir = "/tmp";
    program.add_argument("-t").help("Directory
for temp files").store_into(temp_dir);
    try
    {
        program.parse_args(argc, argv);
    }
    catch (const std::exception &err)
    {
        std::cerr << err.what() << std::endl;
        std::cerr << program;
        return 1;
    }


spdlog::set_level((spdlog::level::level_enum)log_level
    spdlog::info("start");
    spdlog::info("enable ROOT ThreadSafety");
```

```
    ROOT::EnableThreadSafety(); // Must have this line, otherwise segment fault

    spdlog::info("create writer pool of {} thread(s)", w_threadpool_size);
    std::shared_ptr<BS::thread_pool> w_pool = std::make_shared<BS::thread_pool>(w_threadpool_size);
    spdlog::info("create sorter pool of {} thread(s)", s_threadpool_size);
    BS::thread_pool s_pool(s_threadpool_size);


std::queue<std::unique_ptr<EventBatchWriter>> writers;
    size_t writers_cnt = 0;

    std::queue<std::future<void>> sorters;

    auto pop_writer = [&writers, &sorters, &s_pool]()
    {
        auto f = writers.front().release(); // to transfer onwership but not delete
        writers.pop();
        spdlog::debug("pop a writer while submiting and {} left", writers.size());

sorters.emplace(s_pool.submit_task([&writers, f]()

{ into_sorter(f); })); // onwership received here and will be deleted
        spdlog::debug("push a sorter while submiting and {} left", sorters.size());
    };

    auto pop_sorter = [&sorters]()
    {
        if (sorters.front().wait_for(std::chrono::milliseconds(0)) == std::future_status::ready)
        {
            sorters.pop();
            spdlog::debug("pop a sorter and {} left", sorters.size());
        }
    };

    for (size_t i = 0; i < entries_cnt && !should_shutdown.load(); i += per_thread_entries_cnt)
    {
        if (writers.empty() || writers.back()->is_too_large())
        {

writers.emplace(std::make_unique<EventBatchWriter>(output_file_name + "_" + std::to_string(writers_cnt++), temp_dir, w_pool, 100000));
            spdlog::debug("push a writer while submiting and {} left", writers.size());
        }
                spdlog::debug("submit task {}-{}", i, i + per_thread_entries_cnt);
                writers.back()->submit_task(per_thread_entries_cnt, i);

                if (!writers.empty() && writers.front()->is_too_large())
                    pop_writer();

                if (!sorters.empty())
                    pop_sorter();
            }

            spdlog::info("finish submiting tasks");
            while (!writers.empty())
                pop_writer();

            while (!sorters.empty())
                pop_sorter();

            spdlog::info("exit");
            return 0;
        }
```