Wrap Up & Discussion

Some final words on this topic.

WE'LL COVER THE FOLLOWING

- Additional Modifications and Options

The main aim of this section was to show how easy it is to use parallel algorithms.

The final code is located in the two tabs below:

Serial | Parallel

```
#include <algorithm>
#include <charconv>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <optional>
#include <string_view>
#include <string>
#include <utility>
#include <vector>

#include "scope_timer.h"

static const char* const CSV_EXTENSION = ".csv";
static constexpr char DEFAULT_DATE_DELIM = '-';
static constexpr char CSV_DELIM = ';';

namespace fs = std::filesystem;

[[nodiscard]] std::string GetFileContents(const fs::path& filePath)
{
    std::ifstream inFile{ filePath, std::ios::in | std::ios::binary };
    if (!inFile)
        throw std::runtime_error("Cannot open " + filePath.filename().string());

    std::string str(static_cast<size_t>(fs::file_size(filePath)), 0);

    inFile.read(str.data(), str.size());
    if (!inFile)
        throw std::runtime_error("Could not read the full contents from " + filePath.filename


    return str;
}

[[nodiscard]] std::vector<std::string_view> SplitString(std::string_view str, char delim)
{
    std::vector<std::string_view> output;

    const auto last = str.end();
    for (auto first = str.begin(), second = str.begin(); second != last && first != last; fir
    {
        second = std::find(first, last, delim);

        // we might get empty string views here, but that's ok in the case of CSV reader
        //output.emplace_back(str.substr(std::distance(str.begin(), first), std::distance(fir
        output.emplace_back(&*first, std::distance(first, second));

        if (second == last)
            break;
    }

    return output;
}

[[nodiscard]] std::vector<std::string_view> SplitLines(std::string_view str)
{
    auto lines = SplitString(str, '\n');

    if (!lines.empty() && lines[0].back() == '\r') // Windows CR conversion
    {
        for (auto &&line : lines)
        {
            if (line.back() == '\r')
                line.remove_suffix(1);
        }
    }

    return lines;
}

template<typename T>
[[nodiscard]] std::optional<T> TryConvert(std::string_view sv) noexcept
{
    T value{ };
    const auto last = sv.data() + sv.size();
    const auto res = std::from_chars(sv.data(), last, value);
    if (res.ec == std::errc{} && res.ptr == last)
        return value;

    return std::nullopt;
}

// as of Dec 2018 only MAVC supports floating point conversion
// this code might be removed when GCC/Clang handles it as well
#ifndef _MSC_VER
template<>
[[nodiscard]] std::optional<double> TryConvert(std::string_view sv) noexcept
{
    std::string str{ sv };
    return ::atof(str.c_str());
```

```
}
#endif


class Date
{
public:
    enum class Format { DayMonthYear, YearMonthDay };
public:
    Date() = default;
    Date(uint8_t day, uint8_t month, uint16_t year) noexcept : mDay(day), mMonth(month), mYea
    explicit Date(std::string_view sv, char delim = DEFAULT_DATE_DELIM, Format fmt = Format::

    bool IsInvalid() const noexcept {
        return mDay == 0 || mMonth == 0 || mYear == 0 || mDay > 31 || mMonth > 12;
    }

    friend bool operator < (const Date& lhs, const Date& rhs) noexcept {
        return std::tie(lhs.mYear, lhs.mMonth, lhs.mDay) < std::tie(rhs.mYear, rhs.mMonth, rh
    }

    friend bool operator <= (const Date& lhs, const Date& rhs) noexcept {
        return std::tie(lhs.mYear, lhs.mMonth, lhs.mDay) <= std::tie(rhs.mYear, rhs.mMonth, r
    }

    friend std::ostream& operator<<(std::ostream &os, const Date &d) noexcept {
        return os << d.mDay << DEFAULT_DATE_DELIM << d.mMonth << DEFAULT_DATE_DELIM << d.mYea
    }

private:
    uint8_t mDay{ 0 }; // 1...31, 0 is invalid
    uint8_t mMonth{ 0 }; // 1...12, 0 is invalid
    uint16_t mYear{ 0 }; // 0 means invalid
};

Date::Date(std::string_view sv, char delim, Date::Format fmt)
{
    const auto columns = SplitString(sv, delim);
    if (columns.size() == 3)
    {
        mDay = TryConvert<uint8_t>(columns[fmt == Format::DayMonthYear ? 0 : 2]).value_or(0);
        mMonth = TryConvert<uint8_t>(columns[1]).value_or(0);
        mYear = TryConvert<uint16_t>(columns[fmt == Format::DayMonthYear ? 2 : 0]).value_or(0

        if (IsInvalid())
            throw std::runtime_error("Cannot convert date from " + std::string(sv));
    }
    else
        throw std::runtime_error("Cannot convert date, wrong element count/format, " + std::s
}


class OrderRecord
{
public:
    OrderRecord() = default;
    OrderRecord(Date date, std::string coupon, double unitPrice, double discount, unsigned in
        : mDate(date)
        , mCouponCode(std::move(coupon))
        , mUnitPrice(unitPrice)
        , mDiscount(discount)
        , mQuantity(quantity)
    { }
    double CalcRecordPrice() const noexcept { return mQuantity * (mUnitPrice*(1.0 - mDiscount
