```cpp
// bench_funcs.cpp
#include "bench_funcs.h"
#include <omp.h>
#include <cmath>
#include <cstddef>
#include <vector>
#include <algorithm>

//EXERCISE 4
// A: parallel sum
double sum_basic(const std::vector<double>& x)
{
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (std::size_t i = 0; i < x.size(); ++i) sum += x[i];
    return sum;
}

// A: parallel dot product
double dot_basic(const std::vector<double>& x, const std::vector<double>& y)
{
    std::size_t N = x.size();
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (std::size_t i = 0; i < N; ++i) sum += x[i] * y[i];
    return sum;
}

// Kahan remains same
double dot_kahan(const std::vector<double>& x, const std::vector<double>& y)
{
    double sum = 0.0;
    double c = 0.0;
    for (std::size_t i = 0; i < x.size(); ++i)
    {
        double prod = x[i] * y[i];
        double yk = prod - c;
        double t = sum + yk;
        c = (t - sum) - yk;
        sum = t;
    }
    return sum;
}

// Norm (sum of squares)
double norm_basic(const std::vector<double>& x)
{
    double sumsq = 0.0;
    #pragma omp parallel for reduction(+:sumsq)
    for (std::size_t i = 0; i < x.size(); ++i) sumsq += x[i]*x[i];
    return sumsq;
}

// B: matvec (row-major)
void matvec_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t N,
                     const std::vector<double>& x, std::vector<double>& b)
{
    b.assign(M, 0.0);
    #pragma omp parallel for
    for (std::size_t i = 0; i < M; ++i) {
        double tmp = 0.0;
        const double* Ai = &A[i*N];
        for (std::size_t j = 0; j < N; ++j) tmp += Ai[j] * x[j];
        b[i] = tmp;
    }
}

// C: matmul (row-major)
```

```
   69: void matmul_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t L,
   70:                      const std::vector<double>& B, std::size_t N,
   71:                      std::vector<double>& C)
   72: {
   73:     C.assign(M * N, 0.0);
   74:     // Parallelize over output rows (i)
   75:     #pragma omp parallel for
   76:     for (std::size_t i = 0; i < M; ++i) {
   77:         for (std::size_t k = 0; k < L; ++k) {
   78:             double Aik = A[i*L + k];
   79:             const double* Bk = &B[k*N];
   80:             double* Ci = &C[i*N];
   81:             for (std::size_t j = 0; j < N; ++j) {
   82:                 Ci[j] += Aik * Bk[j];
   83:             }
   84:         }
   85:     }
   86: }
   87:
   88: // D: polynomial evaluation (Horner), parallel over x points
   89: void polyp_horner(const std::vector<double>& a, const std::vector<double>& x, std::v
ector<double>& y)
   90: {
   91:     std::size_t p = a.size() - 1;
   92:     std::size_t N = x.size();
   93:     y.assign(N, 0.0);
   94:
   95:     #pragma omp parallel for
   96:     for (std::size_t i = 0; i < N; ++i) {
   97:         double xi = x[i];
   98:         double val = a[p];
   99:         for (std::size_t k = p; k-- > 0; ) {
  100:             val = val * xi + a[k];
  101:         }
  102:         y[i] = val;
  103:     }
  104: }
  105:
  106: //EXERCISE 5
  107: // Parallel assembly of tridiagonal FEM matrix (CSR)
  108: void build_fem_system(std::size_t n, CSR& K, std::vector<double>& f)
  109: {
  110:     K.n = n;
  111:     K.row_ptr.resize(n+1);
  112:     f.assign(n, 1.0);
  113:     K.val.resize(3*n); // Each row has at most 3 non-zero entries: [-1,2,-1]
  114:     K.col.resize(3*n);
  115:
  116:     for (std::size_t i = 0; i < n; ++i)
  117:         K.row_ptr[i] = 3*i;
  118:     K.row_ptr[n] = 3*n;
  119:
  120:     #pragma omp parallel for     // Fill matrix in parallel
  121:     for (std::size_t i = 0; i < n; ++i) {
  122:         std::size_t base = 3*i;
  123:
  124:         if (i > 0) {
  125:             K.val[base] = -1.0;
  126:             K.col[base] = i - 1;
  127:         } else {
  128:             K.val[base] = 0.0;
  129:             K.col[base] = 0;
  130:         }
  131:
  132:         K.val[base + 1] = 2.0;
  133:         K.col[base + 1] = i;
  134:
  135:         if (i + 1 < n) {
```

Try collapse(2)

Doesn't work that simple in 2D FEM.

