```
    1: // see:   http://llvm.org/docs/CodingStandards.html#include-style
    2: #include "geom.h"
    3:
    4: #include <algorithm>
    5: #include <cassert>
    6: #include <fstream>
    7: #include <iostream>
    8: #include <list>
    9: #include <string>
   10: #include <vector>
   11: using namespace std;
   12:
   13: Mesh::Mesh(int ndim, int nvert_e, int ndof_e)
   14:     : _nelem(0), _nvert_e(nvert_e), _ndof_e(ndof_e), _nnode(0), _ndim(ndim), _ia(0),
 _xc(0)
   15: {
   16: }
   17:
   18: Mesh::~Mesh()
   19: {}
   20:
   21: void Mesh::SetValues(std::vector<double> &v, const std::function<double(double, doub
le)> &func) const
   22: {
   23:     int const nnode = Nnodes();            // number of vertices in mesh
   24:     assert( nnode == static_cast<int>(v.size()) );
   25:     #pragma omp parallel for
   26:     for (int k = 0; k < nnode; ++k)
   27:     {
   28:         v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
   29:     }
   30: }
   31:
   32:
   33: void Mesh::Debug() const
   34: {
   35:     cout << "\n ################ Debug  M E S H  ####################\n";
   36:     cout << "\n ..............    Coordinates       ..................\n";
   37:     for (int k = 0; k < _nnode; ++k)
   38:     {
   39:         cout << k << " : " ;
   40:         for (int i = 0; i < _ndof_e; ++i )
   41:         {
   42:             cout << _xc[2*k+i] << "  ";
   43:         }
   44:         cout << endl;
   45:     }
   46:     cout << "\n ..............    Elements        ..................\n";
   47:     for (int k = 0; k < _nelem; ++k)
   48:     {
   49:         cout << k << " : ";
   50:         for (int i = 0; i < _ndof_e; ++i )
   51:             cout << _ia[_ndof_e * k + i] << "  ";
   52:         cout << endl;
   53:     }
   54:     return;
   55: }
   56:
   57: void Mesh::Write_ascii_matlab(std::string const &fname, std::vector<double> const &v
) const
   58: {
   59:     assert(Nnodes() ==  static_cast<int>(v.size()));  // fits vector length to mesh
information?
   60:
   61:     ofstream fout(fname);                             // open file ASCII mode
   62:     if ( !fout.is_open() )
   63:     {
   64:         cout << "\nFile " << fname << " has not been opened.\n\n" ;
```

```
    65:         assert( fout.is_open() && "File not opened."  );
    66:     }
    67:
    68:     string const DELIMETER(" ");    // define the same delimeter as in matlab/ascii_
read*.m
    69:     int const    OFFSET(1);         // convert C-indexing to matlab
    70:
    71:     // Write data: #nodes, #space dimensions, #elements, #vertices per element
    72:     fout << Nnodes() << DELIMETER << Ndims() << DELIMETER << Nelems() << DELIMETER <
< NverticesElements() << endl;
    73:
    74:     // Write cordinates: x_k, y_k   in seperate lines
    75:     assert( Nnodes()*Ndims() ==  static_cast<int>(_xc.size()));
    76:     for (int k = 0, kj = 0; k < Nnodes(); ++k)
    77:     {
    78:         for (int j = 0; j < Ndims(); ++j, ++kj)
    79:         {
    80:             fout << _xc[kj] << DELIMETER;
    81:         }
    82:         fout << endl;
    83:     }
    84:
    85:     // Write connectivity: ia_k,0, ia_k,1 etc  in seperate lines
    86:     assert( Nelems()*NverticesElements() ==  static_cast<int>(_ia.size()));
    87:     for (int k = 0, kj = 0; k < Nelems(); ++k)
    88:     {
    89:         for (int j = 0; j < NverticesElements(); ++j, ++kj)
    90:         {
    91:             fout << _ia[kj] + OFFSET << DELIMETER;     // C to matlab
    92:         }
    93:         fout << endl;
    94:     }
    95:
    96:     // Write vector
    97:     for (int k = 0; k < Nnodes(); ++k)
    98:     {
    99:         fout << v[k] << endl;
   100:     }
   101:
   102:     fout.close();
   103:     return;
   104: }
   105:
   106:
   107: void Mesh::Visualize(std::vector<double> const &v) const
   108: {
   109:     // define external command
   110:     const string exec_m("matlab -nosplash < visualize_results.m");                 /
/ Matlab
   111:     //const string exec_m("octave --no-window-system --no-gui visualize_results.m");
 // Octave
   112:     //const string exec_m("flatpak run org.octave.Octave visualize_results.m");
 // Octave (flatpak): desktop GH
   113:
   114:     const string fname("uv.txt");
   115:     Write_ascii_matlab(fname, v);
   116:
   117:     int ierror = system(exec_m.c_str());                                  // call ext
ernal command
   118:
   119:     if (ierror != 0)
   120:     {
   121:         cout << endl << "Check path to Matlab/octave on your system" << endl;
   122:     }
   123:     cout << endl;
   124:     return;
   125: }
   126:
```

```
127:
128:
129:
130: // ######################################################################
131: Mesh_2d_3_square::Mesh_2d_3_square(int nx, int ny, int myid, int procx, int procy)
132:     : Mesh(2, 3, 3), // two dimensions, 3 vertices, 3 dofs
133:       _myid(myid), _procx(procx), _procy(procy), _neigh{{-1, -1, -1, -1}}, _color(0)
,
134: _xl(0.0), _xr(1.0), _yb(0.0), _yt(1.0), _nx(nx), _ny(ny)
135: {
136:     //void IniGeom(int const myid, int const procx, int const procy, int neigh[], in
t &color)
137:     int const ky = _myid / _procx;
138:     int const kx = _myid % _procy;  //    MOD(myid,procx)
139:     // Determine the neighbors of domain/rank myid
140:     _neigh[0] = (ky == 0)       ?  -1 : _myid - _procx;    //   South
141:     _neigh[1] = (kx == _procx - 1) ?  -1 : _myid + 1;      //   East
142:     _neigh[2] = (ky == _procy - 1) ?  -1 : _myid + _procx;  //   North
143:     _neigh[3] = (kx == 0)       ?  -1 : _myid - 1;        //   West
144:
145:     _color = (kx + ky) & 1 ;
146:
147:     // subdomain is part of unit square
148:     double const hx = 1. / _procx;
149:     double const hy = 1. / _procy;
150:     _xl = kx * hx;                      //  left
151:     _xr = (kx + 1) * hx;                //  right
152:     _yb = ky * hy;                      //  bottom
153:     _yt = (ky + 1) * hy;                //  top
154:
155:     // Calculate coordinates
156:     int const nnode = (_nx + 1) * (_ny + 1); // number of nodes
157:     Resize_Coords(nnode, 2);                 // coordinates in 2D [nnode][ndim]
158:     GetCoordsInRectangle(_nx, _ny, _xl, _xr, _yb, _yt, GetCoords().data());
159:
160:     // Calculate element connectivity (linear triangles)
161:     int const nelem = 2 * _nx * _ny;         // number of elements
162:     Resize_Connectivity(nelem, 3);           // connectivity matrix [nelem][3]
163:     GetConnectivityInRectangle(_nx, _ny, GetConnectivity().data());
164:
165:     return;
166: }
167:
168:
169: void Mesh_2d_3_square::SetU(std::vector<double> &u) const
170: {
171:     int dx    = _nx + 1;
172:     for (int j = 0; j <= _ny; ++j)
173:     {
174:         int k = j * dx;
175:         for (int i = 0; i <= _nx; ++i, ++k)
176:         {
177:             u[k] = 0.0;
178:         }
179:     }
180:
181: }
182:
183: void Mesh_2d_3_square::SetF(std::vector<double> &f) const
184: {
185:     int dx    = _nx + 1;
186:     for (int j = 0; j <= _ny; ++j)
187:     {
188:         int k = j * dx;
189:         for (int i = 0; i <= _nx; ++i, ++k)
190:         {
191:             f[k] = 1.0;
192:         }
```

```
  193:     }
  194:
  195: }
  196:
  197:
  198: std::vector<int> Mesh_2d_3_square::Index_DirichletNodes() const
  199: {
  200:     int const dx = 1,
  201:               dy = _nx + 1,
  202:               bl  = 0,
  203:               br  = _nx,
  204:               tl  = _ny * (_nx + 1),
  205:               tr  = (_ny + 1) * (_nx + 1) - 1;
  206:     int const start[4] = { bl, br, tl, bl},
  207:                         end[4] = { br, tr, tr, tl},
  208:                                  step[4] = { dx, dy, dx, dy};
  209:
  210:     vector<int> idx(0);
  211:     for (int j = 0; j < 4; j++)
  212:     {
  213:         if (_neigh[j] < 0)
  214:         {
  215:             for (int i = start[j]; i <= end[j]; i += step[j])
  216:             {
  217:                 idx.push_back(i);        // node i is Dirichlet node
  218:             }
  219:         }
  220:     }
  221:     //    remove multiple elements
  222:     sort(idx.begin(), idx.end());                           // sort
  223:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
  224:
  225:     return idx;
  226: }
  227:
  228: void Mesh_2d_3_square::SaveVectorP(std::string const &name, vector<double> const &u)
 const
  229: {
  230: //  construct the file name for subdomain myid
  231:     const string tmp( std::to_string(_myid / 100) + to_string((_myid % 100) / 10) +
to_string(_myid % 10) );
  232:
  233:     const string namep = name + "." + tmp;
  234:     ofstream ff(namep.c_str());
  235:     ff.precision(6);
  236:     ff.setf(ios::fixed, ios::floatfield);
  237:
  238:     // assumes tensor product grid in unit square; rowise numbered (as generated in
class constructor)
  239:     // output is provided for tensor product grid visualization ( similar to Matlab-
surf() )
  240:     auto const &xc = GetCoords();
  241:     int k = 0;
  242:     for (int j = 0; j <= _ny; ++j)
  243:     {
  244:         for (int i = 0; i <= _nx; ++i, ++k)
  245:             ff << xc[2 * k + 0] << "   " << xc[2 * k + 1] << "   " << u[k] << endl;
  246:         ff << endl;
  247:     }
  248:
  249:     ff.close();
  250:     return;
  251: }
  252:
  253: void Mesh_2d_3_square::GetCoordsInRectangle(int const nx, int const ny,
  254:         double const xl, double const xr, double const yb, double const yt,
  255:         double xc[])
  256: {
```

```
  257:     const double hx = (xr - xl) / nx,
  258:                  hy = (yt - yb) / ny;
  259:
  260:     int k = 0;
  261:     for (int j = 0; j <= ny; ++j)
  262:     {
  263:         const double y0 = yb + j * hy;
  264:         for (int i = 0; i <= nx; ++i, k += 2)
  265:         {
  266:             xc[k  ] = xl + i * hx;
  267:             xc[k + 1] = y0;
  268:         }
  269:     }
  270:
  271:     return;
  272: }
  273:
  274: void Mesh_2d_3_square::GetConnectivityInRectangle(int const nx, int const ny, int ia
[])
  275: {
  276:     const int dx = nx + 1;
  277:     int k  = 0;
  278:     int l  = 0;
  279:     for (int j = 0; j < ny; ++j, ++k)
  280:     {
  281:         for (int i = 0; i < nx; ++i, ++k)
  282:         {
  283:             ia[l  ] = k;
  284:             ia[l + 1] = k + 1;
  285:             ia[l + 2] = k + dx + 1;
  286:             l += 3;
  287:             ia[l  ] = k;
  288:             ia[l + 1] = k + dx;
  289:             ia[l + 2] = k + dx + 1;
  290:             l += 3;
  291:         }
  292:     }
  293:     return;
  294: }
  295:
  296: // ################### still some old code (--> MPI) ############################
  297:
  298:
  299: //  Copies the values of w corresponding to the boundary
  300: //  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
  301:
  302: void GetBound(int const ib, int const nx, int const ny, double const w[], double s[]
)
  303: {
  304:     const int //dx = 1,
