```cpp
// clang-tidy *.cpp -checks=llvm-*,-llvm-header-guard -header-filter=.* -enable-check-profile -extra-arg="-std=c++17" -extra-arg="-fopenmp" -- *.cpp

#include "task_2.h"
#include "task_3.h"
#include "task_4.h"
#include "timing.h"

#include <algorithm>
#include <cassert>
#include <cmath>
#include <execution>
#include <iomanip>
#include <iostream>
#include <omp.h>
#include <sstream>
#include <vector>

void task_2() {
    printf("\n\n-------------- Task 2 --------------\n\n");

    int threads = 4;
    omp_set_num_threads(threads);
    cout << omp_get_max_threads() << " threads have been started." << endl;

    // Read vector
    vector<double> a;
    read_vector_from_file("data_1.txt", a);

    tic();
    // min and max
        // auto [min, max] = min_max_par(a);
        auto min = *min_element(std::execution::par, a.begin(), a.end());
        auto max = *max_element(std::execution::par, a.begin(), a.end());
    // means
        auto [x,y,z] = means_par(a);
    // deviation
        double deviation(0.0);
        #pragma omp parallel for shared(x,a) reduction(+:deviation)
        for (long unsigned int i=0; i<a.size(); i++){
            deviation += pow(x - a.at(i),2);
        }
        deviation = sqrt(deviation/static_cast<double>(a.size()));
    double t = toc();

    printf("Minimum: %f\n", min);
    printf("Maximum: %f\n", max);
    printf("Arithmetic: %f\n", x);
    printf("Geometric: %f\n", y);
    printf("Harmonic: %f\n", z);
    printf("Deviation: %f\n", deviation);
    printf("Execution time: %f\n", t);

    // write results to file
    vector<double> b = {min,max,x,y,z,deviation};
    write_vector_to_file("out_1.txt", b);
}

void task_3() {
    printf("\n\n-------------- Task 3 --------------\n\n");

    int threads = 4;
    omp_set_num_threads(threads);
    cout << omp_get_max_threads() << " threads have been started." << endl;

    // ####################################
    // single_goldbach(k)
    int k = 694;
```

```
    printf("single_goldbach(k = %d) = %d\n", k, single_goldbach_par(k));

    // Prints decompositions
    print_decomps(k);

    // count_goldbach(n)
    // printf("\nNOTE: For n=2'000'000 it will take ~30 seconds.\n");
    for (int n : {10'000, 100'000, 400'000, 1'000'000, 2'000'000/*, 10'000'000*/}) {
        tic();
        vector<int> counts = count_goldbach_par(n);
        double sec = toc();

        auto max = max_element(counts.begin(), counts.end());
        printf("count_goldbach(n = %d): k = %ld, decompositions = %d, time elapsed:
%f milliseconds\n", n, max-counts.begin(), *max, sec*1000);
    }
    printf("Should be:                k = 9240, 99330, 390390, 990990, 1981980, 96996
90\n");
    printf("            decompositions =  329,  2168,   7094,  15594,   27988,  1241
80\n\n");
}

void task_4() {
    printf("\n\n-------------- Task 4 --------------\n\n");

    int threads = 32;
    omp_set_num_threads(threads);
    cout << omp_get_max_threads() << " threads have been started." << endl;

    size_t M, N, L, p, NLOOPS;

    { //      Matrix-Vector product
    printf("----- Benchmark (B) -----\n");
    // Initialization
        M = 8'000;
        N = 12'000;
        NLOOPS = 30;
        auto [A,x] = init_B(M,N);
    // Benchmark
        tic();
        benchmark_B(A, x, NLOOPS, false);
        double sec = toc() / NLOOPS;
    // Timings and Performance
        size_t memory = M*N + M + N;
        size_t flops  = 2 * M * N;
        print_performance(sec, memory, flops, sizeof(A[0]));
    printf("-------------------------\n");
    }

    { //     Matrix-Matrix product
    printf("----- Benchmark (C) -----\n");
    // Initialization
        M = 1'000;
        N = 2'000;
        L = 500;
        NLOOPS = 20;
        auto [A,B] = init_C(M,N,L);
    // Benchmark
        tic();
        benchmark_C(A, B, L, NLOOPS, false);
        double sec = toc() / NLOOPS;
    // Timings and Performance
        size_t memory = M*L + L*N + M*N;
        size_t flops  = M * 2*L * N;
        print_performance(sec, memory, flops, sizeof(A[0]));