```
136:             K.val[base + 2] = -1.0;
137:             K.col[base + 2] = i + 1;
138:         } else {
139:             K.val[base + 2] = 0.0;
140:             K.col[base + 2] = i;
141:         }
142:     }
143: }
144:
145: // Parallel Jacobi iteration
146: void jacobi_csr_parallel(const CSR& K, const std::vector<double>& f,
147:                          std::vector<double>& u,
148:                          std::size_t max_iter, double omega, double tol)
149: {
150:     std::size_t n = K.n;
151:     u.assign(n, 0.0);
152:     std::vector<double> u_new(n, 0.0);
153:
154:     for (std::size_t iter = 0; iter < max_iter; ++iter) {
155:         double max_err = 0.0;
156:
157:         #pragma omp parallel for reduction(max:max_err)
158:         for (std::size_t i = 0; i < n; ++i) {
159:             double diag = 0.0;
160:             double sum  = 0.0;
161:
162:             for (std::size_t idx = K.row_ptr[i]; idx < K.row_ptr[i+1]; ++idx) {
163:                 std::size_t j = K.col[idx];
164:                 double v = K.val[idx];
165:                 if (j == i) diag = v;
166:                 else sum += v * u[j];
167:             }
168:
169:             double ui_new = u[i] + omega * (f[i] - sum - diag*u[i]) / diag;
170:             u_new[i] = ui_new;
171:
172:             double err = std::fabs(ui_new - u[i]);
173:             if (err > max_err) max_err = err;
174:         }
175:
176:         u.swap(u_new);
177:         if (max_err < tol) break;
178:     }
179: }
180:
181: //EXERCISE 6
182: // Helper: build empty CSR structure (3 entries per row)
183: static void prepare_tridiag_csr(std::size_t n, CSR &K)
184: {
185:     K.n = n;
186:     K.row_ptr.resize(n + 1);
187:     // Each row has 3 slots (left, diag, right)
188:     for (std::size_t i = 0; i < n; ++i) K.row_ptr[i] = 3 * i;
189:     K.row_ptr[n] = 3 * n;
190:
191:     K.val.assign(3 * n, 0.0);
192:     K.col.resize(3 * n);
193:
194:     // fill col indices
195:     for (std::size_t i = 0; i < n; ++i) {
196:         std::size_t base = 3 * i;
197:         // left column index
198:         if (i > 0) K.col[base + 0] = i - 1;
199:         else       K.col[base + 0] = i; // dummy (no entry)
200:         // diagonal
201:         K.col[base + 1] = i;
202:         // right
203:         if (i + 1 < n) K.col[base + 2] = i + 1;
```

why not [ ] for all j !?

that's the Highway to Hell

```
204:         else            K.col[base + 2] = i; // dummy
205:     }
206: }
207:
208: // Element-level element matrix (1D linear element on uniform mesh)
209: // local ordering: node 0 = left, node 1 = right
210: // element matrix (unscaled) = [ 1  -1; -1  1 ]
211: // For a uniform mesh with h=1/(n-1) stiffness would be 1/h. For simplicity we use 1
.0 scaling.
212: inline void element_matrix_vals(double out[4])
213: {
214:     // ordering out = [K00, K01, K10, K11]
215:     out[0] = 1.0; out[1] = -1.0; out[2] = -1.0; out[3] = 1.0;
216: }
217:
218: // Atomic assembly: parallel over elements, atomic adds to K.val
219: void build_fem_system_atomic(std::size_t n, CSR &K, std::vector<double> &f)
220: {
221:     prepare_tridiag_csr(n, K);
222:     f.assign(n, 1.0); // RHS = 1
223:
224:     std::size_t nel = (n >= 2) ? (n - 1) : 0;
225:     // Parallel over elements
226:     #pragma omp parallel for
227:     for (std::size_t e = 0; e < nel; ++e) {
228:         // global node indices for element e: i = e, j = e+1
229:         std::size_t i = e;
230:         std::size_t j = e + 1;
231:         double Em[4];
232:         element_matrix_vals(Em);
233:
234:         // contributions:
235:         // K(i,i) += Em[0]
236:         // K(i,j) += Em[1]
237:         // K(j,i) += Em[2]
238:         // K(j,j) += Em[3]
239:         std::size_t idx_ii = 3 * i + 1;             // diag of row i
240:         std::size_t idx_ij = 3 * i + 2;             // right of row i
241:         std::size_t idx_ji = 3 * j + 0;             // left of row j
242:         std::size_t idx_jj = 3 * j + 1;             // diag of row j
243:
244:         #pragma omp atomic
245:         K.val[idx_ii] += Em[0];
246:         #pragma omp atomic
247:         K.val[idx_ij] += Em[1];
248:         #pragma omp atomic
249:         K.val[idx_ji] += Em[2];
250:         #pragma omp atomic
251:         K.val[idx_jj] += Em[3];
252:     }
253: }
254:
255: // No-atomic grouped assembly (method (a))
256: // We compute each row i's three entries by summing the (<=2) contributing elements.
257: // Parallel over rows; no atomics.
258: void build_fem_system_no_atomic(std::size_t n, CSR &K, std::vector<double> &f)
259: {
260:     prepare_tridiag_csr(n, K);
261:     f.assign(n, 1.0);
262:
263:     // element matrix values (same for all elements here)
264:     double Em00 = 1.0, Em01 = -1.0, Em10 = -1.0, Em11 = 1.0;
265:
266:     #pragma omp parallel for // parallel over rows (global entries)
267:     for (std::size_t i = 0; i < n; ++i) {
268:         std::size_t base = 3 * i;
269:
270:         // left entry (K[i, i-1]) gets contribution only from element e = i-1
```

```
271:          if (i > 0) K.val[base + 0] = Em10; // from element e=i-1 local(1,0)
272:          else       K.val[base + 0] = 0.0;
273:
274:          // diagonal K[i,i] gets contributions from element e=i-1 (local(1,1)) and e=
i (local(0,0))
275:          double diag = 0.0;
276:          if (i > 0)      diag += Em11; // e = i-1 local(1,1)
277:          if (i + 1 < n)  diag += Em00; // e = i local(0,0)
278:          K.val[base + 1] = diag;
279:
280:          // right entry (K[i,i+1]) gets contribution only from element e = i
281:          if (i + 1 < n) K.val[base + 2] = Em01; // e=i local(0,1)
282:          else           K.val[base + 2] = 0.0;
283:      }
284: }
285:
```

That is only a very special case.

