High Performance Computing (Master) in WS19

Exercise 1: Rember your C++; Efficient implementations on one core.

Deadline: Nov 4, 2019, 16:00

Status: 27. September 2019, 16:08

Supervisor: Prof.Dr. G. Haase, gundolf.haase@uni-graz.at

(A) Mean values: [2 pts]

We consider three different means, namely the arithmetic mean, geometric mean and the harmonic mean[1].

a) Start with code (intro_function[2]) and add a function that
   - receives three integers in the input parameter list,
   - evalulates the three means above, and
   - returns these three values via parameter list to the main function.

b) Call the function from your main function and print the results therein.

c) The input data $(1, 4, 16)$ results in the three means $(7, 4, 2.28571)$.

d) The input data $(2, 3, 5)$ results in the three means $(3.33333, 3.10723, 2.90323)$.

e) Check the correctness for $(1000, 4000, 16000)$, besides the limited accuracy of floating point numbers.

f) The same as a) but with one STL vector[3] of arbitrary length containing the input data.

Hints: `#include <cmath>`, `pow`, `#include <vector>`, `vector`

(B) Data-IO and vectors: [2 pts]

Read the data from ASCII file *data_1.txt*[4] into an STL-vector[5] and determine minumum, maximum, mean values and deviation[6] of your data. Write those values into an ASCII file *out_1.txt*.

- See the example[7] on file-IO and reimplement the relevant parts in your code.
- Use minimal data type (wrt. storage) for storing the vector elements.
- You may use the STL for the determination of minumum[8], maximum. Use your own function for arithmetic mean, geometric mean and harmonic mean as well as for the standard deviation.

Hints: `#include <cmath>`, `pow`, `#include <vector>`, `vector`, `#include <algorithm>`

[1] https://en.wikipedia.org/wiki/Mean
[2] http://imsc.uni-graz.at/haasegu/Lectures/Math2CPP/Codes/seq/skalar_stl.tar
[3] http://www.cplusplus.com/reference/vector/vector/
[4] http://imsc.uni-graz.at/haasegu/Lectures/Math2CPP/Examples/Data/data_1.txt
[5] http://www.cplusplus.com/reference/vector/vector/
[6] https://www.mathsisfun.com/data/standard-deviation.html
[7] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS18/file_io.zip
[8] http://www.cplusplus.com/reference/algorithm/min_element/

(C) Summation of specified numbers: [2 pts]

Write a function with input parameter n that adds all those positive integers less or equal n which are a multiples of 3 or of 5 (including or!).

- The easiest approach uses a for-loop.
- Test your function in the main function with various parameters:
  - $n = 15$ results in 60.
  - $n = 1001$ results in 234 168.
  - $n = 1432987$ results in 479 139 074 204.
- Derive a formula for calculating the required sum without executing a loop. Implement it in a second function and test it.
- Compare the run time of your two functions by using the `chrono`[9] functions for time measurement.
  Run each function at least 1000 times to get some measurable timings.

Hints: `cout`, `cin`, `endl`, `for`, `auto`, `std::chrono::high_resolution_clock::now()`, `std::chrono::duration<double>`, `std::chrono::duration_cast<...>(...)`

(D) Kahan summation: [3 pts]

Numerical computation by floating point numbers in the computer cause roundoff errors due to the limited precision available. Summing large and small numbers together might result in a non neglectable final error.

The Kahan summation[10] is one approach to compensate this error.

1. Start with the skalar product code (tar[11]). and extend it with a new function `Kahan_skalar` that performs the summation therein according to Kahan.
2. Calculate the sum
$$s_n := \sum_{k=1}^{n} \frac{1}{k^2}$$
for increasing n and compare the difference of the results from the two functions.
   - Use compiler option **-O1**, not option -O2 or higher for the `Kahan_skalar`.
3. We know that $s_n \to \frac{\pi^2}{6}$ for $n \to \infty$.
   Compare the two results with this value for $n \to \infty$.

Hints: `#include <cmath>`, `M_PI`

[9] `http://www.cplusplus.com/reference/chrono/high_resolution_clock/now/`
[10] `https://en.wikipedia.org/wiki/Kahan_summation_algorithm`
[11] `https://imsc.uni-graz.at/haasegu/Lectures/Math2CPP/Codes/seq/skalar_stl.tar`

(E) Vector versus list: [4 pts]

Assume a sorted container (vector[12]/list[13]) x of length n with ascending entries $x_k = k + 1$, $k = \overline{0, n-1}$.
Let us generate random numbers $\in [1, n]$ and place each new element into the container such that the enlarged container is still sorted.

Write functions for a vector and for a list that insert n random numbers successively into the given ordered container. The container is going to have length $2n$ finally.
Measure the time spent inside functions for various n.
Which container is faster and why!?

- Random numbers can be generated via the C++ random[14] library, see the example[15] code.
- A container can be sorted by `sort()`[16] which is not needed in this task.
- Finding the position to insert the new element into a sorted container can be done either
  - via algorithm `find_if()`[17] together with a lambda-function, or
  - by using algorithm `lower_bound()`[18] (or upper_bound()) which is simpler and faster.

  An iterator[19] is returned indicating the position in the container.
- Inserting a new element at an arbitrary position into a container can be done using the method `insert()`[20].
- Checking whether a container is sorted can be checked with algorithm `is_sorted()`[21]
- The timing can be performed either by `chrono`[22] functions, or via the old C-timing functions ctime[23].