  305:     dy = nx + 1,
  306:     bl  = 0,
  307:     br  = nx,
  308:     tl  = ny * (nx + 1),
  309:     tr  = (ny + 1) * (nx + 1) - 1;
  310:     switch (ib)
  311:     {
  312:         case 1:
  313:         {
  314:             for (int i = bl, j = 0; i <= br; ++i, ++j)
  315:                 s[j] = w[i];
  316:             break;
  317:         }
  318:         case 3:
  319:         {
  320:             for (int i = tl, j = 0; i <= tr; ++i, ++j)
  321:                 s[j] = w[i];
  322:             break;
```

```
  323:         }
  324:         case 4:
  325:         {
  326:             for (int i = bl, j = 0; i <= tl; i += dy, ++j)
  327:                 s[j] = w[i];
  328:             break;
  329:         }
  330:         case 2:
  331:         {
  332:             for (int i = br, j = 0; i <= tr; i += dy, ++j)
  333:                 s[j] = w[i];
  334:             break;
  335:         }
  336:         default:
  337:         {
  338:             cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__
 << endl;
  339:         }
  340:     }
  341:     return;
  342: }
  343:
  344: // ------------------------------------------------------------------------------
-------------------------
  345: // Computes w:  = w + s at nodes on  the boundary
  346: // South (ib == 1), East (ib == 2), North (ib == 3), West (ib == 4)
  347:
  348: void AddBound(int const ib, int const nx, int const ny, double w[], double const s[]
)
  349: {
  350:     int const dy = nx + 1,
  351:               bl  = 0,
  352:               br  = nx,
  353:               tl  = ny * (nx + 1),
  354:               tr  = (ny + 1) * (nx + 1) - 1;
  355:     switch (ib)
  356:     {
  357:         case 1:
  358:         {
  359:             for (int i = bl, j = 0; i <= br; ++i, ++j)
  360:                 w[i] += s[j];
  361:             break;
  362:         }
  363:         case 3:
  364:         {
  365:             for (int i = tl, j = 0; i <= tr; ++i, ++j)
  366:                 w[i] += s[j];
  367:             break;
  368:         }
  369:         case 4:
  370:         {
  371:             for (int i = bl, j = 0; i <= tl; i += dy, ++j)
  372:                 w[i] += s[j];
  373:             break;
  374:         }
  375:         case 2:
  376:         {
  377:             for (int i = br, j = 0; i <= tr; i += dy, ++j)
  378:                 w[i] += s[j];
  379:             break;
  380:         }
  381:         default:
  382:         {
  383:             cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__
 << endl;
  384:         }
  385:     }
  386:     return;
```

```
387: }
388:
389:
390: // ########################################################################
391:
392: Mesh_2d_3_matlab::Mesh_2d_3_matlab(string const &fname)
393:     : Mesh(2, 3, 3),  // two dimensions, 3 vertices, 3 dofs
394:       bedges(0)
395: {
396:     ifstream ifs(fname);
397:     if (!(ifs.is_open() && ifs.good()))
398:     {
399:         cerr << "Mesh_2d_3_matlab: Error cannot open file " << fname << endl;
400:         assert(ifs.is_open());
401:     }
402:
403:     int const OFFSET(1);              // Matlab to C indexing
404:     cout << "ASCI file  " << fname << "  opened" << endl;
405:
406:     // Read some mesh constants
407:     int nnode, ndim, nelem, nvert_e;
408:     ifs >> nnode >> ndim >> nelem >> nvert_e;
409:     cout << nnode << "  " << ndim << "  " << nelem << "  " << nvert_e << endl;
410:     assert(ndim == 2 && nvert_e == 3);
411:
412:     // Allocate memory
413:     Resize_Coords(nnode, ndim);                  // coordinates in 2D [nnode][ndim]
414:     Resize_Connectivity(nelem, nvert_e);         // connectivity matrix [nelem][nvert
]
415:
416:     // Read ccordinates
417:     auto &xc = GetCoords();
418:     for (int k = 0; k < nnode * ndim; ++k)
419:     {
420:         ifs >> xc[k];
421:     }
422:
423:     // Read connectivity
424:     auto &ia = GetConnectivity();
425:     for (int k = 0; k < nelem * nvert_e; ++k)
426:     {
427:         ifs >> ia[k];
428:         ia[k] -= OFFSET;                 // Matlab to C indexing
429:     }
430:
431:     // additional read of boundary information (only start/end point)
432:     int nbedges;
433:     ifs >> nbedges;
434:
435:     bedges.resize(nbedges * 2);
436:     for (int k = 0; k < nbedges * 2; ++k)
437:     {
438:         ifs >> bedges[k];
439:         bedges[k] -= OFFSET;             // Matlab to C indexing
440:     }
441:
442:     return;
443: }
444:
445: // binary
446: //{
447: //ifstream ifs(fname, ios_base::in | ios_base::binary);
448: //if(!(ifs.is_open() && ifs.good()))
449: //{
450: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
451: //assert(ifs.is_open());
452: //}
453: //cout << "ReadBinMatrix: file opened" << file << endl;
```

```
  454:
  455:
  456: //}
  457:
  458: // binaryIO.cpp
  459: //void read_binMatrix(const string& file, vector<int> &cnt, vector<int> &col, vector
<double> &ele)
  460: //{
  461:
  462: //ifstream ifs(file, ios_base::in | ios_base::binary);
  463:
  464: //if(!(ifs.is_open() && ifs.good()))
  465: //{
  466: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
  467: //assert(ifs.is_open());
  468: //}
  469: //cout << "ReadBinMatrix: Opened file " << file << endl;
  470:
  471: //int _size;
  472:
  473: //ifs.read(reinterpret_cast<char*>(&_size), sizeof(int));   // old: ifs.read((char*)
&_size, sizeof(int));
  474: //cnt.resize(_size);
  475: //cout << "ReadBinMatrix: cnt size: " << _size << endl;
  476:
  477: //ifs.read((char*)&_size, sizeof(int));
  478: //col.resize(_size);
  479: //cout << "ReadBinMatrix: col size: " << _size << endl;
  480:
  481: //ifs.read((char*)&_size, sizeof(int));
  482: //ele.resize(_size);
  483: //cout << "ReadBinMatrix: ele size: " << _size << endl;
  484:
  485:
  486: //ifs.read((char*)cnt.data(), cnt.size() * sizeof(int));
  487: //ifs.read((char*)col.data(), col.size() * sizeof(int));
  488: //ifs.read((char*)ele.data(), ele.size() * sizeof(double));
  489:
  490: //ifs.close();
  491: //cout << "ReadBinMatrix: Finished reading matrix.." << endl;
  492:
  493: //}
  494:
  495:
  496: std::vector<int> Mesh_2d_3_matlab::Index_DirichletNodes() const
  497: {
  498:     vector<int> idx(bedges);                                  // copy
  499:
  500:     sort(idx.begin(), idx.end());                             // sort
  501:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
  502:
  503:     return idx;
  504: }
  505:
  506:
  507:
  508:
  509:
  510:
  511:
  512:
  513:
  514:
  515:
  516:
  517:
  518:
  519:
```

```
  520:
  521:
  522:
  523:
```

```
    1: #ifndef GEOM_FILE
    2: #define GEOM_FILE
    3: #include <array>
    4: #include <functional>             // function; C++11
    5: #include <string>
    6: #include <vector>
    7:
    8: /**
    9:  * Basis class for finite element meshes.
   10:  */
   11: class Mesh
   12: {
   13: public:
   14:     /**
   15:      * Constructor initializing the members with default values.
   16:      *
   17:      * @param[in] ndim  space dimensions (dimension for coordinates)
   18:      * @param[in] nvert_e  number of vertices per element (dimension for connectivi
ty)
   19:      * @param[in] ndof_e   degrees of freedom per element (= @p nvert_e for linear
elements)
   20:      */
   21:     explicit Mesh(int ndim, int nvert_e = 0, int ndof_e = 0);
   22:
   23:     /**
   24:      * Destructor.
   25:      *
   26:      * See clang warning on
   27:      * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-vi
rtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
   28:      */
   29:     virtual ~Mesh();
   30:
   31:     /**
   32:      * Number of finite elements in (sub)domain.
   33:      * @return number of elements.
   34:      */
   35:     int Nelems() const
   36:     {
   37:         return _nelem;
   38:     }
   39:
   40:     /**
   41:      * Global number of vertices for each finite element.
   42:      * @return number of vertices per element.
   43:      */
   44:     int NverticesElements() const
   45:     {
   46:         return _nvert_e;
   47:     }
   48:
   49:     /**
   50:      * Global number of degrees of freedom (dof) for each finite element.
   51:      * @return degrees of freedom per element.
   52:      */
   53:     int NdofsElement() const
   54:     {
   55:         return _ndof_e;
   56:     }
   57:
   58:     /**
   59:      * Number of vertices in mesh.
   60:      * @return number of vertices.
   61:      */
   62:     int Nnodes() const
   63:     {
   64:         return _nnode;
   65:     }
```

```
   66:
   67:     /**
   68:      * Space dimension.
   69:      * @return number of dimensions.
   70:      */
   71:     int Ndims() const
   72:     {
   73:         return _ndim;
   74:     }
   75:
   76:     /**
   77:      * (Re-)Allocates memory for the element connectivity and redefines the appropri
ate dimensions.
   78:       *
   79:      * @param[in] nelem    number of elements
   80:       * @param[in] nvert_e  number of vertices per element
   81:      */
   82:     void Resize_Connectivity(int nelem, int nvert_e)
   83:     {
   84:         SetNelem(nelem);               // number of elements
   85:         SetNverticesElement(nvert_e);  // vertices per element
   86:         _ia.resize(nelem * nvert_e);
   87:     }
   88:
   89:     /**
   90:      * Read connectivity information (g1,g2,g3)_i.
   91:      * @return convectivity vector [nelems*ndofs].
   92:      */
   93:     const std::vector<int>  &GetConnectivity() const
   94:     {
   95:         return _ia;
   96:     }
   97:
   98:     /**
   99:      * Access/Change connectivity information (g1,g2,g3)_i.
  100:      * @return convectivity vector [nelems*ndofs].
  101:      */
  102:     std::vector<int>  &GetConnectivity()
  103:     {
  104:         return _ia;
  105:     }
  106:
  107:     /**
  108:      * (Re-)Allocates memory for the element connectivity and redefines the appropri
ate dimensions.
  109:       *
  110:      * @param[in] nnodes    number of nodes
  111:       * @param[in] ndim      space dimension
  112:      */
  113:     void Resize_Coords(int nnodes, int ndim)
  114:     {
  115:         SetNnode(nnodes);       // number of nodes
  116:         SetNdim(ndim);          // space dimension
  117:         _xc.resize(nnodes * ndim);
  118:     }
  119:
  120:     /**
  121:      * Read coordinates of vertices (x,y)_i.
  122:      * @return coordinates vector [nnodes*2].
  123:      */
  124:     const std::vector<double> &GetCoords() const
  125:     {
  126:         return _xc;
  127:     }
  128:
  129:     /**
  130:      * Access/Change coordinates of vertices (x,y)_i.
  131:      * @return coordinates vector [nnodes*2].
```

```
  132:      */
  133:     std::vector<double> &GetCoords()
  134:     {
  135:         return _xc;
  136:     }
  137:
  138:     /**
  139:      * Calculate values in vector @p v via function @p func(x,y)
  140:      * @param[in] v     vector
  141:       * @param[in] func  function of (x,y) returning a double value.