```
 1: #pragma once
 2: #include <vector>
 3: #include <cstddef>
 4:
 5: //EXERCISE 4
 6: // scalar and vector operations
 7: double sum_basic(const std::vector<double>& x);
 8: double dot_basic(const std::vector<double>& x, const std::vector<double>& y);
 9: double dot_kahan(const std::vector<double>& x, const std::vector<double>& y);
10: double norm_basic(const std::vector<double>& x);
11:
12: // matrix-vector, matrix-matrix, polynomial
13: void matvec_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t N,
14:                       const std::vector<double>& x, std::vector<double>& b);
15:
16: void matmul_rowmajor(const std::vector<double>& A, std::size_t M, std::size_t L,
17:                       const std::vector<double>& B, std::size_t N,
18:                       std::vector<double>& C);
19:
20: void polyp_horner(const std::vector<double>& a, const std::vector<double>& x,
21:                    std::vector<double>& y);
22:
23:
24: // EXERCISE 5
25: struct CSR {
26:     std::size_t n;
27:     std::vector<double> val;
28:     std::vector<std::size_t> col;
29:     std::vector<std::size_t> row_ptr;
30: };
31:
32: void jacobi_csr(const CSR& K, const std::vector<double>& f, std::vector<double>& u,
33:                 std::size_t max_iter, double omega, double tol);
34:
35: // build tridiagonal FEM matrix K and rhs f (parallel)
36: void build_fem_system(std::size_t n, CSR& K, std::vector<double>& f);
37:
38: //Jacobi iteration (parallel)
39: void jacobi_csr_parallel(const CSR& K, const std::vector<double>& f,
40:                          std::vector<double>& u,
41:                          std::size_t max_iter, double omega, double tol);
42:
43: // EXERCISE 6
44:
45: // parallel element assembly with atomics
46: void build_fem_system_atomic(std::size_t n, CSR &K, std::vector<double> &f);
47:
48: // No-atomic grouped assembly (method (a))
49: void build_fem_system_no_atomic(std::size_t n, CSR &K, std::vector<double> &f);
50:
```

```
    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: //HOW TO RUN IT ON MY TERMINAL (MAC)
    2: //export LDFLAGS="-L/opt/homebrew/opt/libomp/lib"
    3: //export CPPFLAGS="-I/opt/homebrew/opt/libomp/include"
    4: //clang++ -std=c++17 -O2 -Xpreprocessor -fopenmp $CPPFLAGS main.cpp mylib.cpp -L/opt
/homebrew/opt/libomp/lib -lomp -o dotprod
    5: //./dotprod
    6:
    7: #include "check_env.h"
    8: #include "mylib.h"
    9: #include <cstdlib>          // atoi()
   10: #include <cstring>          // strncmp()
   11: #include <ctime>
   12: #include <iostream>
   13: #include <omp.h>            // OpenMP
   14: #include <sstream>
   15: #include <string>
   16: using namespace std;
   17:
   18: int main(int argc, char const *argv[])
   19: {
   20:     unsigned int N = 5000001;   // smaller default for testing
   21:     int const NLOOPS = 5;       // smaller loop for quick test
   22:
   23:     //#########################################################################
   24:     // Read Parameter from command line
   25:     cout << "=== Checking command line parameters ===" << endl;
   26:     for (int i = 1; i < argc; i++)
   27:     {
   28:         string ss(argv[i]);
   29:         if ("-n"==ss && i + 1 < argc)
   30:             N = static_cast<unsigned int>(atoi(argv[i + 1]));
   31:         else
   32:             cout << "Corect call: " << argv[0] << " -n  <number>\n";
   33:     }
   34:     cout << "\nVector size N = " << N << endl;
   35:
   36:     //#########################################################################
   37:     check_env(argc, argv);
   38:
   39:     //#########################################################################
   40:     cout << "\n=== Starting OpenMP test ===" << endl;
   41:     int nthreads;
   42:     #pragma omp parallel default(none) shared(cout,nthreads)
   43:     {
   44:         int const th_id  = omp_get_thread_num();
   45:         int const nthrds = omp_get_num_threads();
   46:         #pragma omp critical
   47:         cout << "Hello from thread " << th_id << " out of " << nthrds << endl;
   48:         #pragma omp master
   49:         nthreads = nthrds;
   50:     }
   51:     cout << "Total threads started: " << nthreads << endl;
   52:
   53:     //#########################################################################
   54:     cout << "\n=== Memory allocation ===" << endl;
   55:     vector<double> x(N), y(N);
   56:     cout << "Allocated memory for vectors x and y (" << N << " doubles each)" << end
l;
   57:
   58:     //#########################################################################
   59:     cout << "\n=== Data initialization ===" << endl;
   60:     for (unsigned int i = 0; i < N; ++i)
