```
    1: #pragma once
    2:
    3: #include <iostream>
    4: #ifdef _OPENMP
    5: #include <omp.h>
    6: #endif
    7: #include <unordered_map>
    8:
    9: //#####################################
   10: // G.Haase
   11: // See https://sourceforge.net/p/predef/wiki/Compilers/
   12: //     http://www.cplusplus.com/doc/tutorial/preprocessor/
   13: //  also:  export OMP_DISPLAY_ENV=VERBOSE
   14: //#####################################
   15: /**     Checks for compilers, its versions, threads etc.
   16:  *
   17:         @param[in] argc number of command line arguemnts
   18:         @param[in] argv command line arguments as array of C-strings
   19: */
   20: template <class T>
   21: void check_env(T argc, char const *argv[])
   22: {
   23:     std::cout << "\n#########################################################################
#######\n";
   24:     std::cout << "Code    :";
   25:     for (T k = 0; k < argc; ++k) std::cout << "  " << argv[k];
   26:     std::cout << std::endl;
   27:
   28:     // compiler:      https://sourceforge.net/p/predef/wiki/Compilers/
   29:     std::cout <<    "Compiler:  ";
   30: #if defined __INTEL_COMPILER
   31: #pragma message(" ##########  INTEL  ###############")
   32:     std::cout << "INTEL " << __INTEL_COMPILER;
   33:     // Ignore warnings for #pragma acc   unrecognice
   34: #pragma warning disable 161
   35:     // Ignore warnings for #pragma omp   unrecognice
   36: #pragma warning disable 3180
   37:
   38: #elif defined __PGI
   39: #pragma message(" ##########  PGI    ###############")
   40:     std::cout << "PGI " << __PGIC__ << "." << __PGIC_MINOR__ << "." << __PGIC_PATCHL
EVEL__;
   41: #elif defined  __clang__
   42: #pragma message(" ##########  CLANG    ###############")
   43:     std::cout << "CLANG " << __clang_major__ << "." << __clang_minor__ << "."; // <<
 __clang_patchlevel__;
   44: #elif defined __GNUC__
   45: #pragma message(" ##########  Gnu    ###############")
   46:     std::cout << "Gnu " <<  __GNUC__  << "." << __GNUC_MINOR__ << "." << __GNUC_PATC
HLEVEL__;
   47: #else
   48: #pragma message(" ##########  unknown Compiler   ###############")
   49:     std::cout << "unknown";
   50: #endif
   51:     std::cout << "  C++ standard: " << __cplusplus << std::endl;
   52:
   53:     // Parallel environments
   54:     std::cout <<    "Parallel:  ";
   55: #if defined MPI_VERSION
   56: #pragma message(" ##########  MPI    ###############")
   57: #ifdef OPEN_MPI
   58:     std::cout << "OpenMPI ";
   59: #else
   60:     std::cout << "MPI ";
   61: #endif
   62:     std::cout << MPI_VERSION << "." << MPI_SUBVERSION << "   ";
   63: #endif
   64:
```

```
   65: #ifdef _OPENMP
   66: //https://www.openmp.org/specifications/
   67: //https://stackoverflow.com/questions/1304363/how-to-check-the-version-of-openmp-on-
linux
   68:     std::unordered_map<unsigned, std::string> const map{
   69:         {200505, "2.5"}, {200805, "3.0"}, {201107, "3.1"}, {201307, "4.0"}, {201511,
 "4.5"}, {201611, "5.0"}, {201811, "5.0"}};
   70: #pragma message(" ##########  OPENMP    ###############")
   71:     //std::cout << _OPENMP;
   72:     std::cout << "OpenMP ";
   73:     try {
   74:         std::cout << map.at(_OPENMP);
   75:     }
   76:     catch (...) {
   77:         std::cout << _OPENMP;
   78:     }
   79:     #pragma omp parallel
   80:     {
   81:         #pragma omp master
   82:         {
   83:             const int nn = omp_get_num_threads();           // OpenMP
   84:             std::cout << " ---> " <<  nn << " Threads   ";
   85:         }
   86:         #pragma omp barrier
   87:     }
   88:
   89: #endif
   90: #ifdef _OPENACC
   91: #pragma message(" ##########  OPENACC    ###############")
   92:     std::cout << "OpenACC   ";
   93: #endif
   94:     std::cout << std::endl;
   95:     std::cout << "Date    :  " << __DATE__ << "  " << __TIME__;
   96:     std::cout << "\n#############################################################
#######\n";
   97: }
   98: // HG
   99:
```

```
    1: #include "check_env.h"
    2: #include "mylib.h"
    3: #include <cstdlib>          // atoi()
    4: #include <cstring>          // strncmp()
    5: #include <ctime>
    6: #include <iostream>
    7: #include <omp.h>            // OpenMP
    8: #include <sstream>
    9: #include <string>
   10: using namespace std;
   11:
   12: int main(int argc, char const *argv[])
   13: {
   14:     //int const NLOOPS = 5;        // chose a value such that the benchmark runs at
least 10 sec.