    bool CheckDate(const Date& startDate, const Date& endDate) const noexcept {
        return (startDate.IsInvalid() || startDate <= mDate) && (endDate.IsInvalid() || mDate
    }

public:
    // not enum class so that we can easily use it as array index
    enum Indices { DATE, COUPON, UNIT_PRICE, DISCOUNT, QUANTITY, ENUM_LENGTH };

private:
    Date mDate;
    std::string mCouponCode;
    double mUnitPrice{ 0.0 };
    double mDiscount{ 0.0 }; // 0... 1.0
    unsigned int mQuantity{ 0 };
};

[[nodiscard]] OrderRecord LineToRecord(std::string_view sv)
{
    const auto columns = SplitString(sv, CSV_DELIM);
    if (columns.size() == static_cast<size_t>(OrderRecord::ENUM_LENGTH)) // assuming we also
    {
        const auto unitPrice = TryConvert<double>(columns[OrderRecord::UNIT_PRICE]);
        const auto discount = TryConvert<double>(columns[OrderRecord::DISCOUNT]);
        const auto quantity = TryConvert<unsigned int>(columns[OrderRecord::QUANTITY]);

        if (unitPrice && discount && quantity)
        {
            return { Date(columns[OrderRecord::DATE]),
                     std::string(columns[OrderRecord::COUPON]),
                     *unitPrice,
                     *discount,
                     *quantity };
        }
    }
    throw std::runtime_error("Cannot convert Record from " + std::string(sv));
}

[[nodiscard]] std::vector<OrderRecord> LinesToRecords(const std::vector<std::string_view>& li
{
    //ScopeTimer _t(__func__);

    std::vector<OrderRecord> outRecords;
    std::transform(lines.begin(), lines.end(), std::back_inserter(outRecords), LineToRecord);

    return outRecords;
}

[[nodiscard]] std::vector<OrderRecord> LoadRecords(const fs::path& filename)
{
    //ScopeTimer _t(__func__);
    const auto content = GetFileContents(filename);

    ScopeTimer _t("Parsing Strings", /*store*/true);
    const auto lines = SplitLines(content);

    return LinesToRecords(lines);
}

[[nodiscard]] double CalcTotalOrder(const std::vector<OrderRecord>& records, const Date& star
{
```

```
{
    ScopeTimer _t(__func__, /*store*/true);

    return std::accumulate(std::begin(records), std::end(records), 0.0,
        [&startDate, &endDate](double val, const OrderRecord& rec) {
            if (rec.CheckDate(startDate, endDate))
                return val + rec.CalcRecordPrice();
            else
                return val;
        }
    );
}

bool IsCSVFile(const fs::path &p)
{
    return fs::is_regular_file(p) && p.extension() == CSV_EXTENSION;
}

[[nodiscard]] std::vector<fs::path> CollectPaths(const fs::path& startPath)
{
    std::vector<fs::path> paths;
    fs::directory_iterator dirpos{ startPath };
    std::copy_if(fs::begin(dirpos), fs::end(dirpos), std::back_inserter(paths), IsCSVFile);
    return paths;
}

struct Result
{
    std::string mFilename;
    double mSum{ 0.0 };
};

[[nodiscard]] std::vector<Result>
CalcResults(const std::vector<fs::path>& paths, Date startDate, Date endDate)
{
    ScopeTimer _t(__func__, /*store*/true);
    std::vector<Result> results;
    for (const auto& p : paths)
    {
        const auto records = LoadRecords(p);

        const auto totalValue = CalcTotalOrder(records, startDate, endDate);
        results.push_back({ p.string(), totalValue });
    }
    return results;
}

void ShowResults(const std::vector<Result>& results, Date startDate, Date endDate)
{
    const size_t maxStringLen = std::accumulate(std::cbegin(results), std::cend(results), 0,
        [](size_t l, const auto &result) { return std::max(l, result.mFilename.length()); }
    );

    std::cout << std::setw(maxStringLen + 1) << std::left << "Name Of File";
    if (!startDate.IsInvalid() && !endDate.IsInvalid())
        std::cout << " | Total Orders Value between " << startDate << " and " << endDate << '
    else if (!startDate.IsInvalid())
        std::cout << " | Total Orders Value since " << startDate << '\n';
    else
        std::cout << " | Total Orders Value\n";

    for (const auto& [fileName, sum] : results)
        std::cout << std::setw(maxStringLen + 1) << std::left << fileName << " | " << std::fi
    ScopeTimer::ShowStoredResults();
}

int main(int argc, const char** argv)
{
    if (argc <= 1)
    {
        std::cerr << "path (startDate) (endDate)\n";
        return 1;
    }

    try
    {
        const auto paths = CollectPaths(argv[1]);

        if (paths.empty())
        {
            std::cout << "No files to process...\n";
            return 0;
        }

        const auto startDate = argc > 2 ? Date(argv[2]) : Date();
        const auto endDate = argc > 3 ? Date(argv[3]) : Date();

        const auto results = CalcResults(paths, startDate, endDate);

        ShowResults(results, startDate, endDate);
    }
    catch (const fs::filesystem_error& err)
    {
        std::cerr << "filesystem error! " << err.what() << '\n';
    }
    catch (const std::runtime_error& err)
    {
        std::cerr << "runtime error! " << err.what() << '\n';
    }

    return 0;
}
```

