Count all distinct words from a big file

Dipl-Ing. Dr. techn. Florin Ioan Chertes

2021-11-27

1 General description of the problematic

The program counts the number of distinct unique words from a file whose name is passed as an argument to a program. The program should be able to handle large inputs e.g., 32 GB. The solution must utilize all available CPU resources. The program basically

- opens the file,
- reads all the words in a set and
- using the properties of the set, prints the size of the set as the result.

1.1 First implementations ideas

If the file has a moderate length two alternative solutions could be applied.

- If the file is line oriented, it could be read line by line and all the words of each line inserted the the set. The line length is assumed to be moderately long.
- If the file is not very big and not line oriented, then the whole file could be read as a whole in a big string and then inserted in the set.

1.2 Current concrete situation

The file to be read is not line oriented, so it could be not read line by line. The file is to big to be read in one string in the memory and used as such efficiently. There are two solutions that could work:

- read the big file in blocks of the a certain number of bytes or,
- use the proprietary methodology of memory mapped files in two different ways

  offered by different operation systems, or

  by using the well known and appreciated library boost.

2 Current implementation

Experts believe, that memory mapped file solution performs better than any other solution, but it is known, that a memory mapped solution is operation system (OS) dependent. This dependency to OS could be changed to a dependency to the well known and appreciated boost library. I prefer to delay this implementation using this dependencies to OS or to boost library as long as possible and give a good C++ solution, but not an optimal OS solution.

The current implementation considers the solution of reading blocks successively from a file in a string and then inserting them in a set. Many ideas were

taken from [1, 2]. The problem of reading blocks is considered solved. Without loss of generality, the big file was so constructed, that by reading in blocks of fix length words are not fragmented between two successive blocks. This problem could be solved by an algorithm, later implemented. For the proof of concept, a very simplified case was tested. In this case the first block is also the only one.

2.1 Change the length of the read file

To create files with a defined length, please use the the function:

```
void create_big_file(const std::filesystem::path& filePath)
```

In the loop

```
for (int i = 0; i < 1024 * 5000; ++i)
```

please use the value *5000* as a parameter. To test with files in the range of 1.5GB please replace this value with *50000*.

2.2 Read only one block, the implementation

In this very simplified case, in Listing 1 we have only one block, that is read only in one string and then is inserted in a set. The size of the set is the desired result.

Listing 1: count words from file

```
std::size_t count_words_from_string(std::string&& file_as_string)
{
    std::stringstream istring_stream(std::move(file_as_string));
    std::istream_iterator<std::string> it{ istring_stream };
    std::unordered_set<std::string, StringHash,
                       std::equal_to<>> uniques;

    std::transform(it,
                {},
                std::inserter(uniques, uniques.begin()),
                std::identity{});

    return uniques.size();
}
```

The implementation of the *struct StringHash* in this case is the most trivial to keep the things, as simple, as possible.

2.3 Read from file, the implementation

The previous implementation, in Listing 1 uses *std::string* as source for the *std::istream_iterator*. But a *std::istream_iterator* could be obtained also from a *std::filesystem::path*, referring to a valid file. So the file itself could be used

as input. The desired result, as in the previous case is the size of the set representing the number of distinct words in the file. In the Listing 2 bellow, this implementation is presented.

Listing 2: count words from file

```
 std::size_t count_words_from_file(
        const std::filesystem::path& filePath)
{
    std::ifstream ifile(filePath);
    std::istream_iterator<std::string> it{ ifile };
    std::unordered_set<std::string, StringHash,
                std::equal_to<>> uniques;

    std::transform(it,
                   {},
                   std::inserter(uniques, uniques.begin()),
                   std::identity{});

    return uniques.size();
}
```

Observation to the implementation This implementation looks very compact and easy to read. The performance of this implementation must be tested and compared with probably more complex procedural implementation variants.

2.4 Read using string blocks, the implementation

In this case, reading from a string, Listing 1 is reused. Reading from file is performed using blocks of limited length into strings. The string's contain is inserted in a set as in previous cases. The size of the set is the desired result.