  142:      */
  143:     void SetValues(std::vector<double> &v, const std::function<double(double, double
)> &func) const;
  144:
  145:     /**
  146:      * Prints the information for a finite element mesh
  147:      */
  148:     void Debug() const;
  149:
  150:     /**
  151:      * Determines the indices of those vertices with Dirichlet boundary conditions
  152:      * @return index vector.
  153:      */
  154:     virtual std::vector<int> Index_DirichletNodes() const = 0;
  155:
  156:     /**
  157:      * Write vector @p v toghether with its mesh information to an ASCii file @p fna
me.
  158:       *
  159:       * The data are written in C-style.
  160:       *
  161:       * @param[in] fname  file name
  162:       * @param[in] v      vector
  163:      */
  164:     void Write_ascii_matlab(std::string const &fname, std::vector<double> const &v)
const;
  165:
  166:     /**
  167:      * Visualize @p v together with its mesh information via matlab or octave.
  168:       *
  169:       * Comment/uncomment those code lines in method Mesh:Visualize (geom.cpp)
  170:       * that are supported on your system.
  171:       *
  172:       * @param[in] v    vector
  173:       *
  174:       * @warning matlab files ascii_read_meshvector.m  visualize_results.m
  175:       *          must be in the executing directory.
  176:      */
  177:     void Visualize(std::vector<double> const &v) const;
  178:
  179:
  180: protected:
  181:     void SetNelem(int nelem)
  182:     {
  183:         _nelem = nelem;
  184:     }
  185:
  186:     void SetNverticesElement(int nvert)
  187:     {
  188:         _nvert_e = nvert;
  189:     }
  190:
  191:     void SetNdofsElement(int ndof)
  192:     {
  193:         _ndof_e = ndof;
  194:     }
  195:
  196:     void SetNnode(int nnode)
```

```
  197:     {
  198:         _nnode = nnode;
  199:     }
  200:
  201:     void SetNdim(int ndim)
  202:     {
  203:         _ndim = ndim;
  204:     }
  205:
  206: private:
  207:     int _nelem;         //!< number elements
  208:     int _nvert_e;       //!< number of vertices per element
  209:     int _ndof_e;        //!< degrees of freedom (d.o.f.) per element
  210:     int _nnode;         //!< number nodes/vertices
  211:     int _ndim;          //!< space dimension of the problem (1, 2, or 3)
  212:     std::vector<int> _ia;    //!< element connectivity
  213:     std::vector<double> _xc; //!< coordinates
  214: };
  215:
  216: /**
  217:  * 2D finite element mesh of the square consiting of linear triangular elements.
  218:  */
  219: class Mesh_2d_3_square: public Mesh
  220: {
  221: public:
  222:     /**
  223:      * Generates the f.e. mesh for the unit square.
  224:      *
  225:      * @param[in] nx    number of discretization intervals in x-direction
  226:      * @param[in] ny    number of discretization intervals in y-direction
  227:      * @param[in] myid  my MPI-rank / subdomain
  228:      * @param[in] procx number of ranks/subdomains in x-direction
  229:      * @param[in] procy number of processes in y-direction
  230:      */
  231:     Mesh_2d_3_square(int nx, int ny, int myid = 0, int procx = 1, int procy = 1);
  232:
  233:     /**
  234:      * Destructor
  235:      */
  236:     ~Mesh_2d_3_square() override
  237:     {}
  238:
  239:     /**
  240:      * Set solution vector based on a tensor product grid in the rectangle.
  241:      * @param[in] u solution vector
  242:      */
  243:     void SetU(std::vector<double> &u) const;
  244:
  245:     /**
  246:      * Set right hand side (rhs) vector on a tensor product grid in the rectangle.
  247:      * @param[in] f rhs vector
  248:      */
  249:     void SetF(std::vector<double> &f) const;
  250:
  251:     /**
  252:      * Determines the indices of those vertices with Dirichlet boundary conditions
  253:      * @return index vector.
  254:      */
  255:     std::vector<int> Index_DirichletNodes() const override;
  256:
  257:     /**
  258:       * Stores the values of vector @p u of (sub)domain into a file @p name for furt
her processing in gnuplot.
  259:       * The file stores rowise the x- and y- coordinates together with the value fro
m  @p u .
  260:       * The domain [@p xl, @p xr] x [@p yb, @p yt] is discretized into @p nx x @p ny
 intervals.
  261:       *
```

```
  262:       * @param[in] name basename of file name (file name will be extended by the ran
k number)
  263:       * @param[in] u    local vector
  264:       *
  265:       * @warning   Assumes tensor product grid in unit square; rowise numbered
  266:       *            (as generated in class constructor).
  267:       *            The output is provided for tensor product grid visualization
  268:       *            ( similar to Matlab-surf() ).
  269:       *
  270:       * @see Mesh_2d_3_square
  271:       */
  272:     void SaveVectorP(std::string const &name, std::vector<double> const &u) const;
  273:
  274:     // here will still need to implement in the class
  275:     //  GetBound(), AddBound()
  276:     //  or better a generalized way with indices and their appropriate ranks for MPI
 communication
  277:
  278: private:
  279:     /**
  280:       * Determines the coordinates of the dicretization nodes of the domain [@p xl,
@p xr] x [@p yb, @p yt]
  281:       * which is discretized into @p nx x @p ny intervals.
  282:       *
  283:       * @param[in] ny   number of discretization intervals in y-direction
  284:       * @param[in] xl   x-coordinate of left boundary
  285:       * @param[in] xr   x-coordinate of right boundary
  286:       * @param[in] yb   y-coordinate of lower boundary
  287:       * @param[in] yt   y-coordinate of upper boundary
  288:       * @param[out] xc  coordinate vector of length 2n with x(2*k,2*k+1) as coodinat
es of node k
  289:       */
  290:
  291:     void GetCoordsInRectangle(int nx, int ny, double xl, double xr, double yb, doubl
e yt,
  292:                               double xc[]);
  293:     /**
  294:       * Determines the element connectivity of linear triangular elements of a FEM d
iscretization
  295:       * of a rectangle using @p nx x @p ny equidistant intervals for discretization.
  296:       * @param[in] nx   number of discretization intervals in x-direction
  297:       * @param[in] ny   number of discretization intervals in y-direction
  298:       * @param[out] ia  element connectivity matrix with ia(3*s,3*s+1,3*s+2) as node
 numbers od element s
  299:       */
  300:     void GetConnectivityInRectangle(int nx, int ny, int ia[]);
  301:
  302: private:
  303:     int _myid;          //!< my MPI rank
  304:     int _procx;         //!< number of MPI ranks in x-direction
  305:     int _procy;         //!< number of MPI ranks in y-direction
  306:     std::array<int, 4> _neigh; //!< MPI ranks of neighbors (negative: no neighbor bu
t b.c.)
  307:     int _color;         //!< red/black coloring (checker board) of subdomains
  308:
  309:     double _xl;         //!< x coordinate of lower left  corner of square
  310:     double _xr;         //!< x coordinate of lower right corner of square
  311:     double _yb;         //!< y coordinate or lower left  corner of square
  312:     double _yt;         //!< y coordinate of upper right corner of square
  313:     int    _nx;         //!< number of intervals in x-direction
  314:     int    _ny;         //!< number of intervals in y-direction
  315: };
  316:
  317: // ################### still some old code (--> MPI) ############################
  318: /**
  319:  * Copies the values of @p w corresponding to boundary @p ib
  320:  * onto vector s.  South (ib==1), East (ib==2), North (ib==3), West (ib==4).
  321:  * The vector @p s has to be long enough!!
```

```
  322:  * @param[in] ib    my local boundary
  323:  * @param[in] nx    number of discretization intervals in x-direction
  324:  * @param[in] ny    number of discretization intervals in y-direction
  325:  * @param[in] w     vector for all nodes of local discretization
  326:  * @param[out] s    short vector with values on boundary @p ib
  327: */
  328: // GH_NOTE: Absicherung bei s !!
  329: void GetBound(int ib, int nx, int ny, double const w[], double s[]);
  330:
  331:
  332: /**
  333:  * Computes @p w := @p w + @p s  at the interface/boundary nodes on the
  334:  * boundary @p ib .  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
  335:  * @param[in] ib    my local boundary
  336:  * @param[in] nx    number of discretization intervals in x-direction
  337:  * @param[in] ny    number of discretization intervals in y-direction
  338:  * @param[in,out] w vector for all nodes of local discretization
  339:  * @param[in] s     short vector with values on boundary @p ib
  340: */
  341: void AddBound(int ib, int nx, int ny, double w[], double const s[]);
  342:
  343: // ################### Mesh from Matlab ############################
  344: /**
  345:  * 2D finite element mesh of the square consiting of linear triangular elements.
  346:  */
  347: class Mesh_2d_3_matlab: public Mesh
  348: {
  349: public:
  350:     /**
  351:      * Reads mesh data from a binary file.
  352:      *
  353:      * File format, see ascii_write_mesh.m
  354:      *
  355:      * @param[in] fname file name
  356:     */
  357:     explicit Mesh_2d_3_matlab(std::string const &fname);
  358:
  359:     /**
  360:      * Determines the indices of those vertices with Dirichlet boundary conditions.
  361:      * @return index vector.
  362:       *
  363:       * @warning All boundary nodes are considered as Dirchlet nodes.
  364:      */
  365:     std::vector<int> Index_DirichletNodes() const override;
  366:
  367: private:
  368:     /**
  369:      * Determines the indices of those vertices with Dirichlet boundary conditions
  370:      * @return index vector.
  371:      */
  372:     int Nnbedges() const
  373:     {
  374:         return static_cast<int>(bedges.size());
  375:     }
  376:
  377:     std::vector<int> bedges;     //!< boundary edges [nbedges][2] storing start/end
vertex
  378:
  379: };
  380:
  381: #endif
```

```cpp
#include "getmatrix.h"
#include "userset.h"

#include <algorithm>
#include <cassert>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <list>
#include <vector>
using namespace std;


//  general routine for lin. triangular elements

void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3])
//void CalcElem(const int* __restrict__ ial, const double* __restrict__ xc, double*
__restrict__ ske[3], double* __restrict__ fe)
{
    const int  i1  = 2 * ial[0],   i2 = 2 * ial[1],   i3 = 2 * ial[2];
    const double x13 = xc[i3 + 0] - xc[i1 + 0],  y13 = xc[i3 + 1] - xc[i1 + 1],
                 x21 = xc[i1 + 0] - xc[i2 + 0],  y21 = xc[i1 + 1] - xc[i2 + 1],
                 x32 = xc[i2 + 0] - xc[i3 + 0],  y32 = xc[i2 + 1] - xc[i3 + 1];
    const double jac = fabs(x21 * y13 - x13 * y21);

    ske[0][0] = 0.5 / jac * (y32 * y32 + x32 * x32);
    ske[0][1] = 0.5 / jac * (y13 * y32 + x13 * x32);
    ske[0][2] = 0.5 / jac * (y21 * y32 + x21 * x32);
    ske[1][0] = ske[0][1];
    ske[1][1] = 0.5 / jac * (y13 * y13 + x13 * x13);
    ske[1][2] = 0.5 / jac * (y21 * y13 + x21 * x13);
    ske[2][0] = ske[0][2];
    ske[2][1] = ske[1][2];
    ske[2][2] = 0.5 / jac * (y21 * y21 + x21 * x21);

    const double xm    = (xc[i1 + 0] + xc[i2 + 0] + xc[i3 + 0]) / 3.0,
                 ym    = (xc[i1 + 1] + xc[i2 + 1] + xc[i3 + 1]) / 3.0;
    //fe[0] = fe[1] = fe[2] = 0.5 * jac * FunctF(xm, ym) / 3.0;
    fe[0] = fe[1] = fe[2] = 0.5 * jac * fNice(xm, ym) / 3.0;
}


// general routine for lin. triangular elements,
// non-symm. matrix
// node numbering in element:  a s c e n d i n g   indices !!