In most of the cases, all we have to do to add parallel execution is to make sure there's no synchronisation required between the tasks and if we can provide forward iterators.

That's why when doing the conversion we sometimes needed to preallocate vector rather than using `std::back_inserter`.

Another example is that we cannot iterate in a directory in parallel, as `std::filesystem::directory_iterator` is not a forward iterator.

The next part is to select the proper parallel algorithm. In the case of this example, we replaced `std::accumulate` with `std::transform_reduce` for the calculations. There was no need to change `std::transform` for doing the string parsing - as you only have to use extra `execution policy` parameter.

Our application performed a bit better than the serial version. Here are some thoughts we might have:

- Parallel execution needs tasks that are independent. If you have jobs that depend on each other, the performance might be lower than the serial version! This happens due to extra synchronisation steps.
- Your tasks cannot be memory-bound, otherwise CPU will wait for the memory. For example, the string parsing code performed better in parallel as it has many instructions to execute: string search, string conversions.
- You need a lot of data to process and to see the performance gain. In our case, each file required several thousands of lines to show some gains over the sequential version.
- Sum calculations didn't show much improvement and even worse performance for smaller input. As the `std::reduce` algorithm requires extra reduction steps and also our calculations were elementary. Possibly, with more statistical computations in the code, we could bring more performance.
- The serial version of the code is straightforward and there are places where extra performance could be gained. For example, we might reduce additional copies and temporary vectors. It might also be good to use `std::transform_reduce` with sequential execution in the serial version. As it might be faster than `std::accumulate`. You might consider optimising the serial version first and then make it parallel.
- If you rely on exceptions then you might want to implement a handler for `std::terminate`, as exceptions are not re-thrown in code that is invoked with execution policies.