    printf("-------------------------\n");
    }

```

```
133:     { //      Polynomial evaluation
134:     printf("----- Benchmark (D) -----\n");
135:     // Initialization
136:         N = 1'000'000;
137:         p = 200;
138:         NLOOPS = 20;
139:         auto [x,a] = init_D(N,p);
140:     // Benchmark
141:         tic();
142:         benchmark_D(x, a, NLOOPS);
143:         double sec = toc() / NLOOPS;
144:     // Timings  and Performance
145:         size_t memory = 2.0 * N;
146:         size_t flops  = 2.0 * N * p;
147:         print_performance(sec, memory, flops, sizeof(x[0]));
148:     printf("-------------------------\n");
149:     }
150:
151:
152:     // Timing
153:     NLOOPS = 50;
154:     int K=9, T=16;
155:     vector<double> speedup_sum((K-3+1)*T), speedup_scalar((K-3+1)*T);
156:     for (int k=0; k<(K-3+1); ++k) {
157:         N = pow(10,k);
158:         auto [x,y] = init_A(N);
159:         for (int t=0; t<T; t++) {
160:             omp_set_num_threads(t+1);
161:
162:             tic();
163:             benchmark_summation(x, NLOOPS);
164:             speedup_sum[k*T+t] = toc() / NLOOPS;
165:
166:             tic();
167:             benchmark_A(x, y, NLOOPS, false);
168:             speedup_scalar[k*T+t] = toc() / NLOOPS;
169:         }
170:     }
171:
172:     // Calculating speedup
173:     for (int k=0; k<(K-3+1); ++k) {
174:         double t0 = speedup_sum[k*T];
175:         double t00 = speedup_scalar[k*T];
176:         for (int t=0; t<T; t++){
177:             speedup_sum[k*T+t] = t0/speedup_sum[k*T+t];
178:             speedup_scalar[k*T+t] = t00/speedup_scalar[k*T+t];
179:         }
180:     }
181:
182:     // Printing tables
183:     cout << fixed << setprecision(4);
184:     cout << "\n\nSpeedup: summation" << endl;
185:     cout << "k \\ threads |  ";
186:     for (int t=0; t<T; t++) {cout << setw(2) << t+1 << "  |  ";}
187:     cout << endl;
188:     for (int k=3; k<K+1; ++k) {
189:         cout << "     " << k << "      |";
190:         for (int t=0; t<T; t++) {
191:             cout << speedup_sum[(k-3)*T+t] << "|";
192:         }
193:     cout << endl;
194:     }
195:
196:     cout << "\n\nSpeedup: scalar" << endl;
197:     cout << "k \\ threads |  ";
198:     for (int t=0; t<T; t++) {cout << setw(2) << t+1 << "  |  ";}
199:     cout << endl;
200:     for (int k=3; k<K+1; ++k) {
```

```
201:         cout << "      " << k << "       |";
202:         for (int t=0; t<T; t++) {
203:             cout << speedup_scalar[(k-3)*T+t] << "|";
204:         }
205:     cout << endl;
206:     }
207: }
208:
209: int main() {
210:     task_2();
211:     task_3();
212:     task_4();
213:
214:     return 0;
215: }
```

```
    1: #pragma once
    2:
    3: #include <cstring> //memset
    4: #include <vector>
    5: //using namespace std;
    6:
    7: /** \brief Determines all prime numbers in interval [2, @p max].
    8:  *
    9:  *  The sieve of Eratosthenes is used.
   10:  *
   11:  *  The implementation originates from <a href="http://code.activestate.com/recipes/
576559-fast-prime-generator/">Florian Mayer</a>.
   12:  *
   13:  * \param[in]   max end of interval for the prime number search.
   14:  * \return  vector of prime numbers @f$2,3,5, ..., p<=max @f$.
   15:  *
   16:  * \copyright
   17:  * Copyright (c) 2008 Florian Mayer          (adapted by Gundolf Haase 2018)
   18:  *
   19:  * Permission is hereby granted, free of charge, to any person obtaining a copy
   20:  * of this software and associated documentation files (the "Software"), to deal
   21:  * in the Software without restriction, including without limitation the rights
   22:  * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
   23:  * copies of the Software, and to permit persons to whom the Software is
   24:  * furnished to do so, subject to the following conditions:
   25:  *
   26:  * The above copyright notice and this permission notice shall be included in
   27:  * all copies or substantial portions of the Software.
