Count the distinct word from a 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 each line read in 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 all in a big string and then inserted in a 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 a 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 to implement 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 successively blocks from the file in a string then inserting them in the set. Many ideas were taken from [1]. The problem of reading strings blocks is considered solved. For the proof of concept, a very simplified case is tested. In this case the first block is also the only one.

2.1 Change the length of the read file

To test files with a different 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(const std::string& file_as_string)
{
    std::stringstream istring_stream(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 \textit{**struct** StringHash} in **this case** is the mot tri

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 limited blocks, as strings. The string's contain is inserted in a set as in previous case. 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 };
        std::string file_as_string(buffer_size, 0);
        auto insert_file_block_in_set =
                [&in_file, &file_as_string, &uniques]()
        {
            in_file.read(file_as_string.data(), file_as_string.size());

            std::stringstream istring_stream(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

To read and test the source code please use the file *ex_read_file.tar.bz2*. Copy this file to the work directory and input the following commands.

```
tar -xjvf ex_read_file.tar.bz2

mkdir build
```

```
cd build

cp -R ../ex_read_file/data/ ./data

cmake ../ex_read_file

make

./ex_read_file
```

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 2.6 GHz x 4 8GB RAM) but very new OS, i.e., Ubuntu 21 and gcc 11 and Debian 11 gcc 10.2, with MSVC 16.7 on WIN 10 (i5 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.

References

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