   15:     //unsigned int N = 500000001;       // to less memory :(
   16:     int const NLOOPS = 50;        // chose a value such that the benchmark runs at l
east 10 sec.
   17:     unsigned int N = 50000001;
   18: //#########################################################################
   19: //   Read Parameter from command line  (C++ style)
   20:     cout << "Checking command line parameters for: -n <number> " << endl;
   21:     for (int i = 1; i < argc; i++)
   22:     {
   23:         cout << " arg[" << i << "] = " << argv[i] << endl;
   24:         string ss(argv[i]);
   25:         if ("-n"==ss && i + 1 < argc) // found "-n" followed by another parameter
   26:         {
   27:             N = static_cast<unsigned int>(atoi(argv[i + 1]));
   28:         }
   29:         else
   30:         {
   31:             cout << "Corect call: " << argv[0] << " -n  <number>\n";
   32:         }
   33:     }
   34:
   35:     cout << "\nN = " << N << endl;
   36:
   37:     check_env(argc, argv);
   38: //#########################################################################
   39:
   40:     //omp_set_num_threads(16);
   41:
   42:     int nthreads;                                   // OpenMP
   43:     int numproc = omp_get_num_procs();
   44:     cout << "Es stehen maximal " << numproc << " Kerne zur Verfuegung" << endl;
   45:     #pragma omp parallel default(none) shared(cout,nthreads)
   46:     {
   47:         int const th_id  = omp_get_thread_num();   // OpenMP
   48:         int const nthrds = omp_get_num_threads();  // OpenMP
   49:         stringstream ss;
   50:         ss << "C++: Hello World from thread " << th_id << " / " << nthrds << endl;
   51:         #pragma omp critical
   52:         {
   53:             cout << ss.str();                       // output to a shared ressource
   54:         }
   55:         #pragma omp master
   56:         nthreads = nthrds;                          // transfer nn to to master threa
d
   57:     }
   58:     cout << "    " << nthreads << "   threads have been started." << endl;
   59:
   60: //#########################################################################
   61: //  Memory allocation
   62:     cout << "Memory allocation\n";
   63:
   64:     vector<double> x(N), y(N);
   65:
```

```
   66:     cout.precision(2);
   67:     cout << 2.0 * N *sizeof(x[0]) / 1024 / 1024 / 1024 << " GByte Memory allocated\n
";
   68:     cout.precision(6);
   69:
   70: //#############################################################################
   71: //  Data initialization
   72: //  Special:  x_i = i+1;  y_i = 1/x_i  ==> <x,y> == N
   73:     for (unsigned int i = 0; i < N; ++i)
   74:     {
   75:         x[i] = i + 1;
   76:         y[i] = 1.0 / x[i];
   77:     }
   78:
   79: //#############################################################################
   80:     cout << "\nStart Benchmarking\n";
   81:
   82: // Do calculation
   83:     double tstart = omp_get_wtime();                    // OpenMP
   84:
   85:     double sk(0.0);
   86:     for (int i = 0; i < NLOOPS; ++i)
   87:     {
   88:         sk = scalar(x, y);
   89:         sk = scalar_trans(x, y);
   90:         //sk = norm(x);
   91:         //sk = scalar_manual(x,y);
   92:     }
   93:
   94:     double t1 = omp_get_wtime() - tstart;               // OpenMP
   95:     t1 /= NLOOPS;            // divide by number of function calls
   96:
   97: //#############################################################################
   98: // Check the correct result
   99:     cout << "\n <x,y> = " << sk << endl;
  100:     if (static_cast<unsigned int>(sk) != N)
  101:     {
  102:         cout << "  !!   W R O N G  result   !!\n";
  103:     }
  104:     cout << endl;
  105:
  106: //#############################################################################
  107: // Timings  and Performance
  108:     cout << endl;