   61:     {
   62:         x[i] = i + 1;
   63:         y[i] = 1.0 / x[i];
   64:     }
   65:     cout << "Vectors initialized: x[i] = i+1, y[i] = 1/x[i]" << endl;
   66:
```

```
   67:     //#############################################################################
   68:     cout << "\n=== Start Benchmarking inner product ===" << endl;
   69:     double tstart = omp_get_wtime();
   70:     double sk(0.0);
   71:     for (int i = 0; i < NLOOPS; ++i)
   72:     {
   73:         sk = scalar(x, y);
   74:         sk = scalar_trans(x, y);
   75:     }
   76:     double t1 = (omp_get_wtime() - tstart) / NLOOPS;
   77:     cout << "<x, y> = " << sk << endl;
   78:     if (static_cast<unsigned int>(sk) != N)
   79:         cout << "!! WRONG result !!" << endl;
   80:
   81:     cout << "Timing (average per loop) in sec: " << t1 << endl;
   82:     cout << "GFLOPS: " << 2.0 * N / t1 / 1024 / 1024 / 1024 << endl;
   83:     cout << "Memory throughput (GiByte/s): " << 2.0 * N / t1 / 1024 / 1024 / 1024 *
sizeof(x[0]) << endl;
   84:
   85:     //#############################################################################
   86:     cout << "\n=== Testing reduction_vec (combining vectors across threads) ===" <<
endl;
   87:     cout << "Each thread initializes a vector of size 100; then vectors are combined
." << endl;
   88:     auto vr = reduction_vec(100);
   89:     cout << "Resulting combined vector:" << endl;
   90:     cout << vr << endl;
   91:
   92:     cout << "\n=== Program finished ===" << endl;
   93:     return 0;
   94: }
```

```cpp
//HOW TO RUN IT ON MY TERMINAL (MAC)
//export PATH="/opt/homebrew/opt/llvm/bin:$PATH"
//export CPPFLAGS="-I/opt/homebrew/opt/libomp/include"
//export LDFLAGS="-L/opt/homebrew/opt/libomp/lib"
//clang++ -std=c++17 -O2 -Xpreprocessor -fopenmp $CPPFLAGS mainEx2.cpp $LDFLAGS -lomp -o Ex2
//./Ex2

#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <omp.h> //for parallelization functions

using namespace std;

// compute arithmetic, geometric, harmonic means (safe geometric mean)
void computeMeansVector(const vector<int> &v, double &arith, double &geom, double &harm) {
    int n = v.size();
    double sum = 0.0, logSum = 0.0, recSum = 0.0;

    for (int x : v) {
        sum += x;
        logSum += log(x);       // accumulate logarithms instead of multiplying
        recSum += 1.0 / x;
    }
    arith = sum / n;
    geom  = exp(logSum / n);    // geometric mean using exponent of average log
    harm  = n / recSum;
}

// compute standard deviation
double computeStdDev(const vector<int> &v, double mean) {
    double sumSq = 0.0;
    for (int x : v) {
        sumSq += (x - mean) * (x - mean);
    }
    return sqrt(sumSq / v.size());
}

int main() {
    vector<int> v;
    ifstream fin("data_1.txt"); //opens data for reading
    int x;
    while (fin >> x) {
        v.push_back(x);
    }
    double t_start = omp_get_wtime();

    // OpenMP threads check
    #pragma omp parallel
    {
        #pragma omp single
        cout << "Using " << omp_get_num_threads()
             << " OpenMP threads\n";
    }

    int minVal = *min_element(v.begin(), v.end());
    int maxVal = *max_element(v.begin(), v.end());

    double arith, geom, harm;
    computeMeansVector(v, arith, geom, harm);

    double stddev = computeStdDev(v, arith);

    ofstream fout("out_1.txt");
```

```
67:     fout << "Min: " << minVal << "\nMax: " << maxVal
68:          << "\nArithmetic mean: " << arith
69:          << "\nGeometric mean: " << geom
70:          << "\nHarmonic mean: " << harm
71:          << "\nStandard deviation: " << stddev << "\n";
72:
73:     cout << "Min: " << minVal << "\nMax: " << maxVal
74:          << "\nArithmetic mean: " << arith
75:          << "\nGeometric mean: " << geom
76:          << "\nHarmonic mean: " << harm
77:          << "\nStandard deviation: " << stddev << endl;
78:
79:     cout << "Elapsed time (s): " << omp_get_wtime() - t_start << endl;
80: }
```

```cpp
//  HOW TO COMPILE ON MAC
// export LDFLAGS="-L/opt/homebrew/opt/libomp/lib"
// export CPPFLAGS="-I/opt/homebrew/opt/libomp/include"
// clang++ -std=c++17 -O3 -Xpreprocessor -fopenmp $CPPFLAGS MainEx3.cpp -L/opt/homebrew/opt/libomp/lib -lomp -o Ex3
// ./Ex3

#include <iostream>
#include <vector>
#include <chrono>
#include <cmath>
#include <algorithm>
#include <utility>
#include <iomanip>
#include <omp.h>
#include "mayer_primes.h"

using namespace std;

int single_goldbach(int k, const vector<int>& primes, const vector<char>& is_prime)
{
    int count = 0;

    #pragma omp parallel for reduction(+:count)
    for (size_t i = 0; i < primes.size(); ++i)
    {
        int p = primes[i];
        if (p > k / 2) continue;
        int q = k - p;