[[deprecated("Use CRS_Matrix::AddElem_3(...) instead.")]]
void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
             int const id[], int const ik[], double sk[], double f[])
{
    for (int i = 0; i < 3; ++i)
    {
        const int ii  = ial[i],           // row ii (global index)
                  id1 = id[ii],           // start and
                  id2 = id[ii + 1];       // end of row ii in matrix
        int ip  = id1;
        for (int j = 0; j < 3; ++j)       // no symmetry assumed
        {
            const int jj = ial[j];
            bool not_found = true;
            do       // find entry jj (global index) in row ii
            {
                not_found = (ik[ip] != jj);
                ++ip;
            }
            while (not_found && ip < id2);

#ifndef NDEBUG                  // compiler option -DNDEBUG switches off the check
            if (not_found)      // no entry found !!
```

```
   68:             {
   69:                 cout << "Error in AddElem: (" << ii << "," << jj << ") ["
   70:                      << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
   71:                 assert(!not_found);
   72:             }
   73: #endif
   74:             sk[ip - 1] += ske[i][j];
   75:         }
   76:         f[ii] += fe[i];
   77:     }
   78: }
   79:
   80:
   81: // ----------------------------------------------------------------------------
   82:
   83:
   84:
   85:
   86: // ############################################################################
   87:
   88: CRS_Matrix::CRS_Matrix(Mesh const &mesh)
   89:     : _mesh(mesh), _nrows(0), _nnz(0), _id(0), _ik(0), _sk(0)
   90: {
   91:     Derive_Matrix_Pattern();
   92:     return;
   93: }
   94:
   95: void CRS_Matrix::Derive_Matrix_Pattern()
   96: {
   97:     int const nelem(_mesh.Nelems());
   98:     int const ndof_e(_mesh.NdofsElement());
   99:     auto const &ia(_mesh.GetConnectivity());
  100: //  Determine the number of matrix rows
  101:     _nrows = *max_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem);
  102:     ++_nrows;                                   // node numberng: 0 ... nnode-1
  103:     assert(*min_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem) == 0); // numberi
ng starts with 0 ?
  104:
  105: //  Collect for each node those nodes it is connected to (multiple entries)
  106: //  Detect the neighboring nodes
  107:     vector< list<int> > cc(_nrows);             //  cc[i] is the  list of nodes a no
de i is connected to
  108:     for (int i = 0; i < nelem; ++i)
  109:     {
  110:         int const idx = ndof_e * i;
  111:         for (int k = 0; k < ndof_e; ++k)
  112:         {
  113:             list<int> &cck = cc.at(ia[idx + k]);
  114:             cck.insert( cck.end(), ia.cbegin() + idx, ia.cbegin() + idx + ndof_e );
  115:         }
  116:     }
  117: //  Delete the multiple entries
  118:     _nnz = 0;
  119:     for (auto &it : cc)
  120:     {
  121:         it.sort();
  122:         it.unique();
  123:         _nnz += static_cast<int>(it.size());
  124:         // cout << it.size() << " :: "; copy(it->begin(),it->end(), ostream_iterator
<int,char>(cout,"  ")); cout << endl;
  125:     }
  126:
  127: // CSR data allocation
  128:     _id.resize(_nrows + 1);                     // Allocate memory for CSR row pointer
  129:     _ik.resize(_nnz);                           // Allocate memory for CSR column index
 vector
  130:
  131: //  copy CSR data
```

```
  132:     _id[0] = 0;                                    // begin of first row
  133:     for (size_t i = 0; i < cc.size(); ++i)
  134:     {
  135:         //cout << i << "   " << nid.at(i) << endl;;
  136:         const list<int> &ci = cc.at(i);
  137:         const auto nci = static_cast<int>(ci.size());
  138:         _id[i + 1] = _id[i] + nci; // begin of next line
  139:         copy(ci.begin(), ci.end(), _ik.begin() + _id[i] );
  140:     }
  141:
  142:     assert(_nnz == _id[_nrows]);
  143:     _sk.resize(_nnz);                          // Allocate memory for CSR column index v
ector
  144:     return;
  145: }
  146:
  147:
  148: void CRS_Matrix::Debug() const
  149: {
  150: //  ID points to first entry of row
  151: //  no symmetry assumed
  152:     cout << "\nMatrix  (nnz = " << _id[_nrows] << ")\n";
  153:
  154:     for (int row = 0; row < _nrows; ++row)
  155:     {
  156:         cout << "Row " << row << " : ";
  157:         int const id1 = _id[row];
  158:         int const id2 = _id[row + 1];
  159:         for (int j = id1; j < id2; ++j)
  160:         {
  161:             cout.setf(ios::right, ios::adjustfield);
  162:             cout << "[" << setw(2) << _ik[j] << "]  " << setw(4) << _sk[j] << "  ";
  163:         }
  164:         cout << endl;
  165:     }
  166:     return;
  167: }
  168:
  169: void CRS_Matrix::CalculateLaplace(vector<double> &f)
  170: {
  171:     assert(_mesh.NdofsElement() == 3);              // only for triangular, linear
elements
  172:     //cout << _nnz << " vs. " << _id[_nrows] << "  " << _nrows<< endl;
  173:     assert(_nnz == _id[_nrows]);
  174:
  175:     for (int k = 0; k < _nrows; ++k)
  176:     {
  177:         _sk[k] = 0.0;
  178:     }
  179:     for (int k = 0; k < _nrows; ++k)
  180:     {
  181:         f[k] = 0.0;
  182:     }
  183:
  184:     //  Loop over all elements
  185:     auto const nelem = _mesh.Nelems();
  186:     auto const &ia   = _mesh.GetConnectivity();
  187:     auto const &xc   = _mesh.GetCoords();
  188:
  189:     #pragma omp parallel for
  190:     for (int i = 0; i < nelem; ++i)
  191:     {
  192:         double ske[3][3], fe[3];
  193:         CalcElem(ia.data() + 3 * i, xc.data(), ske, fe);
  194:         #pragma omp critical
  195:         AddElem_3(ia.data() + 3 * i, ske, fe, f);
  196:     }
  197:     //Debug();
```

data race avoided, but slower !?

```
  198:
  199:     return;
  200: }
  201:
  202: void CRS_Matrix::ApplyDirichletBC(std::vector<double> const &u, std::vector<double>
&f)
  203: {
  204:     double const PENALTY = 1e6;
  205:     auto const idx = _mesh.Index_DirichletNodes();
  206:     int const nidx = static_cast<int>(idx.size());
  207:
  208:     #pragma omp parallel for
  209:     for (int row = 0; row < nidx; ++row)
  210:     {
  211:         int const k = idx[row];
  212:         int const id1 = fetch(k, k); // Find diagonal entry of row
  213:         assert(id1 >= 0);
  214:         _sk[id1] += PENALTY;            // matrix weighted scaling feasible
  215:         f[k] += PENALTY * u[k];
  216:     }
  217:
  218:     return;
  219: }
  220:
  221: void CRS_Matrix::GetDiag(vector<double> &d) const
  222: {
  223:     assert( _nrows == static_cast<int>(d.size()) );
  224:
  225:     for (int row = 0; row < _nrows; ++row)
  226:     {
  227:         const int ia = fetch(row, row); // Find diagonal entry of row
  228:         assert(ia >= 0);
  229:         d[row] = _sk[ia];
  230:     }
  231:     return;
  232: }
  233:
  234: bool CRS_Matrix::Compare2Old(int nnode, int const id[], int const ik[], double const
 sk[]) const
  235: {
  236:     bool bn = (nnode == _nrows);     // number of rows
  237:     if (!bn)
  238:     {
  239:         cout << "#########   Error: " << "number of rows" << endl;
  240:     }
  241:
  242:     bool bz = (id[nnode] == _nnz);   // number of non zero elements
  243:     if (!bz)
  244:     {
  245:         cout << "#########   Error: " << "number of non zero elements" << endl;
  246:     }
  247:
  248:     bool bd = equal(id, id + nnode + 1, _id.cbegin()); // row starts
  249:     if (!bd)
  250:     {
  251:         cout << "#########   Error: " << "row starts" << endl;
  252:     }
  253:
  254:     bool bk = equal(ik, ik + id[nnode], _ik.cbegin()); // column indices
  255:     if (!bk)
  256:     {
  257:         cout << "#########   Error: " << "column indices" << endl;
  258:     }
  259:
  260:     bool bv = equal(sk, sk + id[nnode], _sk.cbegin()); // values
  261:     if (!bv)
  262:     {
  263:         cout << "#########   Error: " << "values" << endl;
```

nicht parallel

```
  264:     }
  265:
  266:     return bn && bz && bd && bk && bv;
  267: }
  268:
  269:
  270: void CRS_Matrix::Mult(vector<double> &w, vector<double> const &u) const
  271: {
  272:     assert( _nrows == static_cast<int>(w.size()) );
  273:     assert( w.size() == u.size() );
  274:
  275:     for (int row = 0; row < _nrows; ++row)
  276:     {
  277:         double wi = 0.0;
  278:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
  279:         {
  280:             wi += _sk[ij] * u[ _ik[ij] ];
  281:         }
  282:         w[row] = wi;
  283:     }
  284:     return;
  285: }
  286:
  287: void CRS_Matrix::Defect(vector<double> &w,
  288:                         vector<double> const &f, vector<double> const &u) const
  289: {
  290:     assert( _nrows == static_cast<int>(w.size()) );
  291:     assert( w.size() == u.size() && u.size() == f.size() );
  292:
  293:     #pragma omp parallel for
  294:     for (int row = 0; row < _nrows; ++row)
  295:     {
  296:         double wi = f[row];
  297:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
  298:         {
  299:             wi -= _sk[ij] * u[ _ik[ij] ];
  300:         }
  301:         w[row] = wi;
  302:     }
  303:     return;
  304: }
  305:
  306: int CRS_Matrix::fetch(int const row, int const col) const
  307: {
  308:     int const id2 = _id[row + 1];     // end   and
  309:     int       ip  = _id[row];         // start of recent row (global index)
  310:
  311:     while (ip < id2 && _ik[ip] != col)  // find index col (global index)
  312:     {
  313:         ++ip;
  314:     }
  315:     if (ip >= id2)
  316:     {
  317:         ip = -1;
  318: #ifndef NDEBUG                  // compiler option -DNDEBUG switches off the check
  319:         cout << "No column  " << col << "  in row  " << row << endl;
  320:         assert(ip >= id2);
  321: #endif
  322:     }
  323:     return ip;
  324: }
  325:
  326:
  327: // general routine for lin. triangular elements,
  328: // non-symm. matrix
  329: // node numbering in element:  a s c e n d i n g   indices !!
  330: void CRS_Matrix::AddElem_3(int const ial[3], double const ske[3][3], double const fe
[3], vector<double> &f)
```

```
331: {
332:     for (int i = 0; i < 3; ++i)
333:     {
334:         const int ii  = ial[i];           // row ii (global index)
335:         for (int j = 0; j < 3; ++j)       // no symmetry assumed
336:         {
337:             const int jj = ial[j];        // column jj (global index)
338:             int ip = fetch(ii, jj);       // find column entry jj in row ii
339: #ifndef NDEBUG                            // compiler option -DNDEBUG switches off the check
340:             if (ip < 0)                   // no entry found !!
341:             {
342:                 cout << "Error in AddElem: (" << ii << "," << jj << ") ["
343:                      << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
344:                 assert(ip >= 0);
345:             }
346: #endif
347:             _sk[ip] += ske[i][j];
348:         }
349:         f[ii] += fe[i];
350:     }
351: }
352:
```

atomic

```
    1: #ifndef GETMATRIX_FILE
    2: #define GETMATRIX_FILE
    3:
    4: #include "geom.h"
    5: #include <vector>
    6:
    7: /**
    8:  * Calculates the element stiffness matrix @p ske and the element load vector @p fe
    9:  * of one triangular element with linear shape functions.