Hints: `auto`, `std::chrono::high_resolution_clock::now()`, `std::chrono::duration<double>`, `std::chrono::duration_cast<...>(...)`. `clock()`,

[12] `http://www.cplusplus.com/reference/vector/vector/`
[13] `http://www.cplusplus.com/reference/list/list/`
[14] `http://www.cplusplus.com/reference/random/`
[15] `http://www.cplusplus.com/reference/random/linear_congruential_engine/operator()`
[16] `http://www.cplusplus.com/reference/algorithm/sort`
[17] `http://www.cplusplus.com/reference/algorithm/find_if/`
[18] `http://www.cplusplus.com/reference/algorithm/lower_bound/`
[19] `http://www.cplusplus.com/reference/iterator/`
[20] `http://www.cplusplus.com/reference/vector/vector/insert/`
[21] `http://www.cplusplus.com/reference/algorithm/is_sorted/?kw=is_sorted`
[22] `http://www.cplusplus.com/reference/chrono/high_resolution_clock/now/`
[23] `http://www.cplusplus.com/reference/ctime/clock/`

(F) Goldbach's conjecture: [4 pts]

Each even number larger than 3 can be written as sum of two primes (Goldbach's conjecture[24]), i.e., that holds also for all even numbers from $[4, n]$ ($n >= 4$).

1. Incorporate the header mayer_primes.h[25] into your code. The modified code originates from Florian Mayer[26] and generates all primes until n.
2. Write a function `single_goldbach(k)` that counts for a natural number k the number of possible decompositions with 2 primes for k and returns that number to your main code (e.g., $k = 694$ has 19 decompositions).
3. Write a function `count_goldbach(n)` that counts the number of possible decompositions for all even numbers in $[4, n]$ and returns these data.
   Determine in your main code that k with the most decompositions ($n = 100.000 \Longrightarrow k = 99.330$).
4. Measure the run time (see §13.2 im Script[27]) of your function `count_goldbach(n)` for $n = \{10.000 \quad 100.000 \quad 400.000 \quad 1.000.000 \quad 2.000.000 \quad (10.000.000)\}$.

* Write a function similar to the one in 3. but returning all decompositions for all even numbers in the given range.

C++ hints: `vector`, `max_element`, `push_back`

(G) Dense Matrices Access: [4 pts]

We will have a look at the effect of access patterns of dense matrices and of a special dense matrix structure on the run time of the matrix vector product.
We will use a 1D memory layout to store a $n \times n$ dense matrix. Start with example code intro_function_densematrix[28].

a) Write a class for the dense matrix that contains:
   - A constructor with n (#rows) and m (#columns) as input parameters.
     Initialize the matrix elements in this constructor via

     $$M_{i,j} = f(x_i) \cdot f(x_j)$$

     with $x_k = \frac{10k}{nm-1} - 5 \quad \forall k = 0, \ldots, nm-1$ with $nm = \max(n, m)$ and the Sigmoid[29] function $f(x) := (1 + \exp(-x))^{-1}$.
   - Implement a (const[30]!) Method `Mult` for multiplying this matrix with a vector passed as input parameter, returning the resulting vector to your main code. Use rowise access to the matrix elements.
   - Write a (const) Method `MultT` that multiplies the transosed matrix with a vector.
     Do not transpose the matrix. You only have to change the rowise access from the matrix above to a columnwise access.

[24] https://en.wikipedia.org/wiki/Goldbach's_conjecture
[25] https://imsc.uni-graz.at/haasegu/Lectures/Math2CPP/Examples/goldbach/mayer_primes.h
[26] http://code.activestate.com/recipes/576559-fast-prime-generator
[27] https://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf
[28] https://imsc.uni-graz.at/haasegu/Lectures/Math2CPP/Examples/intro_vector_densematrix.tar
[29] https://en.wikipedia.org/wiki/Sigmoid_function
[30] https://isocpp.org/wiki/faq/const-correctness#const-member-fns

b) Use your class and functions in the main function and check the results. Your main function should look like (plus the output of vectors `f1` and `f2`):

```
#include "mylib.h"
#include <iostream>
#include <cassert>
#include <vector>
using namespace std;

int main()
{
    DenseMatrix const M(5,3);      // Dense matrix, also initialized

    vector<double> const u{{1,2,3}};
    vector<double> f1 = M.Mult(u);

    vector<double> const v{{-1,2,-3,4,-5}};
    vector<double> f2 = M.MultT(v);

    return 0;
}
```

c) Contruct a dense square matrix with n rows/columns, choose $n \in [10^3, 10^4]$ depending on the amount of memory in your computer.
Measure the run time for `Mult` and for `MultT` with the same non-zero vector as input parameter. Explain the difference in run time!
Our dense square matrix is symmetric by construction (why?), therefore the two resulting vectors have to be equal. Check this!

```
//   code snippet
#include <ctime>
...
{
    ...
    int const NLOOPS=100;          // the overall code should run approx. 10 sec.
    ...

    double         t1 = clock();   // start timer
    vector<double> f1 = M.Mult(u);
    for (int k=1; k<NLOOPS; ++k)
    {
        f1 = M.Mult(u);
    }
    t1 = (clock()-t1)/CLOCKS_PER_SEC/NLOOPS;
    ...
}
```

d) Write another class for a dense matrix which is defined as $M = u^T * v$ with row vectors u and v.

- Implement the same functionality as above with methods `Mult` and `MultT` but taking advantage of the tensor product structure of the matrix.
- Initialize your matrix with $u = v = sigmoid(x_k)$ for $k = 0, \ldots, n$, i.e., the matrix will be symmetric.
- Perform the same run time tests and checks as above.

Hints: `#include <cmath>`, `exp`, `#include <vector>`, `vector`