  109:     cout.precision(2);
  110:     cout << "Timing in sec. : " << t1 << endl;
  111:     cout << "GFLOPS         : " << 2.0 * N / t1 / 1024 / 1024 / 1024 << endl;
  112:     cout << "GiByte/s        : " << 2.0 * N / t1 / 1024 / 1024 / 1024 * sizeof(x[0])
 << endl;
  113:
  114: //#############################################################################
  115:
  116:     cout << "\n  Try the reduction with an STL-vektor (adding)\n";
  117:
  118:     auto vr = reduction_vec(100);
  119:     cout << "done\n";
  120:     cout << vr << endl;
  121:
  122:
  123: //#############################################################################
  124:
  125:     cout << "\n  Try the reduction with an STL-vektor (appending)\n";
  126:
  127:     auto vra = reduction_vec_append(10);      ✓
  128:     cout << "done\n";
  129:     cout << vra << endl;
  130:
  131:
```

```
  132:     return 0;
  133: }  // memory for x and y will be deallocated their destructors
  134:
```

```cpp
#include "mylib.h"
#include <cassert>       // assert()
#include <cmath>
#include <iostream>
#include <functional>    // multiplies<>{}
#include <list>
#include <numeric>       // iota()
#ifdef _OPENMP
#include <omp.h>
#endif
#include <vector>
using namespace std;

double scalar(vector<double> const &x, vector<double> const &y)
{
    assert(x.size() == y.size()); // switch off via compile flag: -DNDEBUG
    size_t const N = x.size();
    double sum = 0.0;
//    int check;         // for using omp_in_parallel
//#pragma omp parallel for default(none) shared(x,y,N,check) reduction(+:sum)
#pragma omp parallel for default(none) shared(x,y,N) reduction(+:sum)
    for (size_t i = 0; i < N; ++i)
    {
        sum += x[i] * y[i];
//        check = omp_in_parallel();
        //sum += exp(x[i])*log(y[i]);
    }
//    cout << "in parallel: " << check << endl;
    return sum;
}


double scalar_manual(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 default(none) shared(x,y,N,sum)
    {
        int tid  = omp_get_thread_num();
        int nt   = omp_get_num_threads();

        // manual splitting of the index area
        size_t start = (N * tid)     / nt;
        size_t end   = (N * (tid+1)) / nt;

        double local_sum = 0.0;
        for (size_t i = start; i < end; ++i) {
            local_sum += x[i] * y[i];
        }

        // local subtotal combined to global sum
        #pragma omp atomic
        sum += local_sum;
    }

    return sum;
}


double norm(vector<double> const &x)
{
    size_t const N = x.size();
    double sum = 0.0;
#pragma omp parallel for default(none) shared(x,N) reduction(+:sum)
    for (size_t i = 0; i < N; ++i)
    {
```

```
    69:         sum += x[i]*x[i];
    70:     }
    71:     return sum;
    72: }
    73:
    74:
    75:
    76: vector<int> reduction_vec(int n)
    77: {
    78:     vector<int> vec(n);
    79: #pragma omp parallel default(none) shared(cout) reduction(VecAdd:vec)
    80:     {
    81:         #pragma omp barrier
    82:         #pragma omp critical
    83:         cout << omp_get_thread_num() << " : " << vec.size() << endl;
    84:         #pragma omp barrier
    85:         iota( vec.begin(),vec.end(), omp_get_thread_num() );
    86:         #pragma omp barrier
    87:
    88:     }
    89:     return vec;
    90: }
    91:
    92:
    93: vector<int> reduction_vec_append(int n)
    94: {
    95:     vector<int> vec;
    96:
    97: #pragma omp parallel default(none) shared(n,cout) reduction(VecAppend:vec)
    98:     {
    99:         std::vector<int> local(n);      // local vector of each thread
   100:
   101:         // consecutive numbers starting from thread ID
   102:         iota(local.begin(), local.end(), omp_get_thread_num());
   103:
   104:         // output for checking
   105:         #pragma omp critical
   106:         cout << "Thread " << omp_get_thread_num() << " local size = " << local.size(