   10:  * @param[in]   ial     node indices of the three element vertices
   11:  * @param[in]   xc      vector of node coordinates with x(2*k,2*k+1) as coodinates o
f node k
   12:  * @param[out] ske      element stiffness matrix
   13:  * @param[out] fe       element load vector
   14:  */
   15: void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3]);
   16:
   17: /**
   18:  * Adds the element stiffness matrix @p ske and the element load vector @p fe
   19:  * of one triangular element with linear shape functions to the appropriate position
s in
   20:  * the symmetric stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik)
   21:  *
   22:  * @param[in] ial   node indices of the three element vertices
   23:  * @param[in] ske       element stiffness matrix
   24:  * @param[in] fe        element load vector
   25:  * @param[out] sk       vector non-zero entries of CSR matrix
   26:  * @param[in] id        index vector containing the first entry in a CSR row
   27:  * @param[in] ik        column index vector of CSR matrix
   28:  * @param[out] f        distributed local vector storing the right hand side
   29:  *
   30:  * @warning Algorithm requires indices in connectivity @p ial in ascending order.
   31:  *          Currently deprecated.
   32: */
   33: void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
   34:              int const id[], int const ik[], double sk[], double f[]);
   35:
   36:
   37: // ########################################################################
   38: /**
   39:  * Square matrix in CRS format (compressed row storage; also named CSR),
   40:  * see an <a href="https://en.wikipedia.org/wiki/Sparse_matrix">introduction</a>.
   41:  */
   42: class CRS_Matrix
   43: {
   44:     public:
   45:        /**
   46:         * Intializes the CRS matrix structure from the given discetization in @p mes
h.
   47:         *
   48:         * The sparse matrix pattern is generated but the values are 0.
   49:         *
   50:         * @param[in] mesh given discretization
   51:         *
   52:         * @warning A reference to the discretization @p mesh is stored inside this c
lass.
   53:         *          Therefore, changing @p mesh outside requires also
   54:         *          to call method @p Derive_Matrix_Pattern explicitely.
   55:         *
   56:         * @see Derive_Matrix_Pattern
   57:         */
   58:        explicit CRS_Matrix(Mesh const & mesh);
   59:
   60:       /**
   61:         * Destructor.
   62:         */
   63:        ~CRS_Matrix()
   64:        {}
```

```
   65:
   66:         /**
   67:          * Generates the sparse matrix pattern and overwrites the existing pattern.
   68:          *
   69:          * The sparse matrix pattern is generated but the values are 0.
   70:          */
   71:         void Derive_Matrix_Pattern();
   72:
   73:          /**
   74:          * Calculates the entries of f.e. stiffness matrix and load/rhs vector @p f f
or the Laplace operator in 2D.
   75:          * No memory is allocated.
   76:          *
   77:          * @param[in,out] f (preallocated) rhs/load vector
   78:          */
   79:         void CalculateLaplace(std::vector<double> &f);
   80:
   81:         /**
   82:          * Applies Dirichlet boundary conditions to stiffness matrix and to load vect
or @p f.
   83:          * The <a href="https://www.jstor.org/stable/2005611?seq=1#metadata_info_tab_
contents">penalty method</a>
   84:          * is used for incorporating the given values @p u.
   85:          *
   86:          * @param[in]     u (global) vector with Dirichlet data
   87:          * @param[in,out] f load vector
   88:          */
   89:         void ApplyDirichletBC(std::vector<double> const &u, std::vector<double> &f);
   90:
   91:         /**
   92:          * Extracts the diagonal elemenst of the sparse matrix.
   93:          *
   94:          * @param[in,out]  d  (prellocated) vector of diagonal elements
   95:          */
   96:         void GetDiag(std::vector<double> &d) const;
   97:
   98:         /**
   99:          * Performs the matrix-vector product  w := K*u.
  100:          *
  101:          * @param[in,out] w resulting vector (preallocated)
  102:          * @param[in]     u vector
  103:          */
  104:         void Mult(std::vector<double> &w, std::vector<double> const &u) const;
  105:
  106:          /**
  107:          * Calculates the defect/residuum w := f - K*u.
  108:          *
  109:          * @param[in,out] w resulting vector (preallocated)
  110:          * @param[in]     f load vector
  111:          * @param[in]     u vector
  112:          */
  113:         void Defect(std::vector<double> &w,
  114:                     std::vector<double> const &f, std::vector<double> const &u) const
;
  115:
  116:         /**
  117:                   * Number rows in matrix.
  118:                   * @return number of rows.
  119:                   */
  120:         int Nrows() const
  121:            {return _nrows;}
  122:
  123:         /**
  124:                   * Show the matrix entries.
  125:                   */
  126:         void Debug() const;
  127:
  128:         /**
```

```
  129:                  * Finds in a CRS matrix the access index for an entry at row @p row
 and column @p col.
  130:                  *
  131:                  * @param[in] row       row index
  132:                  * @param[in] col       column index
  133:                  * @return index for element (@p row, @p col). If no appropriate ent
ry exists then -1 will be returned.
  134:                  *
  135:                  * @warning assert() stops the function in case that matrix element
(@p row, @p col) doesn't exist.
  136:                 */
  137:         int fetch(int row, int col) const;
  138:
  139:        /**
  140:         * Adds the element stiffness matrix @p ske and the element load vector @p fe
  141:         * of one triangular element with linear shape functions to the appropriate p
ositions in
  142:         * the stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik).
  143:         *
  144:         * @param[in]     ial   node indices of the three element vertices
  145:         * @param[in]     ske   element stiffness matrix
  146:         * @param[in]     fe    element load vector
  147:         * @param[in,out] f         distributed local vector storing the right hand s
ide
  148:         *
  149:         * @warning Algorithm assumes  linear triangular elements (ndof_e==3).
  150:         */
  151:         void AddElem_3(int const ial[3], double const ske[3][3], double const fe[3],
std::vector<double> &f);
  152:
  153:         /**
  154:         * Compare @p this CRS matrix with an external CRS matrix stored in C-Style.
  155:         *
  156:         * The method prints statements on differences found.
  157:         *
  158:         * @param[in]     nnode  row number of external matrix
  159:         * @param[in]     id     start indices of matrix rows of external matrix
  160:         * @param[in]     ik     column indices of external matrix
  161:         * @param[in]     sk     non-zero values of external matrix
  162:         *
  163:         * @return true iff all data are identical.
  164:         */
  165:         bool Compare2Old(int nnode, int const id[], int const ik[], double const sk[]
) const;
  166:
  167:     private:
  168:         Mesh const & _mesh;       //!< reference to discretization
  169:         int _nrows;               //!< number of rows in matrix
  170:         int _nnz;                 //!< number of non-zero entries
  171:         std::vector<int> _id;     //!< start indices of matrix rows
  172:         std::vector<int> _ik;     //!< column indices
  173:         std::vector<double> _sk; //!< non-zero values
  174:
  175: };
  176:
  177:
  178: #endif
```

```cpp
#include "vdop.h"
#include "getmatrix.h"
#include "jacsolve.h"

#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

// ######################################################################
//      const int neigh[], const int color,
//      const MPI::Intracomm& icomm,
void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u)
{
    const double omega   = 1.0;
    const int    maxiter = 1000;
    const double tol  = 1e-5,                   // tolerance
                 tol2 = tol * tol;              // tolerance^2

    int nrows = SK.Nrows();                     // number of rows == number of columns
    assert( nrows == static_cast<int>(f.size()) && f.size() == u.size() );

    cout << endl << " Start Jacobi solver for " << nrows << " d.o.f.s"  << endl;
    //  Choose initial guess
    for (int k = 0; k < nrows; ++k)
    {
        u[k] = 0.0;                             //  u := 0
    }

    vector<double> dd(nrows);                   // matrix diagonal
    vector<double>  r(nrows);                   // residual
    vector<double>  w(nrows);                   // correction

    SK.GetDiag(dd);                             //  dd := diag(K)
    ////DebugVector(dd);{int ijk; cin >> ijk;}

    //  Initial sweep
    SK.Defect(r, f, u);                         //  r := f - K*u

    vddiv(w, r, dd);                            //  w := D^{-1}*r
    double sigma0 = dscapr(w, r);               // s0 := <w,r>

    // Iteration sweeps
    int iter  = 0;
    double sigma = sigma0;
    while ( sigma > tol2 * sigma0 && maxiter > iter)
    {
        ++iter;
        vdaxpy(u, u, omega, w );                //  u := u + om*w
        SK.Defect(r, f, u);                     //  r := f - K*u
        vddiv(w, r, dd);                        //  w := D^{-1}*r
        sigma = dscapr(w, r);                   // s0 := <w,r>
//              cout << "Iteration " << iter << " : " << sqrt(sigma/sigma0) << endl;
    }
    cout << "aver. Jacobi rate :  " << exp(log(sqrt(sigma / sigma0)) / iter) << "  (
" << iter << " iter)" << endl;
    cout << "final error: " << sqrt(sigma / sigma0) << " (rel)    " << sqrt(sigma) <<
 " (abs)\n";

    return;
}

```

```
    1: #ifndef JACSOLVE_FILE
    2: #define JACSOLVE_FILE
    3: #include "getmatrix.h"
    4: #include <vector>
    5:
    6:
    7: /**
    8:  * Solves linear system of equations  K @p u = @p f  via the Jacobi iteration.
    9:  * We use a distributed symmetric  CSR matrix @p SK and initial guess of the
   10:  * solution is set to 0.
   11:  * @param[in] SK        CSR matrix
   12:  * @param[in] f         distributed local vector storing the right hand side
   13:  * @param[out] u        accumulated local vector storing the solution.
   14: */
   15: void JacobiSolve(CRS_Matrix const &SK, std::vector<double> const &f, std::vector<dou
ble> &u);
   16:
   17:
   18: #endif
```

```
    1: // compile with      make
    2: // run with          ./main.GCC_ -n X^2       for an integer X
    3:
    4:
    5: //              MPI code in C++.
    6: //              See [Gropp/Lusk/Skjellum, "Using MPI", p.33/41 etc.]
    7: //              and  /opt/mpich/include/mpi2c++/comm.h  for details
    8:
    9: #include "geom.h"
   10: #include "getmatrix.h"
   11: #include "jacsolve.h"
   12: #include "userset.h"
   13: #include "vdop.h"
   14:
   15: #include <chrono>              // timing
   16: #include <cmath>
   17: #include <iostream>
   18: #include <omp.h>
   19: using namespace std;
   20: using namespace std::chrono;  // timing
   21:
   22:
   23: int main(int argc, char ** argv)
   24: {
   25:     int nthreads = 1;
   26:     if (argc > 2 && std::string(argv[1]) == "-n")
   27:         nthreads = std::stoi(argv[2]);
   28:     omp_set_num_threads(nthreads);
   29:
   30:     #pragma omp parallel
   31:     #pragma omp single
   32:     cout << "\n There are " << omp_get_num_threads() << " processes running.\n \n";
   33:
   34:     const auto procx = static_cast<int>(sqrt(omp_get_max_threads() + 0.0));
   35:     const int  procy = procx;
   36:
   37:     if (procy * procx != omp_get_max_threads())
   38:     {
   39:         cout << "\n Wrong number of processors !\n \n";
   40:     }
   41:     else
   42:     {
   43: // ######################################################################
   44: //      Here starts the real code
   45: // ######################################################################
   46:         //bool ScaleUp = !true;
   47:         int nx, ny, NXglob, NYglob; /* number of local intervals on (xl,xr)=:nx, (yb
,yt)=:ny */
   48:         //nx = 1024;
   49:         //ny = 1024;
   50:         nx = 100;
   51:         ny = 100;
   52:         NXglob = nx * procx;
   53:         NYglob = ny * procy;
   54:         cout << "Intervalls: " << NXglob << " x " << NYglob << endl;
   55:
   56: // ##################### STL ##########################################
   57:         {
   58:             Mesh_2d_3_square const mesh(nx, ny);
   59:             //mesh.Debug();
   60:
   61:             CRS_Matrix SK(mesh);                    // CRS matrix
   62:             //SK.Debug();
   63:             vector<double> uv(SK.Nrows(), 0.0);     // temperature
   64:             vector<double> fv(SK.Nrows(), 0.0);     // r.h.s.