Listing 3: count words from file

```
std::size_t count_words_from_file_read_in_blocks(
        const std::filesystem::path& filePath)
{
    // Buffer size 1/32 Megabyte
    constexpr std::size_t buffer_size = 1 << 15; // 20 is 1 Megabyte
    std::cout << "buffersize: " << buffer_size << std::endl;

    std::unordered_set<std::string,
                StringHash,
                std::equal_to<>> uniques;

    try {
```

```
        std::ifstream in_file{ filePath,
                               std::ios::in | std::ios::binary };
        if (!in_file)
            throw std::runtime_error("Cannot open "
                + filePath.filename().string());

        const auto fsize{
                static_cast<size_t>(std::filesystem::file_size(filePath)) };
        const auto loops{ fsize / buffer_size };
        const auto lastChunk{ fsize % buffer_size };
        auto insert_file_block_in_set =
                [&in_file, buffer_size, &uniques]()
        {
            std::string file_as_string(buffer_size, 0);
            in_file.read(file_as_string.data(), file_as_string.size());

            std::stringstream istring_stream(std::move(file_as_string));
            std::istream_iterator<std::string> it{ istring_stream };

            std::transform(it,
                           {},
                           std::inserter(uniques, uniques.begin()),
                           std::identity{});
        };

        for (std::size_t i = 0; i < loops; ++i)
        {
            insert_file_block_in_set();
        }

        if (lastChunk > 0)
        {
            insert_file_block_in_set();
        }
    }
    catch (const std::filesystem::filesystem_error& err) {
        std::cerr << "filesystem error! " << err.what() << '\n';
    }
    catch (const std::runtime_error& err) {
        std::cerr << "runtime error! " << err.what() << '\n';
    }

    return  uniques.size();
}
```

3 Source code

The source code relevant for these is placed in the file:
ex603_mcp_001/test/src/test_read_file.cpp
To read and test the source code please clone the repository in the work directory from:
https://github.com/FlorinChertes/ex603_mcp_001.git.
To create the executable *ex603_mcp_002_ex* and to run it, please use the *CMakeLists.txt* file. In the work directory input the following commands:

```
mkdir build
cd build
cp -R ../ex603_mcp_001/data/ ./data
cmake -DCMAKE_BUILD_TYPE=Release ../ex603_mcp_001
cmake --build .
./ex603_mcp_002_ex
```

4 Conclusions

The presented functions were tested with files in the range of 1.5 GB this is far more than my experience, 0.5 GB is already big enough . I know that the task spoke about 32 GB, this is in the moment above what I can simulate with my machines. I solved the problem using files in the range of 1.5 GB and they are read in some 15-20 seconds, in the best case. I do not know if this satisfies the requirements. I am using by intention an old processor (i5 2,5GHz 4GB RAM), a new processor (i7-6600U CPU 4 x 2.6 GHz 8GB RAM) but very new OS, i.e., Ubuntu 21 with gcc 11 and Debian 11 with gcc 10.2; with MSVC 16.7 on WIN 10 (i5-8350U CPU 4 x 1,7GHz 24GB RAM) the time is much longer, at a factor 2 to 3 in *Release*.

As far as I could measure the bottleneck is not the way I read from the disk, but the operation of inserting the strings in the set, or the *std::unordered_set* itself, or its configuration. I believe that the used methodologies: $std :: istream\_iterator < std :: string >$ and *std::transform* are at the root of the performance limitations, I encounter.

The work on this project in the direction of *memory mapped files* could be done in collaboration with one or two other colleges to be able to change ideas and experience. The used machines must be dedicated for the production, not as in my case, general usage hardware, above mentioned.

The measured time of read the file and insert in the set was almous similar in *Release* and *Debug* versions with *gcc* on *Linux* but different (factor 10) on *WIN10* with *MSCV 16.7* for this behavior on *Linux* with *gcc* I do not have an exhaustive explanation.

This case has potentially very many words, the files are very big, e.g., 32 GB, the necessary time to compute the result is not hundreds of milliseconds but about 300 seconds. This case is what Teodorescu [3] calls a case were parallel

algorithms, introduced with C++17 could play an important role. Further Teodorescu remarks that *concurrency* is a design decision but *parallelism* is a local implementation decision. The used *STL* algorithm: *std::transform* has a version employing the parallel behavior. This is very good presented in Williams [4] in Chapter 10. In the current implementation the usage of the parallel version of the algorithm was not possible to be used because the *std::istream_iterator* is a *input* iterator. For parallel behavior a *forward* iterator is necessary. This implies the usage of something different, not the *std::istream_iterator* but something very similar, this could be an interesting future task.

References

[1] "Bartlomiej filipek c++ stories." https://www.cppstories.com/.

[2] "Bartlomiej filipek how to parallelise csv reader - c++17 in practice." https://www.cppstories.com/2021/csvreader-cpp17/.

[3] L. R. Teodorescu, "A case against blind use of c++ parallel algorithms," *ACCU Overload*, vol. 161, pp. 4–7, 2021.

[4] A. Williams, *C++ Concurrency in Action*. Manning, 2019.