) << std::endl;
   107:
   108:         // append local to vec
   109:         vec.insert(vec.end(), local.begin(), local.end());
   110:     }
   111:
   112:     return vec;
   113: }
   114:
   115:
   116:
   117: double scalar_trans(vector<double> const &x, vector<double> const &y)
   118: {
   119:     assert(x.size() == y.size()); // switch off via compile flag: -DNDEBUG
   120:     vector<double> z(x.size());
   121:     //list<double> z(x.size());  // parallel for-loop on iterators not possible (mis
sing 'operator-')
   122:                                  // c++-20 CLANG_, ONEAPI_:condition of OpenMP for l
oop must be a relational comparison
   123:
   124:     transform(cbegin(x),cend(x),cbegin(y),begin(z),std::multiplies<>{});
   125:
   126:     double sum = 0.0;
   127: #pragma omp parallel for default(none) shared(z) reduction(+:sum)
   128:     for (auto pi = cbegin(z); pi!=cend(z); ++pi)
   129:     {
   130:         sum += *pi;
   131:     }
   132:     //for (auto val: z)
   133:     //{
```

```
134:         //sum += val;
135:     //}
136:     return sum;
137: }
138:
139:
140:
```

```
    1: #pragma once
    2: #include <cassert>
    3: #include <iomanip>              // setw()
    4: #include <iostream>
    5: #include <omp.h>
    6: #include <vector>
    7:
    8: /**     Inner product
    9:         @param[in] x    vector
   10:         @param[in] y    vector
   11:         @return             resulting Euclidian inner product <x,y>
   12: */
   13: double scalar(std::vector<double> const &x, std::vector<double> const &y);
   14: double scalar_manual(std::vector<double> const &x, std::vector<double> const &y);
   15: double scalar_trans(std::vector<double> const &x, std::vector<double> const &y);
   16:
   17:
   18: /**     l2-norm
   19:         @param[in] x    vector
   20:         @return             resulting Euclidian norm
   21: */
   22: double norm(std::vector<double> const &x);
   23:
   24: /**      Vector @p b adds its elements to vector @p a .
   25:         @param[in] a    vector
   26:         @param[in] b    vector
   27:         @return              a+=b componentwise
   28: */
   29: template<class T>
   30: std::vector<T> &operator+=(std::vector<T> &a, std::vector<T> const &b)
   31: {
   32:     assert(a.size()==b.size());
   33:     for (size_t k = 0; k < a.size(); ++k) {
   34:         a[k] += b[k];
   35:     }
   36:     return a;
   37: }
   38:
   39: // Declare the reduction operation in OpenMP for an STL-vector
   40: //   omp_out += omp_in  requires operator+=(vector<int> &, vector<int> const &) from
 above
   41: // ------------------------------------------------------------
   42: // https://scc.ustc.edu.cn/zlsc/tc4600/intel/2016.0.109/compiler_c/common/core/GUID-
7312910C-D175-4544-99C5-29C12D980744.htm
   43: // https://gist.github.com/eruffaldi/7180bdec4c8c9a11f019dd0ba9a2d68c
   44: // https://stackoverflow.com/questions/29633531/user-defined-reduction-on-vector-of-
varying-size
   45: //  see also p.74ff in  https://www.fz-juelich.de/ias/jsc/EN/AboutUs/Staff/Hagemeier
_A/docs-parallel-programming/OpenMP-Slides.pdf
   46: #pragma omp declare reduction(VecAdd : std::vector<int>  : omp_out += omp_in) \
   47:  initializer (omp_priv=omp_orig)
   48:
   49: #pragma omp declare reduction (VecAppend : std::vector<int> : \
   50:     omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end())) \
   51:     initializer (omp_priv = omp_orig)
   52:
   53: //   Templates are  n o t  possible, i.e. the reduction has to be declared fore a sp
ecified type.
   54: //template <class T>
   55: //#pragma omp declare reduction(VecAdd : std::vector<T>  : omp_out += omp_in) initia
lizer (omp_priv(omp_orig))
   56: // MS: template nach #pragma  !?
   57:
   58: // ------------------------------------------------------------
   59:
   60:
   61: /**     Test for vector reduction.