        if (q >= 0 && q < (int)is_prime.size() && is_prime[q])
            count += 1;
    }
    return count;
}

vector<pair<int,int>> count_goldbach(int n, const vector<int>& primes, const vector<char>& is_prime)
{
    vector<pair<int,int>> results(n/2 - 1); // exact size

    #pragma omp parallel for
    for (int j = 0; j < (int)results.size(); ++j)
    {
        int k = 4 + 2*j;
        int c = single_goldbach(k, primes, is_prime);
        results[j] = {k, c};
    }
    return results;
}

vector<vector<pair<int,int>>> all_decompositions(int n, const vector<int>& primes, const vector<char>& is_prime)
{
    vector<vector<pair<int,int>>> out(n/2 - 1);

    #pragma omp parallel for
    for (int j = 0; j < (int)out.size(); ++j)
    {
        int k = 4 + 2*j;
        vector<pair<int,int>> local;

        for (size_t i = 0; i < primes.size(); ++i)
        {
            int p = primes[i];
            if (p > k / 2) break;
            int q = k - p;
            if (q >= 0 && q < (int)is_prime.size() && is_prime[q])
                local.push_back({p, q});
```

nested parallelization:
* did that pay off?

```
66:         }
67:
68:         out[j] = std::move(local);
69:     }
70:     return out;
71: }
72:
73: vector<char> make_is_prime(int n, const vector<int>& primes)
74: {
75:     vector<char> is_prime(n + 1, 0);
76:     for (int p : primes)
77:         if (p <= n) is_prime[p] = 1;
78:     return is_prime;
79: }
80:
81: int main()
82: {
83:     vector<int> test_ns = {10000, 100000, 400000};
84:
85:     for (int n : test_ns)
86:     {
87:         cout << "Computing Goldbach up to n = " << n << "\n";
88:
89:         // CHECKS to see if I'm actually paralellizing
90:         #pragma omp parallel
91:         {
92:             #pragma omp single
93:             cout << "Using " << omp_get_num_threads()
94:                  << " OpenMP threads\n";
95:         }
96:
97:         // Generate primes
98:         auto t0 = chrono::system_clock::now();
99:         vector<int> primes = get_primes<int>(n);
100:        auto t1 = chrono::system_clock::now();
101:        chrono::duration<double> gen_time = t1 - t0;
102:
103:        cout << "Generated " << primes.size()
104:             << " primes in " << gen_time.count() << " s\n";
105:
106:        vector<char> is_prime = make_is_prime(n, primes);
107:
108:        // Count Goldbach decompositions
109:        auto start = chrono::system_clock::now();
110:        vector<pair<int,int>> results = count_goldbach(n, primes, is_prime);
111:        auto end = chrono::system_clock::now();
112:        chrono::duration<double> elapsed = end - start;
113:
114:        // Find k with maximum decompositions
115:        int max_k = 0;
116:        int max_count = 0;
117:
118:        for (auto &pr : results)
119:        {
120:            if (pr.second > max_count)
121:            {
122:                max_k = pr.first;
123:                max_count = pr.second;
124:            }
125:        }
126:
127:        cout << "Max decompositions: k = " << max_k
128:             << " with " << max_count << " decompositions\n";
129:
130:        cout << "Time to count decompositions: "
131:             << elapsed.count() << " s\n";
132:    }
133:
```

```
  134:     return 0;
  135: }
```

```
    1: // HOW TO COMPILE ON MAC
    2: // export CPPFLAGS="-I/opt/homebrew/opt/libomp/include"
    3: // export LDFLAGS="-L/opt/homebrew/opt/libomp/lib"
    4: // clang++ -std=c++17 -O3 -Xpreprocessor -fopenmp $CPPFLAGS main.cpp bench_funcs.cpp
 $LDFLAGS -lomp -o Ex4
    5: // ./Ex4
    6:
    7:
    8: #include <iostream>
    9: #include <vector>
   10: #include <cmath>
   11: #include <iomanip>
   12: #include <chrono>
   13: #include "bench_funcs.h"
   14: #include <omp.h>
   15:
   16: using namespace std;
   17: using namespace std::chrono;
   18:
   19: void gen_vector_x_y(size_t N, vector<double>& x, vector<double>& y) {
   20:     x.resize(N);
   21:     y.resize(N);
   22:     for (size_t i = 0; i < N; ++i) {
   23:         x[i] = (i % 219) + 1;
   24:         y[i] = 1.0 / x[i];
   25:     }
   26: }
   27:
   28: void gen_matrix_A(size_t M, size_t N, vector<double>& A) {
   29:     A.resize(M * N);
   30:     for (size_t i = 0; i < M; ++i)
   31:         for (size_t j = 0; j < N; ++j)
   32:             A[i * N + j] = ((i + j) % 219) + 1;
   33: }
   34:
   35: high_resolution_clock::time_point tic_timer;
   36: void tic() { tic_timer = high_resolution_clock::now(); }
   37: double toc() {
   38:     auto t1 = high_resolution_clock::now();
   39:     duration<double> elapsed = t1 - tic_timer;
   40:     return elapsed.count();
   41: }
   42:
   43: // CHANGE THE flag VARIABLE IN main() TO CHOOSE A, B, C, D.