   28:  *
   29:  *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLI
ED,
   30:  * INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   31:  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
   32:  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
   33:  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
   34:  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOF
TWARE.
   35:  *
   36:  */
   37: template <class T>
   38: std::vector<T> get_primes(T max)
   39: {
   40:     std::vector<T> primes;
   41:     char *sieve;
   42:     sieve = new char[max / 8 + 1];
   43:     // Fill sieve with 1
   44:     memset(sieve, 0xFF, (max / 8 + 1) * sizeof(char));
   45:     for (T x = 2; x <= max; x++)
   46:     {
   47:         if (sieve[x / 8] & (0x01 << (x % 8))) {
   48:             primes.push_back(x);
   49:             // Is prime. Mark multiplicates.
   50:             for (T j = 2 * x; j <= max; j += x)
   51:             {
   52:                 sieve[j / 8] &= ~(0x01 << (j % 8));
   53:                         }
   54:         }
   55:         }
   56:     delete[] sieve;
   57:     return primes;
   58: }
   59:
   60: //----------------------------------------------------------------
   61: //int main()        // by  Florian Mayer
   62: //{g++ -O3  -std=c++14 -fopenmp main.cpp && ./a.out
   63: //    vector<unsigned long> primes;
   64: //    primes = get_primes(10000000);
   65: //    // return 0;
```

```
66: //    // Print out result.
67: //    vector<unsigned long>::iterator it;
68: //    for(it=primes.begin(); it < primes.end(); it++)
69: //        cout << *it << " ";
70: //
71: //    cout << endl;
72: //    return 0;
73: //}
74:
```

```cpp
#include "task_2.h"
#include <cassert>          // assert
#include <cmath>
#include <fstream>
#include <omp.h>

tuple<double, double> min_max_par(const vector<double> &v) {
    int min_val = v[0];
    int max_val = v[0];

    #pragma omp parallel for reduction(min:min_val) reduction(max:max_val)
    for (size_t i = 0; i < v.size(); ++i) {
        if (v[i] < min_val) min_val = v[i];
        if (v[i] > max_val) max_val = v[i];
    }

    return make_tuple(min_val,max_val);
}


tuple<double, double, double> means_par(const vector<double>& v){
    size_t n = v.size();
    double sum = 0;
    double logsum = 0;
    double invsum = 0;

    #pragma omp parallel for shared(v,n) reduction(+:sum, logsum, invsum)
    for (size_t i = 0; i<n; ++i){
        sum += v[i];
        logsum += log(v[i]);
        invsum += 1.0/v[i];
    }

    double arith = sum / static_cast<double>(n);
    double geo = exp(1.0/static_cast<double>(n) * logsum);
    double harm = static_cast<double>(n) / invsum;
    return make_tuple(arith, geo, harm);
}

void fill_vector(istream& istr, vector<double>& v)
{
    double d=0;
    while ( istr >> d) v.push_back(d); // Einlesen
    if (!istr.eof())
    { // Fehlerbehandlung
        cout << " Error handling \n";
        if ( istr.bad() )  throw runtime_error("Schwerer Fehler in istr");
        if ( istr.fail() )   // Versuch des Aufraeumens
        {
            cout << " Failed in reading all data.\n";
            istr.clear();
        }
    }
    v.shrink_to_fit();                  // C++11
    return;
}


void read_vector_from_file(const string& file_name, vector<double>& v)
{
    ifstream fin(file_name);            // Oeffne das File im ASCII-Modus
    if( fin.is_open() )                 // File gefunden:
    {
       v.clear();                       // Vektor leeren
       fill_vector(fin, v);
    }
    else                                // File nicht gefunden:
    {
```

```
   69:          cout << "\nFile " << file_name << " has not been found.\n\n" ;
   70:          assert( fin.is_open() && "File not found." );       // exeption handling for
 the poor programmer
   71:      }
   72:
   73:   return;
   74: }
   75:
   76: void write_vector_to_file(const string& file_name, const vector<double>& v)
   77: {
   78:      ofstream fout(file_name);          // Oeffne das File im ASCII-Modus
   79:      if( fout.is_open() )
   80:      {
   81:         for (size_t k=0; k<v.size(); ++k)
   82:         {
   83:            fout << v.at(k) << endl;
   84:         }
   85:      }
   86:      else
   87:      {
   88:          cout << "\nFile " << file_name << " has not been opened.\n\n" ;
   89:          assert( fout.is_open() && "File not opened."  );         // exeption handlin
g for the poor programmer
   90:      }
   91:
   92:   return;
   93: }
```

```
    1: #pragma once
    2: #include <iostream>
    3: #include <vector>
    4: using namespace std;
    5:
    6: tuple<double, double> min_max_par(const vector<double> &v);
    7: tuple<double, double, double> means_par(const vector<double>& v);
    8:
    9: /**
   10:   This function opens the ASCII-file named @p file_name and reads the
   11:   double data into the C++ vector @p v.