We can sum up the experiment with the following summary:

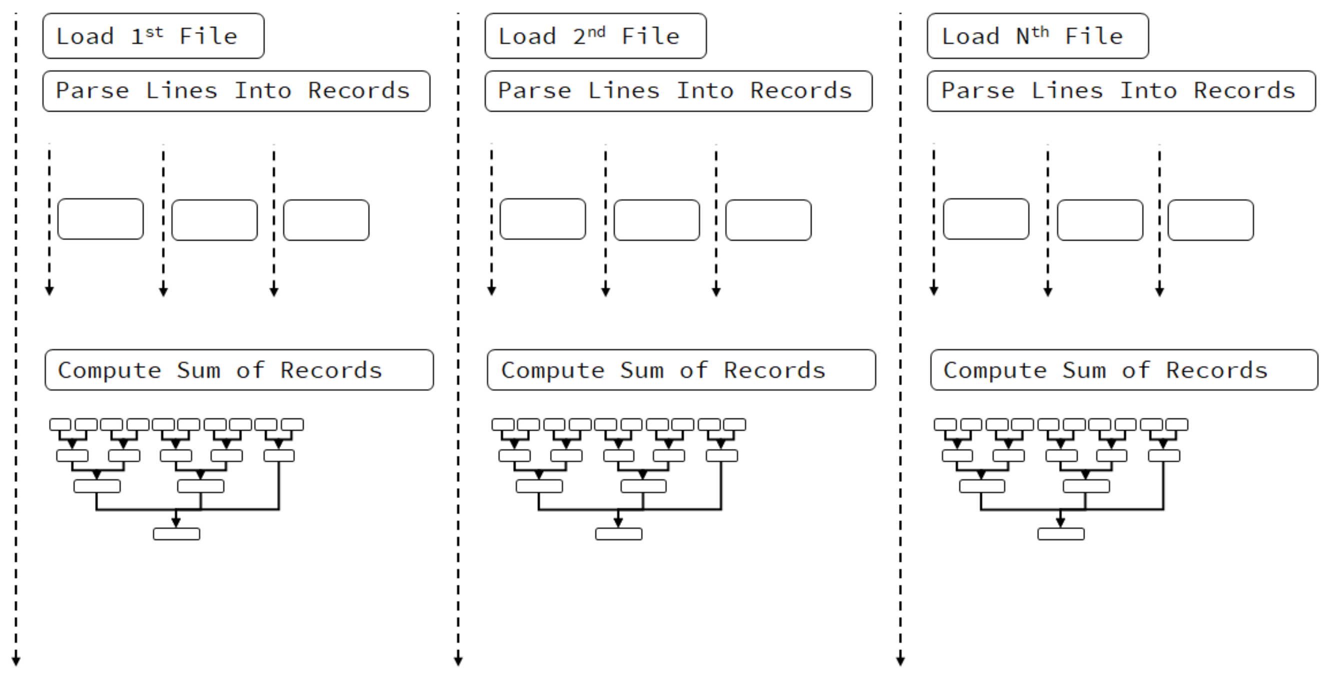
> *Parallel algorithms can bring extra speed to the application, but they have to be used wisely. They introduce an additional cost of parallel execution framework, and it's essential to have lots of tasks that are independent and good for parallelisation. As always, it's important to measure the performance between the versions to be able to select the final approach*

> *with confidence.*

Additional Modifications and Options

The code in the parallel version skipped one option: parallel access to files. So far we read files one by one, but how about reading separate files from separate threads?

Here’s a diagram that illustrates this option:



In the above diagram, the situation is a bit complicated. If we assume that OS cannot handle multiple file access, then threads will wait on files. But once the files are available, the processing might go in parallel.

If you want to play with this technique, you can replace `std::execution::seq` in `CalcResults()` with `std::execution::par`. That will allow compiler to run `LoadRecords()` and `CalcTotalOrder()` in parallel.

Can OS handle accessing files from separate threads?

In general, the answer might be tricky, as it depends on many elements: hardware, system, cost of computations, etc. For example on a machine with a fast SSD drive the system can handle several files reads, but on a HDD drive, the performance might be slower. Modern drives also use Native Command Queues, so even if you access from multiple threads the command to the drive will be serial and also rearranged into the optimal way. We leave the experiments to the readers as this topic goes beyond the range of this course.

That’s a wrap.