   44: int main() {
   45:     size_t N = 1'000'000;
   46:     size_t M = 3000, L = 3000;
   47:
   48:     // Check OpenMP threads
   49:     #pragma omp parallel
   50:     {
   51:         #pragma omp single
   52:         cout << "Using " << omp_get_num_threads() << " OpenMP threads\n";
   53:     }
   54:
   55:     // Show menu
   56:     cout << "Choose an option:\n";
   57:     cout << "  1) SUM + DOT PRODUCT\n";
   58:     cout << "  2) MATRIX-VECTOR PRODUCT\n";
   59:     cout << "  3) MATRIX-MATRIX PRODUCT\n";
   60:     cout << "  4) POLYNOMIAL EVALUATION\n";
   61:     cout << "Enter your choice (1-4): ";
   62:
   63:     int flag;
   64:     cin >> flag;
   65:
   66:     // validate input
   67:     if(flag < 1 || flag > 4){
```

```
    68:         cout << "Invalid choice. Exiting.\n";
    69:         return 1;
    70:     }
    71:
    72:     // A) SUM + DOT PRODUCT (parallel)
    73:     if (flag == 1) {
    74:         vector<double> x, y;
    75:         gen_vector_x_y(N, x, y);
    76:
    77:         cout << "Running parallel SUM" << endl;
    78:         tic();
    79:         double ssum = sum_basic(x); (void)ssum;
    80:         double dt_sum = toc();
    81:         cout << "  sum time = " << dt_sum << " s\n";
    82:
    83:         cout << "Running parallel DOT PRODUCT" << endl;
    84:         tic();
    85:         double sdot = dot_basic(x, y); (void)sdot;
    86:         double dt_dot = toc();
    87:         cout << "  dot time = " << dt_dot << " s\n";
    88:
    89:         double flops_dot = 2.0 * N;
    90:         double gflops_dot = (flops_dot / dt_dot) / 1e9;
    91:
    92:         cout << "A: N=" << N << "\n";
    93:         cout << "   SUM time=" << dt_sum << " s\n";
    94:         cout << "   DOT time=" << dt_dot << " s  GFLOPS=" << gflops_dot << "\n";
    95:     }
    96:
    97:     // B) MATRIXâ\200\223VECTOR PRODUCT (parallel)
    98:     else if (flag == 2) {
    99:         size_t m = M, n = 5000;
   100:         vector<double> A, x, b;
   101:
   102:         gen_matrix_A(m, n, A);
   103:         x.resize(n);
   104:         for (size_t j = 0; j < n; ++j)
   105:             x[j] = 1.0 / (((17 + j) % 219) + 1);
   106:
   107:         cout << "Running Matrix times vector (parallel)\n";
   108:         tic();
   109:         matvec_rowmajor(A, m, n, x, b);
   110:         double dt = toc();
   111:
   112:         double flops = 2.0 * m * n;
   113:         double gflops = (flops / dt) / 1e9;
   114:
   115:         cout << "B: M=" << m << " N=" << n
   116:              << " time=" << dt << " s  GFLOPS=" << gflops << endl;
   117:     }
   118:
   119:     // C) MATRIXâ\200\223MATRIX PRODUCT (parallel)
   120:     else if (flag == 3) {
   121:         size_t m = M, l = L, n = 500;
   122:         vector<double> A, B, C;
   123:
   124:         gen_matrix_A(m, l, A);
   125:         gen_matrix_A(l, n, B);
   126:
   127:         cout << "Running Multiplication of matrices (parallel)\n";
   128:         tic();
   129:         matmul_rowmajor(A, m, l, B, n, C);
   130:         double dt = toc();
   131:
   132:         double flops = 2.0 * m * l * n;
   133:         double gflops = (flops / dt) / 1e9;
   134:
   135:         cout << "C: M=" << m << " L=" << l << " N=" << n
```

```
136:                 << " time=" << dt << " s  GFLOPS=" << gflops << endl;
137:     }
138:
139:     // D) POLYNOMIAL EVALUATION (parallel Horner)
140:     else if (flag == 4) {
141:         size_t p = 100; // degree
142:         vector<double> a(p+1), x(N), y;
143:
144:         for (size_t k = 0; k <= p; ++k)
145:             a[k] = 1.0 / (k+1);
146:
147:         for (size_t i = 0; i < N; ++i)
148:             x[i] = (i % 219) * 0.001 + 1.0;
149:
150:         cout << "Running polynomial function (parallel)\n";
151:         tic();
152:         polyp_horner(a, x, y);
153:         double dt = toc();
154:
155:         double flops = 2.0 * p * N;
156:         double gflops = (flops / dt) / 1e9;
157:
158:         cout << "D: p=" << p << " N=" << N
159:              << " time=" << dt << " s  GFLOPS=" << gflops << endl;
160:     }
161:
162:     else {
163:         cout << "Invalid flag â\200\224 choose 1, 2, 3, or 4.\n";
164:     }
165:
166:     cout << "\nDone \n";
167:     return 0;
168: }
```

```cpp
// HOW TO COMPILE ON MACOS
// export CPPFLAGS="-I/opt/homebrew/opt/libomp/include"
// export LDFLAGS="-L/opt/homebrew/opt/libomp/lib"