   12:   If the file  @p file_name does not exist then the code stops with an appropriate m
essage.
   13:   @param[in]    file_name   name of the ASCII-file
   14:   @param[out]   v           C++ vector with double values
   15: */
   16:
   17: void read_vector_from_file(const string& file_name, vector<double>& v);
   18:
   19:
   20: /**
   21:   This function opens the ASCII-file named @p file_name and rewrites its with the
   22:   double data from the C++ vector @p v.
   23:   If there are problems in opening/generating file  @p file_name
   24:   then the code stops with an appropriate message.
   25:   @param[in]    file_name   name of the ASCII-file
   26:   @param[in]    v           C++ vector with double values
   27: */
   28:
   29: void write_vector_to_file(const string& file_name, const vector<double>& v);
   30:
   31: /**
   32:   Fills the double-vector @p v with data from an input stream @p istr until this inp
ut stream
   33:   ends regularily. The vector is cleared and its memory is automatically allocated.
   34:   @param[in]    istr        input stream
   35:   @param[out]    v           C++ vector with double values
   36:   @warning     An exception is thrown in case of wrong data format or corrupted data
.
   37: */
   38: void fill_vector(istream& istr, vector<double>& v);
```

```cpp
#include "task_3.h"
#include "mayer_primes.h"
#include "timing.h"

#include <algorithm>
#include <cassert>
#include <iostream>
#include <omp.h>
#include <vector>
using namespace std;


int single_goldbach_par(int k) {
    const vector<int> primes = get_primes(k);
    int count = 0;

    #pragma omp parallel for reduction(+:count)
    for (size_t i = 0; i < primes.size(); i++) {
        for (size_t j = i; j < primes.size(); j++) {
            if (primes[i] + primes[j] == k) {
                count++;
            }
        }
    }

    return count;
}


vector<int> count_goldbach_par(int n) {
    const vector<int> primes = get_primes(n);
    vector<int> counts(n+1);

    #pragma omp parallel reduction(VecAdd:counts)
    // #pragma omp parallel
    {
        vector<int> local_counts(n+1, 0);

            #pragma omp for
        for (size_t i = 1; i < primes.size(); i++) {
            for (size_t j = i; j < primes.size(); j++) {
                int sum = primes[i] + primes[j];
                if (sum <= n) {
                    local_counts[sum]++;
                }
            }
        }

        counts += local_counts;
        // #pragma omp critical
        // {
        //     for(int k=0; k<n+1; k++){
        //     counts[k] += local_counts[k];
        //     }
        // }
    }
    return counts;
}


void print_decomps(int k) {
    const vector<int> primes = get_primes(k);
    cout << "\nDecompositions for k = " << k << ": ";

    for (size_t i = 0; i < primes.size(); i++) {
        for (size_t j = i; j < primes.size(); j++) {
            if (primes[i] + primes[j] == k) {
                cout << primes[i] << " + " << primes[j] << ", ";
```

nicht initialisierter Speicher bei Thread 0 (und bei allen weiteren Threads) ✓ sehr gut

```
   69:             }
   70:         }
   71:     }
   72:     cout << endl;
   73: }
```

```
    1: #pragma once
    2: #include <cassert>
    3: #include <vector>
    4: using namespace std;
    5:
    6:
    7: // Counts number of possible decompositions with 2 primes that sum up to k.
    8: int single_goldbach_par(int k);
    9:
   10: // Counts number of possible decompositions with 2 primes that sum up to k for all e
ven numbers k \in {4,...,n}.
   11: vector<int> count_goldbach_par(int n);
   12:
   13: // Prints all decompositions of k.
   14: void print_decomps(int k);
   15:
   16:
   17: /**      Vector @p b adds its elements to vector @p a .
