```cpp
#include "mylib.h"
#include <fstream>
#include <iostream>
#include <omp.h>
#include <vector>
using namespace std;


int main()
{
    // read vector from file
    vector<size_t> data_vector = {};

    ifstream input_stream("data_1.txt");

    size_t line;
    while(input_stream >> line)
    {
        data_vector.push_back(line);
    }
    data_vector.shrink_to_fit();




    // specify loops
    size_t NLOOPS = 10000;



    // ############# Parallelization with openMP #############
    // calculate arithmetic mean, geometric mean and harmonic mean
    double am_omp, gm_omp, hm_omp;

    double tstart = omp_get_wtime();

    for (size_t i = 0; i < NLOOPS; ++i)
        means_omp(data_vector, am_omp, gm_omp, hm_omp);

    double t_means_omp = (omp_get_wtime() - tstart)/NLOOPS;

    // calculate minimum and maximum
    size_t min, max;

    tstart = omp_get_wtime();

    for (size_t i = 0; i < NLOOPS; ++i)
        minmax_omp(data_vector, min, max);

    double t_minmax_omp = (omp_get_wtime() - tstart)/NLOOPS;







    // ############# Parallelization with C++ algorithms #############
    // calculate arithmetic mean, geometric mean and harmonic mean
    double am_cpp, gm_cpp, hm_cpp;

    tstart = omp_get_wtime();

    for (size_t i = 0; i < NLOOPS; ++i)
        means_cpp(data_vector, am_cpp, gm_cpp, hm_cpp);

    double t_means_cpp = (omp_get_wtime() - tstart)/NLOOPS;

    // calculate minimum and maximum
```

```
69:     size_t min_cpp, max_cpp;
70:
71:     tstart = omp_get_wtime();
72:
73:     for (size_t i = 0; i < NLOOPS; ++i)
74:         minmax_cpp(data_vector, min_cpp, max_cpp);
75:
76:     double t_minmax_cpp = (omp_get_wtime() - tstart)/NLOOPS;
77:
78:
79:
80:
81:
82:     // print results
83:     cout << "####### OpenMP #######" << endl;
84:     cout << "minimum: " << min << endl;
85:     cout << "maximum: " << max << endl;
86:     cout << "duration: " << t_minmax_omp << endl << endl;
87:
88:     cout << "arithmetic mean: " << am_omp << endl;
89:     cout << "geometric mean: " << gm_omp << endl;
90:     cout << "harmonic mean: " << hm_omp << endl;
91:     cout << "duration: " << t_means_omp << endl << endl;
92:
93:
94:     cout << "####### C++ #######" << endl;
95:     cout << "minimum: " << min_cpp << endl;
96:     cout << "maximum: " << max_cpp << endl;
97:     cout << "duration: " << t_minmax_cpp << endl << endl;
98:
99:     cout << "arithmetic mean: " << am_cpp << endl;
100:    cout << "geometric mean: " << gm_cpp << endl;
101:    cout << "harmonic mean: " << hm_cpp << endl;
102:    cout << "duration: " << t_means_cpp << endl << endl;
103:
104:
105:
106:    // ####### OpenMP #######
107:    // minimum: 1
108:    // maximum: 1000
109:    // duration: 3.52086e-06
110:
111:    // arithmetic mean: 498.184
112:    // geometric mean: 364.412
113:    // harmonic mean: 95.6857
114:    // duration: 5.90171e-06
115:
116:    // ####### C++ #######
117:    // minimum: 1
118:    // maximum: 1000
119:    // duration: 1.76816e-05
120:
121:    // arithmetic mean: 498.184
122:    // geometric mean: 364.412
123:    // harmonic mean: 95.6857
124:    // duration: 2.35728e-05
125:
126:    // --> the openMP variant is faster in both cases
127:
128:
129:    return 0;
130: }
```

Laufzeiten zu kurz für diese Aussage.