// clang++ -std=c++17 -O3 -Xpreprocessor -fopenmp $CPPFLAGS main.cpp bench_funcs.cpp $LDFLAGS -lomp -o Ex5
// ./Ex5

#include <iostream>
#include <vector>
#include <chrono>
#include <omp.h>
#include "bench_funcs.h"

using namespace std;
using namespace std::chrono;

int main() {
    size_t n = 200000;       // FEM system size
    size_t maxit = 5000;
    double omega = 1.0;
    double tol = 1e-8;

    // show threads
    #pragma omp parallel
    {
        #pragma omp single
        cout << "Using " << omp_get_num_threads() << " OpenMP threads\n";
    }

    CSR K;
    vector<double> f, u;

    cout << "\nBuilding FEM system (parallel)\n";
    auto t0 = high_resolution_clock::now();
    build_fem_system(n, K, f);
    auto t1 = high_resolution_clock::now();
    cout << "Assembly time = " << duration<double>(t1 - t0).count() << " s\n";

    cout << "\nRunning Jacobi solver (parallel)\n";
    t0 = high_resolution_clock::now();
    jacobi_csr_parallel(K, f, u, maxit, omega, tol);
    t1 = high_resolution_clock::now();

    cout << "Jacobi time = " << duration<double>(t1 - t0).count() << " s\n";
    cout << "Done\n";
}
```

```
    1: // HOW TO COMPILE ON MAC (paste these lines in terminal):
    2: // export CPPFLAGS="-I/opt/homebrew/opt/libomp/include"
    3: // export LDFLAGS="-L/opt/homebrew/opt/libomp/lib"
    4: // clang++ -std=c++17 -O3 -Xpreprocessor -fopenmp $CPPFLAGS mainEx6.cpp bench_funcs.
cpp $LDFLAGS -lomp -o Ex6
    5: // ./Ex6
    6: //
    7: // You can set OMP_NUM_THREADS in the shell to control threads.
    8:
    9: #include <iostream>
   10: #include <vector>
   11: #include <chrono>
   12: #include <omp.h>
   13: #include "bench_funcs.h"
   14:
   15: using namespace std;
   16: using namespace std::chrono;
   17:
   18: int main()
   19: {
   20:     size_t n = 200000;
   21:     size_t maxit = 5000;
   22:     double omega = 1.0;
   23:     double tol = 1e-8;
   24:
   25:     // show threads
   26:     #pragma omp parallel
   27:     {
   28:         #pragma omp single
   29:         cout << "Using " << omp_get_num_threads() << " OpenMP threads\n";
   30:     }
   31:
   32:     CSR K_atomic, K_noatom;
   33:     vector<double> f;
   34:
   35:     cout << "\n--- Build with ATOMICS (parallel over elements with atomic adds) ---\
n";
   36:     auto t0 = high_resolution_clock::now();
   37:     build_fem_system_atomic(n, K_atomic, f);
   38:     auto t1 = high_resolution_clock::now();
   39:     cout << "Assembly (atomic) time = " << duration<double>(t1 - t0).count() << " s\
n";
   40:
   41:     cout << "\nBuild with NO ATOMICS (grouped / per-row accumulation)\n";
   42:     t0 = high_resolution_clock::now();
   43:     build_fem_system_no_atomic(n, K_noatom, f);
   44:     t1 = high_resolution_clock::now();
   45:     cout << "Assembly (no-atomic) time = " << duration<double>(t1 - t0).count() << "
 s\n";
   46:
   47:     // quick check: compare val arrays (they should be equal)
   48:     bool same = true;
   49:     if (K_atomic.val.size() == K_noatom.val.size()) {
   50:         for (size_t i = 0; i < K_atomic.val.size(); ++i) {
   51:             if (std::fabs(K_atomic.val[i] - K_noatom.val[i]) > 1e-12) {
   52:                 same = false; break;
   53:             }
   54:         }
   55:     } else same = false;
   56:
   57:     cout << "\nMatrix equality check (atomic vs no-atomic): " << (same ? "OK" : "DIF
FER") << "\n";
   58:
   59:     // Run Jacobi solver on the no-atomic assembled matrix
   60:     cout << "\nRun Jacobi on NO-ATOMIC matrix\n";
   61:     vector<double> u;
   62:     t0 = high_resolution_clock::now();
   63:     jacobi_csr_parallel(K_noatom, f, u, maxit, omega, tol);
```

```
64:     t1 = high_resolution_clock::now();
65:     cout << "Jacobi time = " << duration<double>(t1 - t0).count() << " s\n";
66:
67:     cout << "\nDone.\n";
68:     return 0;
69: }
```

```
    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 "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;
#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];
        //sum += exp(x[i])*log(y[i]);
    }
    return sum;
}

double scalar_trans(vector<double> const &x, vector<double> const &y)
{
    assert(x.size() == y.size()); // switch off via compile flag: -DNDEBUG
    vector<double> z(x.size());
    transform(cbegin(x),cend(x),cbegin(y),begin(z),std::multiplies<>{});
    double sum = 0.0;
#pragma omp parallel for default(none) shared(z) reduction(+:sum)
    for (auto pi = cbegin(z); pi!=cend(z); ++pi)
    {
        sum += *pi;
    }