   18:         @param[in] a    vector
   19:         @param[in] b    vector
   20:         @return              a+=b componentwise
   21: */
   22: template<class T>
   23: std::vector<T> &operator+=(std::vector<T> &a, std::vector<T> const &b)
   24: {
   25:     assert(a.size()==b.size());
   26:     for (size_t k = 0; k < a.size(); ++k) {
   27:         a[k] += b[k];
   28:     }
   29:     return a;
   30: }
   31:
   32: #pragma omp declare reduction(VecAdd : std::vector<int>  : omp_out += omp_in) \
   33:  initializer (omp_priv=omp_orig)
```

```cpp
#include "task_4.h"
#include "timing.h"
#include <cassert>
#include <cblas.h>                // cBLAS Library
#include <iostream>
#include <vector>
using namespace std;

vector<double> matrix_vec(vector<double> const &A, vector<double> const &x) {
    size_t const N = x.size();
    size_t const M = A.size() / N;
    vector<double> b(M);

    #pragma omp parallel for shared(A,x,N,M,b)
    for (size_t i = 0; i < M; ++i) {
        for (size_t j = 0; j < N; ++j) {
            b[i] += A[i*N + j] * x[j];
        }
    }
    return b;
}


vector<double> matrix_matrix(vector<double> const &A, vector<double> const &B, size_t const &M) {
    size_t const L = A.size() / M;
    size_t const N = B.size() / L;
    vector<double> C(M*N,0);

    #pragma omp parallel for shared(A,B,M,L,N,C)
    for (size_t i = 0; i < M; ++i) {
        for (size_t k = 0; k < L; ++k) {
            for (size_t j = 0; j < N; ++j) {
                C[i*N + j] += A[i*L + k] * B[k*N + j];
            }
        }
    }
    return C;
}


vector<double> poly(vector<double> const &x, vector<double> const &a) {
    size_t N = x.size();
    size_t p = a.size();
    vector<double> y(N);

    #pragma omp parallel for shared(x,a,N,p,y)
    for (size_t i = 0; i < N; ++i) {
        y[i] = a[p];
        for (size_t k = 1; k < p; ++k) {
            y[i] = y[i]*x[i] + a[p-k];
        }
    }
    return y;
}

double scalar(vector<double> const &x, vector<double> const &y) {
    assert(x.size() == y.size());
    size_t const N = x.size();
    double sum = 0.0;

    #pragma omp parallel for shared(x,y,N) reduction(+:sum)
    for (size_t i = 0; i < N; ++i) {
        sum += x[i] * y[i];
    }
    return sum;
}

```

Try: collapse(2)

```
   68: double summation(vector<double> const &x){
   69:     size_t N = x.size();
   70:     double sum = 0.0;
   71:
   72:     #pragma omp parallel for shared(x,N) reduction(+:sum)
   73:     for (size_t i = 0; i < N; ++i) {
   74:         sum += x[i];
   75:     }
   76:
   77:     return sum;
   78: }
   79:
   80:
   81:
   82: // ########################################################################
   83:
   84:
   85: void print_performance(double sec, size_t memory, size_t flops, unsigned int size) {
   86:     printf("Memory allocated  : %.3f GByte\n", 1.0 * memory / 1024 / 1024 / 1024 * s
ize);
   87:     printf("Duration per loop : %.3f sec\n", sec);
   88:     printf("GFLOPS            : %.3f\n", 1.0 * flops / sec / 1024 / 1024 / 1024);
   89:     printf("GiByte/s          : %.3f\n", 1.0 * memory / sec / 1024 / 1024 / 1024 * s
ize);
   90: }
   91:
   92: tuple<vector<double>, vector<double>> init_A(size_t N) {
   93:     vector<double> x(N), y(N);
   94:     for (size_t i = 0; i < N; ++i) {
   95:         x[i] = i%219 + 1.0;
   96:         y[i] = 1.0 / x[i];
   97:     }
   98:     return make_tuple(x, y);
   99: }
  100:
  101: void benchmark_A(vector<double> const &x, vector<double> const &y, size_t NLOOPS, bo
ol cblas) {
  102:     size_t N = x.size();
  103:
  104:     double s(0.0), sum(0.0);
  105:     if (cblas == false) {
  106:         for (size_t i = 0; i < NLOOPS; ++i) {
  107:             s = scalar(x, y);
  108:             sum += s;
  109:         }
  110:     } else if (cblas == true) {
  111:         for (size_t i = 0; i < NLOOPS; ++i) {
  112:             s = cblas_ddot(N, x.data(), 1, y.data(), 1);
  113:             sum += s;
  114:         }
  115:     }
  116:
  117:     // Check correctness
  118:     if (static_cast<size_t>(sum) != N*NLOOPS) {printf("  !!  W R O N G  result  !!