   62:  *
```

```
   63:  * The thread-private vectors of size @p n are initialized via @f$v_k^{tID}=tID+k@f$
.
   64:  * Afterwards these vectors are accumulated, i.e.,
   65:  * @f$v_k= \sum_{tID=0}^{numThreads} v_k^{tID}@f$.
   66:  *
   67:  *      @param[in] n  size of global/private vector
   68:  *      @return  resulting global vector.
   69: */
   70: std::vector<int> reduction_vec(int n);
   71:
   72:
   73: /**     Test for vector reduction with appending the local vectors to a global
   74:  *      size of global vector: n*#threads
   75:  *
   76:  *  @param[in]  n       size of local/private vector
   77:  *  @return     resultign global vector
   78:  */
   79: std::vector<int> reduction_vec_append(int n);
   80:
   81:
   82: /**     Output of a vector.
   83:         @param[in,out] s        output stream
   84:         @param[in]     x        vector
   85:         @return             modified output stream
   86: */
   87: template <class T>
   88: std::ostream &operator<<(std::ostream &s, std::vector<T> const &x)
   89: {
   90:     for (auto const &v : x)  s << std::setw(4) << v << "  ";
   91:     return s;
   92: }
   93:
```

```
    1: #pragma once
    2: #include <chrono>                    // timing
    3: #include <stack>
    4:
    5: using Clock = std::chrono::system_clock;   //!< The wall clock timer chosen
    6: //using Clock = std::chrono::high_resolution_clock;
    7: using TPoint= std::chrono::time_point<Clock>;
    8:
    9: // [Galowicz, C++17 STL Cookbook, p. 29]
   10: inline
   11: std::stack<TPoint> MyStopWatch; //!< starting time of stopwatch
   12:
   13: /** Starts stopwatch timer.
   14:  *  Use as @code tic(); myfunction(...) ; double tsec = toc();  @endcode
   15:  *
   16:  *  The timining is allowed to be nested and the recent time is stored on top of the
 stack.
   17:  *
   18:  *  @return recent time
   19:  *  @see toc
   20:  */
   21: inline auto tic()
   22: {
   23:     MyStopWatch.push(Clock::now());
   24:     return MyStopWatch.top();
   25: }
   26:
   27: /** Returns the elapsed time from stopwatch.
   28:  *
   29:  * The time from top of the stack is used
   30:  * if time point @p t_b is not passed as input parameter.
   31:  * Use as @code tic(); myfunction(...) ; double tsec = toc();  @endcode
   32:  * or as @code auto t_b = tic(); myfunction(...) ; double tsec = toc(t_b);  @endcode
   33:  * The last option is to be used in the case of
   34:  * non-nested but overlapping time measurements.
   35:  *
   36:  * @param[in]  t_b start time of some stop watch
   37:  * @return elapsed time in seconds.
   38:  *
   39: */
   40: inline double toc(TPoint const &t_b = MyStopWatch.top())
   41: {
   42:     // https://en.cppreference.com/w/cpp/chrono/treat_as_floating_point
   43:     using Unit       = std::chrono::seconds;
   44:     using FpSeconds = std::chrono::duration<double, Unit::period>;
   45:     auto t_e = Clock::now();
   46:     MyStopWatch.pop();
   47:     return FpSeconds(t_e-t_b).count();
   48: }
   49:
   50: #include <iostream>
   51: #include <string>
   52: /** Executes function @p f and measures/prints elapsed wall clock time in seconds
   53:  *
   54:  * Call as
   55:  * @code measure("Time for (b = b + 1)", [&]() {
   56:         thrust::transform(b.begin(), b.end(), b.begin(), increment());
   57:     }); @endcode
   58:  *
   59:  * @param[in]  label  additional string to be printed with the measurement.
   60:  * @param[in]  f      function to execute.
   61:  * @author Therese BÃ¶smÃ¼ller, 2025
   62:  *
   63: */
   64: auto measure = [](const std::string& label, auto&& f) {
   65:         auto start    = std::chrono::high_resolution_clock::now();
   66:         f();
   67:         auto stop     = std::chrono::high_resolution_clock::now();
```

```
   68:         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop -
 start).count();
   69:         std::cout << label << ": " << duration << " microseconds" << std::endl;
   70: };          // ';' is needed for a visible documentation of this lambda-function
   71:
```