   65:
   66:             SK.CalculateLaplace(fv);
   67:             //SK.Debug();
```

```
   68:
   69:             //mesh.SetU(uv);         // deprecated
   70:             //mesh.SetF(fv);         // deprecated
   71:             // Two ways to initialize the vector
   72:             //mesh.SetValues(uv,f_zero);             // functional
   73:             mesh.SetValues(uv, [](double x, double y) -> double {return 0.0 * x *y;}
 ); // lambda function
   74:
   75:             SK.ApplyDirichletBC(uv, fv);
   76:             //SK.Compare2Old(nnode, id, ik, sk);
   77:             //SK.Debug();
   78:
   79:             auto tstart = system_clock::now();       // start timer
   80:
   81:             JacobiSolve(SK, fv, uv );          // solve the system of equations
   82:
   83:             auto tend = system_clock::now();         // end timer
   84:             auto duration = duration_cast<microseconds>(tend - tstart);
   85:             auto t1 = static_cast<double>(duration.count()) / 1e6 ;   // t1 in secon
ds
   86:             cout << "JacobiSolve: timing in sec. : " << t1 << endl;
   87:
   88:             //CompareVectors(uv, nnode, u, 1e-6);    // Check correctness
   89:
   90:             //mesh.SaveVectorP("t.dat", uv);
   91:             //mesh.Visualize(uv);
   92:
   93:         }
   94: // ##################### STL ##########################################
   95:         {
   96:             //Mesh_2d_3_matlab const mesh("square_tiny.txt");
   97:             Mesh_2d_3_matlab const mesh("square_100.txt");
   98:             //Mesh_2d_3_matlab const mesh("L_shape.txt");
   99:             //mesh.Debug();
  100:
  101:             CRS_Matrix SK(mesh);                    // CRS matrix
  102:             //SK.Debug();
  103:
  104:             vector<double> uv(SK.Nrows(), 0.0);    // temperature
  105:             vector<double> fv(SK.Nrows(), 0.0);    // r.h.s.
  106:
  107:             SK.CalculateLaplace(fv);
  108:             //SK.Debug();
  109:
  110:             //mesh.SetU(uv);         // deprecated
  111:             // Two ways to initialize the vector
  112:             //mesh.SetValues(uv,f_zero);             // user function
  113:             mesh.SetValues(uv, [](double x, double y) -> double {return 0.0 * x *y;}
 ); // lambda function
  114:
  115:             SK.ApplyDirichletBC(uv, fv);
  116:             //SK.Compare2Old(nnode, id, ik, sk);
  117:             //SK.Debug();
  118:
  119:             auto tstart = system_clock::now();       // start timer
  120:
  121:             JacobiSolve(SK, fv, uv );          // solve the system of equations
  122:
  123:             auto tend = system_clock::now();         // end timer
  124:             auto duration = duration_cast<microseconds>(tend - tstart);
  125:             auto t1 = static_cast<double>(duration.count()) / 1e6 ;   // t1 in secon
ds
  126:             cout << "JacobiSolve: timing in sec. : " << t1 << endl;
  127:
  128:             //mesh.Write_ascii_matlab("uv.txt", uv);
  129:             //mesh.Visualize(uv);
  130:         }
  131:         return 0;
```

```
  132:     }
  133: }
  134:
  135:
```

```cpp
#include "userset.h"
#include <cmath>


double FunctF(double const x, double const y)
{
// return  std::sin(3.14159*1*x)*std::sin(3.14159*1*y);
//  return 16.0*1024. ;
// return (double)1.0 ;
    return x * x * std::sin(2.5 * 3.14159 * y);
}

double FunctU(const double /* x */, double const /* y */)
{
    return 1.0 ;
}
```

```
    1: #ifndef USERSET_FILE
    2: #define USERSET_FILE
    3: #include <cmath>
    4: /**
    5:  * User function: f(@p x,@p y)
    6:  * @param[in] x         x-coordinate of discretization point
    7:  * @param[in] y         y-coordinate of discretization point
    8:  * @return  value for right hand side f(@p x,@p y)
    9:  */
   10: double FunctF(double const x, double const y);
   11:
   12: /**
   13:  * User function: u(@p x,@p y)
   14:  * @param[in] x         x-coordinate of discretization point
   15:  * @param[in] y         y-coordinate of discretization point
   16:  * @return  value for solution vector u(@p x,@p y)
   17:  */
   18: double FunctU(double const x, double const y);
   19:
   20:
   21: /**
   22:  * User function: f(@p x,@p y) = @f$ x^2 \sin(2.5\pi y)@f$.
   23:  * @param[in] x         x-coordinate of discretization point
   24:  * @param[in] y         y-coordinate of discretization point
   25:  * @return  value f(@p x,@p y)
   26:  */
   27: inline double fNice(double const x, double const y)
   28: {
   29:     return x * x * std::sin(2.5 * 3.14159 * y);
   30: }
   31:
   32: /**
   33:  * User function: f(@p x,@p y) = 0$.
   34:  * @param[in] x         x-coordinate of discretization point
   35:  * @param[in] y         y-coordinate of discretization point
   36:  * @return  value 0
   37:  */
   38: inline double f_zero(double const x, double const y)
   39: //double f_zero(double const  /*x*/, double const /*y*/)
   40: {
   41:     return 0.0 + 0.0*(x+y);
   42: }
   43:
   44: #endif
```

```cpp
#include "vdop.h"
#include <cassert>                // assert()
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;


void vddiv(vector<double> &x, vector<double> const &y,
           vector<double> const &z)
{
    assert( x.size() == y.size() && y.size() == z.size() );
    size_t n = x.size();

    #pragma omp parallel for
    for (size_t k = 0; k < n; ++k)
    {
        x[k] = y[k] / z[k];
    }
    return;
}

//****************************************************************************

void vdaxpy(std::vector<double> &x, std::vector<double> const &y,
            double alpha, std::vector<double> const &z )
{
    assert( x.size() == y.size() && y.size() == z.size() );
    size_t n = x.size();

    #pragma omp parallel for
    for (size_t k = 0; k < n; ++k)
    {
        x[k] = y[k] + alpha * z[k];
    }
    return;
}
//****************************************************************************

double dscapr(std::vector<double> const &x, std::vector<double> const &y)
{
    assert( x.size() == y.size());
    size_t n = x.size();

    double    s = 0.0;
    #pragma omp parallel for reduction(+:s)
    for (size_t k = 0; k < n; ++k)
    {
        s += x[k] * y[k];
    }

    return s;
}

//****************************************************************************
void DebugVector(vector<double> const &v)
{
    cout << "\nVector  (nnode = " << v.size() << ")\n";
    for (size_t j = 0; j < v.size(); ++j)
    {
        cout.setf(ios::right, ios::adjustfield);
        cout << v[j] << "   ";
    }
    cout << endl;

    return;
}
//****************************************************************************
```

```
   69: bool CompareVectors(std::vector<double> const &x, int const n, double const y[], dou
ble const eps)
   70: {
   71:     bool bn = (static_cast<int>(x.size()) == n);
   72:     if (!bn)
   73:     {
   74:         cout << "#########   Error: " << "number of elements" << endl;
   75:     }
   76:     //bool bv = equal(x.cbegin(),x.cend(),y);
   77:     bool bv = equal(x.cbegin(), x.cend(), y,
   78:                     [eps](double a, double b) -> bool
   79:     { return std::abs(a - b) < eps * (1.0 + 0.5 * (std::abs(a) + std::abs(a))); }
   80:                    );
   81:     if (!bv)
   82:     {
   83:         assert(static_cast<int>(x.size()) == n);
   84:         cout << "#########   Error: " << "values" << endl;
   85:     }
   86:     return bn && bv;
   87: }
```

```
    1: #ifndef VDOP_FILE
    2: #define VDOP_FILE
    3: #include <vector>
    4:
    5: /** @brief  Element-wise vector divison x_k = y_k/z_k.
    6:  *
    7:  * @param[out] x  target vector
    8:  * @param[in]  y  source vector
    9:  * @param[in]  z  source vector
   10:  *
   11:  */
   12: void vddiv(std::vector<double> & x, std::vector<double> const& y,
   13:                                     std::vector<double> const& z);
   14:
   15: /** @brief  Element-wise daxpy operation x(k) = y(k) + alpha*z(k).
   16:  *
   17:  * @param[out] x  target vector
   18:  * @param[in]  y  source vector
   19:  * @param[in]  alpha  scalar
   20:  * @param[in]  z  source vector
   21:  *
   22:  */
   23: void vdaxpy(std::vector<double> & x, std::vector<double> const& y,
   24:                         double alpha, std::vector<double> const& z );
   25:
   26:
   27: /** @brief  Calculates the Euclidian inner product of two vectors.
   28:  *
   29:  * @param[in]  x vector
   30:  * @param[in]  y vector
   31:  * @return Euclidian inner product @f$\langle x,y \rangle@f$
   32:  *
   33:  */
   34: double dscapr(std::vector<double> const& x, std::vector<double> const& y);
   35:
   36:
   37: /**
   38:  * Print entries of a vector.
   39:  * @param[in] v     vector values
   40: */
   41: void DebugVector(std::vector<double> const &v);
   42:
   43: /** @brief  Compares an STL vector with POD vector.
   44:  *
   45:  * The accuracy criteria @f$ |x_k-y_k| < \varepsilon \left({1+0.5(|x_k|+|y_k|)}\righ
t) @f$
   46:  * follows the book by
   47:  * <a href="https://www.springer.com/la/book/9783319446592">Stoyan/Baran</a>, p.8.
   48:  *
   49:  * @param[in]  x    STL vector
   50:  * @param[in]  n    length of POD vector
   51:  * @param[in]  y    POD vector
   52:  * @param[in]  eps  relative accuracy criteria (default := 0.0).
   53:  * @return true iff pairwise vector elements are relatively close to each other.
   54:  *
   55:  */
   56: bool CompareVectors(std::vector<double> const& x, int n, double const y[], double co
nst eps=0.0);
   57:
   58: #endif
```

```
    1: #include "mylib.h"
    2:
    3: #include <chrono>              // timing
    4: #include <cstdlib>             // atoi()
    5: #include <cstring>             // strncmp()
    6: #include <iostream>
    7: #include <sstream>
    8:
    9: using namespace std;
   10: using namespace std::chrono;  // timing
   11:
   12: int main(int argc, char **argv)
   13: {
   14:     int const NLOOPS = 50;         // chose a value such that the benchmark runs at l
east 10 sec.