\n");}
  119: }
  120:
  121: tuple<vector<double>, vector<double>> init_B(size_t M, size_t N) {
  122:     vector<double> A(M*N), x(N);
  123:     for (size_t i = 0; i < M; ++i) {
  124:         for (size_t j = 0; j < N; ++j) {
  125:             A[i*N + j] = (i+j)%219 + 1.0;
  126:         }
  127:     }
  128:     for (size_t j = 0; j < N; ++j) {
  129:         x[j] = 1.0/A[17*N + j];
  130:     }
  131:     return make_tuple(A, x);
```

```
  132: }
  133:
  134: void benchmark_B(vector<double> const &A, vector<double> const &x, size_t NLOOPS, bo
ol cblas) {
  135:     size_t N = x.size();
  136:     size_t M = A.size() / N;
  137:     vector<double> b(M);
  138:     double sum(0.0);
  139:
  140:     if (cblas == false) {
  141:         for (size_t i = 0; i < NLOOPS; ++i) {
  142:             b = matrix_vec(A,x);
  143:             sum += b[17];
  144:         }
  145:     } else if (cblas == true) {
  146:         for (size_t i = 0; i < NLOOPS; ++i) {
  147:             cblas_dgemv(CblasRowMajor, CblasNoTrans, M, N, 1.0, A.data(), N, x.data(
), 1, 0, b.data(), 1);
  148:             sum += b[17];
  149:         }
  150:     }
  151:
  152:     // Check correctness
  153:     if (static_cast<size_t>(sum) != N*NLOOPS) {printf("  !!   W R O N G  result   !!
\n");}
  154: }
  155:
  156: tuple<vector<double>, vector<double>> init_C(size_t M, size_t N, size_t L) {
  157:     vector<double> A(M*L), B(L*N);
  158:     for (size_t i = 0; i < M; ++i) {
  159:         for (size_t j = 0; j < L; ++j) {
  160:             A[i*L + j] = (i+j)%219 + 1.0;
  161:         }
  162:     }
  163:     // B chosen such that C[0,17]=L
  164:     // so B[i,17] = 1/A[0,i]
  165:     for (size_t i = 0; i < L; ++i) {
  166:         for (size_t j = 0; j < N; ++j) {
  167:             if (j==17) {
  168:                 B[i*N + 17] = 1.0/A[i];
  169:             } else {
  170:                 B[i*N + j] = (i+j)%219 + 1.0;
  171:             }
  172:         }
  173:     }
  174:     return make_tuple(A, B);
  175: }
  176:
  177: void benchmark_C(vector<double> const &A, vector<double> const &B, size_t L, size_t
NLOOPS, bool cblas) {
  178:     size_t M = A.size() / L;
  179:     size_t N = B.size() / L;
  180:     vector<double> C(M*N);
  181:     double sum(0.0);
  182:
  183:     if (cblas == false) {
  184:         for (size_t i = 0; i < NLOOPS; ++i) {
  185:             C = matrix_matrix(A,B,M);
  186:             sum += C[17];
  187:         }
  188:     } else if (cblas == true) {
  189:         for (size_t i = 0; i < NLOOPS; ++i) {
  190:             cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, L, 1.0, A.d
ata(), L, B.data(), N, 0.0, C.data(), N);
  191:             sum += C[17];
  192:         }
  193:     }
  194:
```

```
  195:     // Check correctness
  196:     if (static_cast<size_t>(sum) != L*NLOOPS) {printf("  !!   W R O N G  result   !!