    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)
    {
        sum += x[i]*x[i];
    }
    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) reduction(+:sum)
    {
        int tid = omp_get_thread_num();
        int nth = omp_get_num_threads();
        // manual cyclic distribution
        for (size_t i = static_cast<size_t>(tid); i < N; i += static_cast<size_t>(nt
h)) {
            sum += x[i] * y[i];
```

```
    68:         }
    69:     }
    70:     return sum;
    71: }
    72:
    73: // ----------------------------------------------------------------------
    74: vector<int> reduction_vec(int n)
    75: {
    76:     vector<int> vec(n);
    77: #pragma omp parallel default(none) shared(cout) reduction(VecAdd:vec)
    78:     {
    79:         #pragma omp barrier
    80:         #pragma omp critical
    81:         cout << omp_get_thread_num() << " : " << vec.size() << endl;
    82:         #pragma omp barrier
    83:         iota( vec.begin(),vec.end(), omp_get_thread_num() );
    84:         #pragma omp barrier
    85:
    86:     }
    87:     return vec;
    88: }
    89:
    90: // ----------------------------------------------------------------------
    91: vector<int> reduction_vec_append(int n)
    92: {
    93:     // determine number of threads that will be used
    94:     int nth = 1;
    95: #pragma omp parallel
    96:     {
    97:         #pragma omp master
    98:         nth = omp_get_num_threads();
    99:     }
   100:
   101:     vector<int> result;
   102:     result.resize(static_cast<size_t>(n) * static_cast<size_t>(nth));
   103:
   104:     // Each thread will fill its own contiguous block [tid*n, tid*n + n)
   105: #pragma omp parallel default(none) shared(result,n)
   106:     {
   107:         int tid = omp_get_thread_num();
   108:         // create local vector and initialize with a pattern (e.g., tid + k)
   109:         vector<int> local(n);
   110:         iota(local.begin(), local.end(), tid); // local[k] = tid + k
   111:
   112:         size_t offset = static_cast<size_t>(tid) * static_cast<size_t>(n);
   113:         for (int k = 0; k < n; ++k) {
   114:             result[offset + static_cast<size_t>(k)] = local[static_cast<size_t>(k)];
   115:         }
   116:     }
   117:     return result;
   118: }
```

```
    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_trans(std::vector<double> const &x, std::vector<double> const &y);
   15:
   16: /**  Second inner product: use #pragma omp parallel (no "for" in pragma)
   17:     The work is split manually inside the parallel region (stride or chunk).
   18: */
   19: double scalar_manual(std::vector<double> const &x, std::vector<double> const &y);
   20:
   21: /**     l2-norm
   22:     @param[in] x    vector
   23:     @return         resulting Euclidian norm
   24: */
   25: double norm(std::vector<double> const &x);
   26:
   27: /**      Vector @p b adds its elements to vector @p a .
   28:     @param[in] a    vector
   29:     @param[in] b    vector
   30:     @return          a+=b componentwise
   31: */
   32: template<class T>
   33: std::vector<T> &operator+=(std::vector<T> &a, std::vector<T> const &b)
   34: {
   35:     assert(a.size()==b.size());
   36:     for (size_t k = 0; k < a.size(); ++k) {
   37:         a[k] += b[k];
   38:     }
   39:     return a;
   40: }
   41:
   42: // Declare the reduction operation in OpenMP for an STL-vector (existing)
   43: #pragma omp declare reduction(VecAdd : std::vector<int>  : omp_out += omp_in) \
   44:  initializer (omp_priv=omp_orig)
   45:
   46: /**     Test for vector reduction.
   47:  *  existing: converts thread-private vectors into a componentwise sum
   48:  *  @param[in] n  size of global/private vector
   49:  *  @return  resulting global vector.
   50: */
   51: std::vector<int> reduction_vec(int n);
   52:
   53: /**     New: append per-thread vectors into a single big vector.
   54:  *  The result will have size n * numThreads, where each thread contributes a contig
uous block.
   55:  */
   56: std::vector<int> reduction_vec_append(int n);
   57:
   58: /**     Output of a vector.
   59:     @param[in,out] s    output stream
   60:     @param[in]     x    vector
   61:     @return        modified output stream
   62: */
   63: template <class T>
   64: std::ostream &operator<<(std::ostream &s, std::vector<T> const &x)
   65: {
   66:     for (auto const &v : x)  s << std::setw(4) << v << "  ";
   67:     return s;
```

```
68: }
```