   15:     unsigned int N = 50000001;
   16: //#########################################################################
   17: //   Read Paramater from command line  (C++ style)
   18:     cout << "Checking command line parameters for: -n <number> " << endl;
   19:     for (int i = 1; i < argc; i++)
   20:     {
   21:         cout << " arg[" << i << "] = " << argv[i] << endl;
   22:         if (std::strncmp(argv[i], "-n", 2) == 0 && i + 1 < argc) // found "-n" follo
wed by another parameter
   23:         {
   24:             N = static_cast<unsigned int>(atoi(argv[i + 1]));
   25:         }
   26:         else
   27:         {
   28:             cout << "Corect call: " << argv[0] << " -n  <number>\n";
   29:         }
   30:     }
   31:
   32:     cout << "\nN = " << N << endl;
   33:
   34: //#########################################################################
   35: //  Memory allocation
   36:     cout << "Memory allocation\n";
   37:
   38:     double *x, *y;
   39:     x = new double [N];
   40:     y = new double [N];
   41:
   42:     cout.precision(2);
   43:     cout << 2.0 * N *sizeof(x[0]) / 1024 / 1024 / 1024 << " GByte Memory allocated\n
";
   44:     cout.precision(6);
   45:
   46: //#########################################################################
   47: //  Data initialization
   48: //  Special:  x_i = i+1;  y_i = 1/x_i  ==> <x,y> == N
   49:     for (unsigned int i = 0; i < N; ++i)
   50:     {
   51:         x[i] = i + 1;
   52:         y[i] = 1.0 / x[i];
   53:     }
   54:
   55: //#########################################################################
   56:     cout << "\nStart Benchmarking\n";
   57:
   58:     auto tstart = system_clock::now();       // start timer
   59:
   60: // Do calculation
   61:     double sk(0.0);
   62:
   63:     for (int i = 0; i < NLOOPS; ++i)
   64:     {
   65:         sk = scalar(N, x, y);
```

```
   66: //      sk = norm(N,x);
   67:     }
   68:
   69:     auto tend = system_clock::now();         // end timer
   70:     auto duration = duration_cast<microseconds>(tend - tstart);
   71:     auto t1 = static_cast<double>(duration.count()) / 1e6 ;   // t1 in seconds
   72:     t1 /= NLOOPS;           // divide by number of function calls
   73:
   74: //#############################################################################
   75: // Check the correct result
   76:     cout << "\n <x,y> = " << sk << endl;
   77:     if (static_cast<unsigned int>(sk) != N)
   78:     {
   79:         cout << "  !!   W R O N G  result   !!\n";
   80:     }
   81:     cout << endl;
   82:
   83: //#############################################################################
   84: // Timings  and Performance
   85:     cout << endl;
   86:     cout.precision(2);
   87:     cout << "Timing in sec. : " << t1 << endl;
   88:     cout << "GFLOPS         : " << 2.0 * N / t1 / 1024 / 1024 / 1024 << endl;
   89:     cout << "GiByte/s        : " << 2.0 * N / t1 / 1024 / 1024 / 1024 * sizeof(x[0])
 << endl;
   90:
   91: //#############################################################################
   92: //  Free allocated memory
   93:     delete [] y;
   94:     delete [] x;
   95:
   96:     return 0;
   97: }
```

```cpp
#include "mylib.h"

#include <chrono>               // timing
#include <cstdlib>              // atoi()
#include <cstring>              // strncmp()
#include <execution>            // policy
#include <iostream>
#include <numeric>              // transform_reduce
#include <sstream>
#include <vector>

using namespace std;
using namespace std::chrono;  // timing

int main(int argc, char **argv)
{
    int const NLOOPS = 50;        // chose a value such that the benchmark runs at l
east 10 sec.
    unsigned int N = 50000001;
//#############################################################################
//   Read Paramater from command line  (C++ style)
    cout << "Checking command line parameters for: -n <number> " << endl;
    for (int i = 1; i < argc; i++)
    {
        cout << " arg[" << i << "] = " << argv[i] << endl;
        if (std::strncmp(argv[i], "-n", 2) == 0 && i + 1 < argc) // found "-n" follo
wed by another parameter
        {
            N = static_cast<unsigned int>(atoi(argv[i + 1]));
        }
        else
        {
            cout << "Corect call: " << argv[0] << " -n  <number>\n";
        }
    }

    cout << "\nN = " << N << endl;

//#############################################################################
//  Memory allocation
    cout << "Memory allocation\n";

    //double *x = new double [N];
    //double *y = new double [N];
    vector<double> x(N);
    vector<double> y(N);
    //alignas(16) double x[N], y[N];

    cout.precision(2);
    cout << 2.0 * N *sizeof(x[0]) / 1024 / 1024 / 1024 << " GiByte Memory allocated\
n";
    cout.precision(6);

//#############################################################################
//  Data initialization
//  Special:  x_i = i+1;  y_i = 1/x_i  ==> <x,y> == N
    for (unsigned int i = 0; i < N; ++i)
    {
        x[i] = i + 1;
        y[i] = 1.0 / x[i];
    }

//#############################################################################
    cout << "\nStart Benchmarking\n";

    auto tstart = system_clock::now();       // start timer

// Do calculation
```

```
   66:     double sk(0.0);
   67:
   68:     for (int i = 0; i < NLOOPS; ++i)
   69:     {
   70:         //sk = scalar(N, x, y);
   71:         sk += scalar_unroll(N, x.data(), y.data());
   72:         //sk += transform_reduce(std::execution::par_unseq,cbegin(x),cend(x),cbegin(
y),0.0);
   73: //      sk = norm(N,x);
   74:     }
   75:
   76:     auto tend = system_clock::now();         // end timer
   77:     auto duration = duration_cast<microseconds>(tend - tstart);
   78:     auto t1 = static_cast<double>(duration.count()) / 1e6 ;   // t1 in seconds
   79:     t1 /= NLOOPS;           // divide by number of function calls
   80:
   81: //============================================================================
   82: // Check the correct result
   83:     sk /= NLOOPS;
   84:     cout << "\n <x,y> = " << sk << endl;
   85:     if (static_cast<unsigned int>(sk) != N)
   86:     {
   87:         cout << "  !!   W R O N G  result   !!\n";
   88:     }
   89:     cout << endl;
   90:
   91: //============================================================================
   92: // Timings  and Performance
   93:     cout << endl;
   94:     cout.precision(2);
   95:     cout << "Timing in sec. : " << t1 << endl;
   96:     cout << "GFLOPS         : " << 2.0 * N / t1 / 1024 / 1024 / 1024 << endl;
   97:     cout << "GiByte/s        : " << 2.0 * N / t1 / 1024 / 1024 / 1024 * sizeof(x[0])
 << endl;
   98:
   99: //============================================================================
  100: //  Free allocated memory
  101:     //delete [] y;
  102:     //delete [] x;
  103:
  104:     return 0;
  105: }
```

```cpp
#include "mylib.h"
#include <cassert>                  // assert()
#include <cmath>
#include <cstdlib>                  // alignof()
//#include <iostream>
#include <numeric>
#include <vector>
using namespace std;


double scalar(unsigned int const N, double const x[], double const y[])
{
    double sum = 0.0;
    for (unsigned int i = 0; i < N; ++i)
    {
        sum += x[i] * y[i];
//      sum += exp(x[i])*log(y[i]);
    }
    return sum;
}

double scalar_unroll(unsigned int const N, double const x[], double const y[])
{
    constexpr unsigned int Stride{8};
    alignas(32) double sk[Stride] = {0.0};
    assert(alignof(sk)==32);
    assert(alignof(x)==8);
    assert(alignof(y)==8);
    for (unsigned int i = 0; i < (N/Stride)*Stride; i+=Stride)
    {
        for (unsigned int k=0; k<Stride; ++k)
        {
            sk[k] += x[i+k] * y[i+k];
        }
    }
    double sum = std::accumulate(sk, sk+Stride,0.0);
    for (unsigned int i = (N/Stride)*Stride; i < N; ++i)
    {
        sum += x[i]*y[i];
    }

    return sum;
}



double norm(unsigned int const N, double const x[])
{
    double sum = 0.0;
    for (unsigned int i = 0; i < N; ++i)
    {
        sum += x[i] * x[i];
    }
    return std::sqrt(sum);
}

```

```
 1: #pragma once
 2:
 3: /**      Inner product
 4:          @param[in] N    number of vector elements
 5:          @param[in] x    vector
 6:          @param[in] y    vector
 7:          @return             resulting Euclidian inner product <x,y>
 8: */
 9: double scalar(unsigned int N, double const x[], double const y[]);
10: double scalar_unroll(unsigned int const N, double const x[], double const y[]);
11:
12: /**      L_2 Norm of a vector
13:          @param[in] N    number of vector elements
14:          @param[in] x    vector
15:          @return             resulting Euclidian norm <x,y>
16: */
17: double norm(unsigned int N, double const x[]);
```

```
    1: #include "mylib.h"
    2: #include <cassert>
    3: #include <chrono>            // timing
    4: #include <cmath>             // sqrt()
    5: #include <cstdlib>           // atoi()
    6: #include <cstring>           // strncmp()
    7: #include <ctime>
    8: #include <iostream>
    9: #include <sstream>
   10: using namespace std;
   11: using namespace std::chrono;  // timing
   12:
   13: int main(int argc, char **argv)
   14: {
   15:     int const NLOOPS = 50;        // chose a value such that the benchmark runs at l
east 10 sec.
   16:     unsigned int N = 50000001;
   17: //#########################################################################
   18: //   Read Paramater from command line  (C++ style)
   19:     cout << "Checking command line parameters for: -n <number> " << endl;
   20:     for (int i = 1; i < argc; i++)
   21:     {
   22:         cout << " arg[" << i << "] = " << argv[i] << endl;
   23:         if (std::strncmp(argv[i], "-n", 2) == 0 && i + 1 < argc) // found "-n" follo
wed by another parameter
   24:         {
   25:             N = static_cast<unsigned int>(atoi(argv[i + 1]));
   26:         }
   27:         else
   28:         {
   29:             cout << "Corect call: " << argv[0] << " -n  <number>\n";
   30:         }
   31:     }
   32:
   33:     cout << "\nN = " << N << endl;
   34:
   35: //#########################################################################
   36: //  Memory allocation
   37:     cout << "Memory allocation\n";
   38:
   39:     vector<double> x(N), y(N);
   40:
   41:     cout.precision(2);
   42:     cout << 2.0 * N *sizeof(x[0]) / 1024 / 1024 / 1024 << " GByte Memory allocated\n
";
   43:     cout.precision(6);
   44:
   45: //#########################################################################
   46: //  Data initialization
   47: //  Special:  x_i = i+1;  y_i = 1/x_i  ==> <x,y> == N
   48:     for (unsigned int i = 0; i < N; ++i)
   49:     {
   50:         x[i] = i + 1;
   51:         y[i] = 1.0 / x[i];
   52:     }
   53:
   54: //#########################################################################
   55:     cout << "\nStart Benchmarking\n";
   56:
   57:     auto t1 = system_clock::now(); // start timer
   58: // Do calculation
   59:     double sk(0.0),ss(0.0);
   60:     for (int i = 0; i < NLOOPS; ++i)
   61:     {
   62:         sk = scalar(x, y);
   63:         //sk = scalar_cblas(x, y);
   64:       //sk = norm(x);
   65:         ss += sk;                   // prevents the optimizer from removing unused c
```

```
alculation results.
   66:     }
   67:
   68:     auto t2 = system_clock::now();  // stop timer
   69:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
   70:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
   71:     t_diff = t_diff/NLOOPS;                                      // duration per loo
p seconds
   72:
   73:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
   74:
   75: //###########################################################################
   76: // Check the correct result
   77:     cout << "\n <x,y> = " << sk << endl;
   78:     if (static_cast<unsigned int>(sk) != N)
   79:     {
   80:         cout << "  !!   W R O N G  result   !!\n";
   81:     }
   82:     cout << endl;
   83:
   84: //###########################################################################
   85: // Timings  and Performance
   86:     cout << endl;
   87:     cout.precision(2);
   88:     cout << "Timing in sec. : " << t_diff << endl;
   89:     cout << "GFLOPS         : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 << endl;
   90:     cout << "GiByte/s       : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 * sizeof(x[
0]) << endl;
   91:
   92: //###########################################################################
   93:     cout << "\nStart Benchmarking norm\n";
   94:
   95:     auto t3 = system_clock::now(); // start timer
   96: // Do calculation
   97:     double ss2(0.0);
   98:     for (int i = 0; i < NLOOPS; ++i)
   99:     {
  100:         auto sk = sqrt(scalar(x, x));
  101:         //auto sk = norm(x);         // the same timing as scalar(x, x)
  102:         ss2 += sk;                   // prevents the optimizer from removing unused
calculation results.