\n");}
  197: }
  198:
  199: tuple<vector<double>, vector<double>> init_D(size_t N, size_t p) {
  200:     // x_i = i/N for i=0,...,N-1
  201:     // a_j = 1   for j=0,...,p-1
  202:     vector<double> x(N), a(p);
  203:     for (size_t i = 0; i < N; ++i) {
  204:         x[i] = static_cast<double>(i) / N;
  205:     }
  206:     for (size_t j = 0; j < p; ++j) {
  207:         a[j] = 1.0;
  208:     }
  209:     return make_tuple(x, a);
  210: }
  211:
  212: void benchmark_D(vector<double> const &x, vector<double> const &a, size_t NLOOPS) {
  213:     size_t N = x.size();
  214:     vector<double> y(N);
  215:     double sum(0.0);
  216:
  217:     for (size_t i = 0; i < NLOOPS; ++i) {
  218:         y = poly(x,a);
  219:         sum += y[0];
  220:     }
  221:
  222:     // Check correctness
  223:     if (static_cast<size_t>(sum) != NLOOPS) {printf("  !!   W R O N G  result sum =
%f  !!\n", sum);}
  224: }
  225:
  226: void benchmark_summation(vector<double> const &x, size_t NLOOPS) {
  227:     double s(0.0), sum(0.0);
  228:     for (size_t i = 0; i < NLOOPS; ++i) {
  229:         s = summation(x);
  230:         sum += s;
  231:     }
  232: }
```

```
    1: #pragma once
    2: #include <vector>
    3: using namespace std;
    4:
    5: vector<double> matrix_vec(vector<double> const &A, vector<double> const &x);
    6: vector<double> matrix_matrix(vector<double> const &A, vector<double> const &B, size_
t const &M);
    7: vector<double> poly(vector<double> const &x, vector<double> const &a);
    8: double scalar(vector<double> const &x, vector<double> const &y);
    9: double summation(vector<double> const &x);
   10:
   11: void print_performance(double sec, size_t memory, size_t flops, unsigned int size);
   12: tuple<vector<double>, vector<double>> init_A(size_t N);
   13: tuple<vector<double>, vector<double>> init_B(size_t M, size_t N);
   14: tuple<vector<double>, vector<double>> init_C(size_t M, size_t N, size_t L);
   15: tuple<vector<double>, vector<double>> init_D(size_t N, size_t p);
   16:
   17:
   18: void benchmark_A(vector<double> const &x, vector<double> const &y, size_t NLOOPS, bo
ol cblas);
   19: void benchmark_B(vector<double> const &A, vector<double> const &x, size_t NLOOPS, bo
ol cblas);
   20: void benchmark_C(vector<double> const &A, vector<double> const &B, size_t L, size_t
NLOOPS, bool cblas);
   21: void benchmark_D(vector<double> const &x, vector<double> const &a, size_t NLOOPS);
   22: void benchmark_summation(vector<double> const &x, size_t NLOOPS);
```

```
    1: //
    2: //    Gundolf Haase, Oct 18 2024
    3: //
    4: #pragma once
    5: #include <chrono>                   // timing
    6: #include <stack>
    7:
    8: //using Clock = std::chrono::system_clock;   //!< The wall clock timer chosen
    9: using Clock = std::chrono::high_resolution_clock;
   10: using TPoint= std::chrono::time_point<Clock>;
   11:
   12: // [Galowicz, C++17 STL Cookbook, p. 29]
   13: inline
   14: std::stack<TPoint> MyStopWatch; //!< starting time of stopwatch
   15:
   16: /** Starts stopwatch timer.
   17:  *  Use as @code tic(); myfunction(...) ; double tsec = toc();  @endcode
   18:  *
   19:  *  The timining can be nested and the recent time point is stored on top of the sta
ck.
   20:  *
   21:  *  @return recent time point
   22:  *  @see toc
   23:  */
   24: inline auto tic()
   25: {
   26:     MyStopWatch.push(Clock::now());
   27:     return MyStopWatch.top();
   28: }
   29:
   30: /** Returns the elapsed time from stopwatch.
   31:  *
   32:  * The time point from top of the stack is used
   33:  * if time point @p t_b is not passed as input parameter.
   34:  * Use as @code tic(); myfunction(...) ; double tsec = toc();  @endcode
   35:  * or as @code auto t_b = tic(); myfunction(...) ; double tsec = toc(t_b);  @endcode
   36:  * The last option is to be used in the case of
   37:  * non-nested but overlapping time measurements.
   38:  *
   39:  * @param[in]  t_b start time of some stop watch
   40:  * @return elapsed time in seconds.
   41:  *
   42: */
   43: inline double toc(TPoint const &t_b = MyStopWatch.top())
   44: {
   45:     // https://en.cppreference.com/w/cpp/chrono/treat_as_floating_point
   46:     using Unit       = std::chrono::seconds;
   47:     using FpSeconds = std::chrono::duration<double, Unit::period>;
   48:     auto t_e = Clock::now();
   49:     MyStopWatch.pop();
   50:     return FpSeconds(t_e-t_b).count();
   51: }
```