```
    1: #include "mylib.h"
    2: #include <algorithm>
    3: #include <cmath>
    4: #include <execution>
    5: #include <iostream>
    6: #include <numeric>
    7: #include <omp.h>
    8:
    9: using namespace std;
   10:
   11:
   12: void means_omp(const std::vector<size_t> numbers, double &am, double &gm, double &hm
)
   13: {
   14:     size_t const n = numbers.size();
   15:
   16:     am = 0.;
   17:     gm = 0.;
   18:     hm = 0.;
   19:
   20: #pragma omp parallel for shared(numbers, n, cout) reduction(+:am, gm, hm)
   21:     for (size_t i = 0; i < n; ++i)
   22:     {
   23:         am += numbers[i];
   24:         gm += log(numbers[i]);
   25:         hm += 1.0/numbers[i];
   26:
   27:         // #pragma omp critical
   28:         // {
   29:         //     cout << "Thread number " << omp_get_thread_num() << " processes value
 " << numbers[i] << endl;
   30:         // }
   31:     }
   32:
   33:     am /= n;
   34:     gm = exp(gm/n);
   35:     hm = n/hm;
   36: }
   37:
   38:
   39: void minmax_omp(const std::vector<size_t> numbers, size_t &global_min, size_t &globa
l_max)
   40: {
   41:     size_t const n = numbers.size();
   42:
   43:     global_min = -1; // gives the maximum size_t value
   44:     global_max = 0;
   45:
   46: #pragma omp parallel shared(numbers, n, global_min, global_max)
   47:     {
   48:         const size_t nthreads = omp_get_num_threads();
   49:         const size_t threadnum = omp_get_thread_num();
   50:         const size_t chunksize = n/nthreads;
   51:
   52:
   53:         size_t start = threadnum*chunksize;
   54:         size_t end = start + chunksize;
   55:         if (threadnum == nthreads - 1)
   56:             end = n;
   57:
   58:
   59:         size_t local_min = -1;
   60:         size_t local_max = 0;
   61:         for (size_t i = start; i < end ; ++i)
   62:         {
   63:             if (numbers[i] < local_min)
   64:                 local_min = numbers[i];
   65:
```

```
    66:             if (numbers[i] > local_max)
    67:                 local_max = numbers[i];
    68:         }
    69:
    70:         #pragma omp critical
    71:         {
    72:             if (local_min < global_min)
    73:                 global_min = local_min;
    74:
    75:             if (local_max > global_max)
    76:                 global_max = local_max;
    77:         }
    78:     }
    79: }
    80:
    81:
    82: void means_cpp(const std::vector<size_t> numbers, double &am, double &gm, double &hm
)
    83: {
    84:     size_t const n = numbers.size();
    85:
    86:     am = reduce(std::execution::par, numbers.begin(), numbers.end());
    87:     gm = transform_reduce(std::execution::par, numbers.begin(), numbers.end(), 0.0,
plus{}, [] (size_t x) -> double { return log(x); } );
    88:     hm = transform_reduce(std::execution::par, numbers.begin(), numbers.end(), 0.0,
plus{}, [] (size_t x) -> double { return 1.0/x; });
    89:
    90:     am /= n;
    91:     gm = exp(gm/n);
    92:     hm = n/hm;
    93: }
    94:
    95:
    96: void minmax_cpp(const std::vector<size_t> numbers, size_t &global_min, size_t &globa
l_max)
    97: {
    98:     auto min_it = min_element(std::execution::par, numbers.begin(), numbers.end());
    99:     auto max_it = max_element(std::execution::par, numbers.begin(), numbers.end());
   100:
   101:     global_min = *min_it;
   102:     global_max = *max_it;
   103: }
```

```
    1: #include <vector>
    2:
    3: /**
    4:   This function calculates arithmetic mean, geometric mean and harmonic mean of an i
nteger vector.
    5:   Uses openMP parallelization.
    6:   @param[in]    numbers      vector containing integers
    7:   @param[out]   am           arithmetic mean
    8:   @param[out]   gm           geometric mean
    9:   @param[out]   hm           harmonic mean
   10: */
   11: void means_omp(const std::vector<size_t> numbers, double &am, double &gm, double &hm
);
   12:
   13:
   14: /**
   15:   This function calculates the minimum and maximum of a vector.
   16:   Uses openMP parallelization.
   17:   @param[in]    numbers         vector containing integers
   18:   @param[out]   global_min      minimum
   19:   @param[out]   global_max      maximum
   20: */
   21: void minmax_omp(const std::vector<size_t> numbers, size_t &global_min, size_t &globa
l_max);
   22:
   23:
   24: /**
   25:   This function calculates arithmetic mean, geometric mean and harmonic mean of an i
nteger vector.
   26:   Uses C++ parallelization.
   27:   @param[in]    numbers      vector containing integers
   28:   @param[out]   am           arithmetic mean
   29:   @param[out]   gm           geometric mean
   30:   @param[out]   hm           harmonic mean
   31: */
   32: void means_cpp(const std::vector<size_t> numbers, double &am, double &gm, double &hm
);
   33:
   34:
   35: /**
   36:   This function calculates the minimum and maximum of a vector.
   37:   Uses C++ parallelization.
   38:   @param[in]    numbers         vector containing integers
   39:   @param[out]   global_min      minimum
   40:   @param[out]   global_max      maximum
   41: */
   42: void minmax_cpp(const std::vector<size_t> numbers, size_t &global_min, size_t &globa
l_max);
```