  103:     }
  104:
  105:     auto t4 = system_clock::now();  // stop timer
  106:     auto duration2 = duration_cast<microseconds>(t4 - t3);        // duration in mic
roseconds
  107:     double t_diff2 = static_cast<double>(duration2.count()) / 1e6; // overall durati
on in seconds
  108:     t_diff2 = t_diff2/NLOOPS;                                     // duration per l
oop seconds
  109:
  110:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  111:     cout << "ss(norm): " << ss2 << endl;
  112:     cout << "Timing in sec. : " << t_diff2 << endl;
  113:
  114: //###########################################################################
  115:
  116:
  117:     return 0;
  118: }  // memory for x and y will be deallocated by their destructors
  119:
```

```cpp
#include "mylib.h"
#include <cassert>       // assert()
#include <cmath>
#include <vector>

#ifdef __INTEL_CLANG_COMPILER
#pragma message(" ##########  Use of MKL  ###############")
#include <mkl.h>
#else
#pragma message(" ##########  Use of CBLAS  ###############")
//extern "C"
//{
#include <cblas.h>                // cBLAS Library
#include <lapacke.h>              // Lapack
//}
#endif

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;
    for (size_t i = 0; i < N; ++i)
    {
        sum += x[i] * y[i];
        //sum += exp(x[i])*log(y[i]);
    }
    return sum;
}


double scalar_cblas(vector<double> const &x, vector<double> const &y)
{
    int const asize = static_cast<int>(size(x));
    int const bsize = static_cast<int>(size(y));
    assert(asize == bsize); // switch off via compile flag: -DNDEBUG
        return cblas_ddot(asize,x.data(),1,y.data(),1);
    //assert(x.size() == y.size()); // switch off via compile flag: -DNDEBUG
        //return cblas_ddot(x.size(),x.data(),1,y.data(),1);
}

float scalar_cblas(vector<float> const &x, vector<float> const &y)
{
    int const asize = static_cast<int>(size(x));
    int const bsize = static_cast<int>(size(y));
    assert(asize == bsize); // switch off via compile flag: -DNDEBUG
        return cblas_sdot(asize,x.data(),1,y.data(),1);
    //assert(x.size() == y.size()); // switch off via compile flag: -DNDEBUG
        //return cblas_ddot(x.size(),x.data(),1,y.data(),1);
}


double norm(vector<double> const &x)
{
    size_t const N = x.size();
    double sum = 0.0;
    for (size_t i = 0; i < N; ++i)
    {
        sum += x[i] * x[i];
    }
    return std::sqrt(sum);
}

```

```
 1: #ifndef FILE_MYLIB
 2: #define FILE_MYLIB
 3: #include <vector>
 4:
 5: /**     Inner product
 6:         @param[in] x    vector
 7:         @param[in] y    vector
 8:         @return             resulting Euclidian inner product <x,y>
 9: */
10: double scalar(std::vector<double> const &x, std::vector<double> const &y);
11:
12: /**     Inner product using BLAS routines
13:         @param[in] x    vector
14:         @param[in] y    vector
15:         @return             resulting Euclidian inner product <x,y>
16: */
17: double scalar_cblas(std::vector<double> const &x, std::vector<double> const &y);
18: float scalar_cblas(std::vector<float> const &x, std::vector<float> const &y);
19:
20:
21: /**     L_2 Norm of a vector
22:         @param[in] x    vector
23:         @return             resulting Euclidian norm <x,y>
24: */
25: double norm(std::vector<double> const &x);
26:
27:
28:
29:
30: #endif
```

```
    1: // C++ Vorlesung  xxx
    2: // C++-17: Threads, execution policies
    3: //   Great web pages, great book by Filipek:   https://www.bfilipek.com/2018/06/parstl-tests.html
    4:
    5: /*
    6:  g++ -O3 -std=c++17 gh_main.cpp -ltbb
    7:  g++ -O3 -std=c++17 -pedantic -Weffc++ -Wall -Wextra -pedantic -Wswitch-default -Wfloat-equal -Wundef -Wredundant-decls -Winit-self -Wshadow -Wparentheses -Wshadow -Wunreachable-code -Wuninitialized -Wmaybe-uninitialized gh_main.cpp -ltbb
    8:  ---
    9:  cppcheck --enable=all --inconclusive --std=c++11 --std=posix --suppress=missingIncludeSystem gh_main.cpp
   10:  clang++ -O3 -std=c++17 -ltbb gh_main.cpp
   11:  clang++ -std=c++17 -fsyntax-only -Wdocumentation -Wconversion -Wshadow -Wfloat-conversion -pedantic gh_main.cpp
   12:  clang++ -std=c++17 -Weverything -Wno-c++98-compat -Wno-padded -ltbb gh_main.cpp
   13:  clang++ -cc1 --help
   14:  ---
   15:  icpc -O3 -std=c++17 -ltbb -Wall -Wextra -pedantic gh_main.cpp
   16: */
   17: #include <algorithm>
   18: #include <chrono>
   19: #include <execution>          // execution policy
   20: #include <iostream>
   21: #include <list>               // list
   22: #include <numeric>            // accumulate
   23: #include <random>
   24: #include <vector>
   25: using namespace std;
   26: using namespace std::chrono;  // timing
   27:
   28:
   29: /** \brief Prints the whole vector of base class pointers
   30:  *  \param[in,out]  s   output stream
   31:  *  \param[in]      v   vector
   32:  *  \return        changed output stream
   33:  */
   34: template <class T>
   35: ostream& operator<<(ostream &s, const vector<T>& v)
   36: {
   37:     for (const auto& it: v)             // Reference is required with unique_ptr. No copy constructor for unique_ptr available!
   38:     {
   39:         cout << *it << " ";
   40:     }
   41:     return s;
   42: }
   43:
   44:
   45: int main()
   46: {
   47:     cout << "Threads C++17" << endl;
   48:
   49:     size_t const N = 1<<25;
   50:     vector<double> v(N);
   51:     //list<double> v(N);       // execution::par not possible : missing 'operator-'
   52:     iota(v.begin(), v.end(), 1);
   53:     std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});
   54:
   55:     auto const v_bak(v);
   56:
   57:     {
   58:         v = v_bak;
   59:         cout << "----   sort old   ----" << endl;
   60:         auto t1 = system_clock::now();
   61:         sort(v.begin(), v.end());
   62:         auto t2 = system_clock::now();
```

```
   63:         auto duration = duration_cast<microseconds>(t2 - t1);
   64:         cout << "sort old   : " << static_cast<double>(duration.count()) / 1e6 << "
sec." << endl;
   65:     }
   66:
   67:     {
   68:         v = v_bak;
   69:         cout << "----   sort seq   ----" << endl;
   70:         auto t1 = system_clock::now();
   71:         sort(std::execution::seq, v.begin(), v.end());
   72:         auto t2 = system_clock::now();
   73:         auto duration = duration_cast<microseconds>(t2 - t1);
   74:         cout << "sort seq   : " << static_cast<double>(duration.count()) / 1e6 << "
sec." << endl;
   75:     }
   76:
   77:     {
   78:         v = v_bak;
   79:         cout << "----   sort par   ----" << endl;
   80:         auto t1 = system_clock::now();
   81:         sort(std::execution::par, v.begin(), v.end());
   82:         //auto cnt = count(std::execution::par, v.begin(), v.end(), 17);
   83:         auto t2 = system_clock::now();
   84:         auto duration = duration_cast<microseconds>(t2 - t1);
   85:         cout << "sort par   : " << static_cast<double>(duration.count()) / 1e6 << "
sec." << endl;
   86:     }
   87:
   88:     {
   89:         v = v_bak;
   90:         cout << "----   sort par_unseq   ----" << endl;
   91:         auto t1 = system_clock::now();
   92:         sort(std::execution::par_unseq, v.begin(), v.end());
   93:         auto t2 = system_clock::now();
   94:         auto duration = duration_cast<microseconds>(t2 - t1);
   95:         cout << "sort par_unseq: " << static_cast<double>(duration.count()) / 1e6 <<
 " sec." << endl;
   96:     }
   97:
   98:
   99:     return 0;
  100: }
  101:
```

```cpp
// C++ Vorlesung  xxx
// C++-17: Threads, execution policies
//   Great web pages, great book by Filipek:   https://www.bfilipek.com/2018/06/parstl-tests.html

/*
  g++ -O3 -std=c++17 main.cpp -ltbb
  g++ -O3 -std=c++17 -pedantic -Weffc++ -Wall -Wextra -pedantic -Wswitch-default -Wfloat-equal -Wundef -Wredundant-decls -Winit-self -Wshadow -Wparentheses -Wshadow -Wunreachable-code -Wuninitialized -Wmaybe-uninitialized main.cpp -ltbb
  ---
  cppcheck --enable=all --inconclusive --std=c++11 --std=posix --suppress=missingIncludeSystem main.cpp
  clang++ -O3 -std=c++17 -ltbb main.cpp
  clang++ -std=c++17 -fsyntax-only -Wdocumentation -Wconversion -Wshadow -Wfloat-conversion -pedantic main.cpp
  clang++ -std=c++17 -Weverything -Wno-c++98-compat -Wno-padded -ltbb main.cpp
  clang++ -cc1 --help
  ---
  icpc -O3 -std=c++17 -ltbb -Wall -Wextra -pedantic main.cpp
*/

#include <algorithm>
#include <chrono>
#include <execution>          // execution policy
#include <iostream>
#include <numeric>            //  accumulate
#include <random>
#include <vector>
using namespace std;
using namespace std::chrono;  // timing

// Great web pages, great book by Filipek
// https://www.bfilipek.com/2018/06/parstl-tests.html
template <typename TFunc> void RunAndMeasure(const char *title, TFunc func)
{
    const auto start = std::chrono::steady_clock::now();
    auto ret = func();
    const auto end = std::chrono::steady_clock::now();
    std::cout << title << ": " <<
              std::chrono::duration <double, std::milli>(end - start).count()
              << " ms, res " << ret << "\n";
}

int main()
{
    //std::vector<double> v(6000000, 0.5);
    std::vector<double> v(1<<30, 0.5);

    RunAndMeasure("std::warm up", [&v]
    {
        return std::reduce(std::execution::seq, v.begin(), v.end(), 0.0);
    });

    RunAndMeasure("std::accumulate", [&v]
    {
        return std::accumulate(v.begin(), v.end(), 0.0);
    });

    RunAndMeasure("std::reduce, seq", [&v]
    {
        return std::reduce(std::execution::seq, v.begin(), v.end(), 0.0);
    });

    RunAndMeasure("std::reduce, par", [&v]
    {
        return std::reduce(std::execution::par, v.begin(), v.end(), 0.0);
    });
```

```
64:
65:     RunAndMeasure("std::reduce, par_unseq", [&v]
66:     {
67:         return std::reduce(std::execution::par_unseq, v.begin(), v.end(), 0.0);
68:     });
69:
70:     RunAndMeasure("std::find, seq", [&v]
71:     {
72:         auto res = std::find(std::execution::seq, std::begin(v), std::end(v), 0.6);
73:         return res == std::end(v) ? 0.0 : 1.0;
74:     });
75:
76:     RunAndMeasure("std::find, par", [&v]
77:     {
78:         auto res = std::find(std::execution::par, std::begin(v), std::end(v), 0.6);
79:         return res == std::end(v) ? 0.0 : 1.0;
80:     });
81:
82:     return 0;
83: }
84:
```