```cpp
#include "bsp_3_lib_bench_par.h"
#include <cassert>
#include <chrono>
#include <cmath>
#include <iostream>
#include <ctime>
#include <cblas.h>
#include <lapacke.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

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


double scalar_cblas(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 = cblas_ddot(N, x.data(), 1, y.data(), 1);

    return sum;
}


double scalar_kahan(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;
    double c = 0.0;
    for (size_t i = 0; i < N; ++i)
    {
        double yk = x[i] * y[i] - c;
        double t = sum + yk;
        c = t - sum - yk;
        sum = t;
        //sum += exp(x[i])*log(y[i]);
```

```
    }
    return sum;
}


double norm_eucl(std::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];
        //sum += exp(x[i])*log(y[i]);
    }
    sum = sqrt(sum);
    return sum;
}


double sum(std::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];
    }

    return sum;
}


double sum_par(std::vector<double> const &x)
{
    size_t const N = x.size();
    double sum = 0.0;

#pragma omp parallel for default(none) shared(x, N) reduction(+:sum)
    for (size_t i = 0; i < N; ++i)
    {
        sum += x[i];
    }

    return sum;
}


vector<double> MatVec(vector<double> const & a, vector<double> const & x)   // row w
ise access
{
    int const nelem = static_cast<int>(a.size());      // #elements in matrix
    int const mcols = static_cast<int>(x.size());      // #elements in vector <==> #
columns in matrix

    assert(nelem % mcols == 0);                        // nelem has to be a multiple
 of mcols (==> #rows)
    int const nrows = nelem/mcols;                     // integer division!

    vector<double> b(nrows);                  // allocate resulting vector

    for(size_t i = 0; i < nrows; ++i)
    {
        double tmp = 0.0;
        for(size_t j = 0; j < mcols; ++j)
        {
            tmp = tmp + a[i*mcols+j] * x[j];
        }
```

```
  134:         b[i] = tmp;
  135:     }
  136:
  137:     return b;
  138: }
  139:
  140:
  141: vector<double> MatVec_par(vector<double> const & a, vector<double> const & x)   // r
ow wise access
  142: {
  143:     int const nelem = static_cast<int>(a.size());      // #elements in matrix
  144:     int const mcols = static_cast<int>(x.size());      // #elements in vector <==> #
columns in matrix
  145:
  146:     assert(nelem % mcols == 0);                        // nelem has to be a multiple
 of mcols (==> #rows)
  147:     int const nrows = nelem/mcols;                     // integer division!
  148:
  149:     vector<double> b(nrows);                  // allocate resulting vector
  150:
  151: #pragma omp parallel for default(none) shared(a, x, b, nrows, mcols)    // no reduct
ion: b[i] is safe from other threads
  152:     for(size_t i = 0; i < nrows; ++i)
  153:     {
  154:         double tmp = 0.0;
  155:         for(size_t j = 0; j < mcols; ++j)
  156:         {
  157:             tmp = tmp + a[i*mcols+j] * x[j];
  158:         }
  159:         b[i] = tmp;
  160:     }
  161:
  162:     return b;
  163: }
  164:
  165:
  166: vector<double> MatVec_cblas(vector<double> const & a, vector<double> const & x)   //
 row wise access
  167: {
  168:     int const nelem = static_cast<int>(a.size());      // #elements in matrix
  169:     int const mcols = static_cast<int>(x.size());      // #elements in vector <==> #
columns in matrix
  170:
  171:     assert(nelem % mcols == 0);                        // nelem has to be a multiple
 of mcols (==> #rows)
  172:     int const nrows = nelem/mcols;                     // integer division!
  173:
  174:     vector<double> b(nrows);                  // allocate resulting vector
  175:
  176:         double const alpha = 1.0;
  177:         double const beta = 0.0;
  178:
  179:     cblas_dgemv(CblasRowMajor, CblasTrans,
  180:                 nrows, mcols,
  181:                 alpha, a.data(), nrows,
  182:                 x.data(), 1,
  183:                 beta, b.data(), 1);
  184:
  185:     return b;
  186: }
  187:
  188:
  189: vector<double> MatVec_column(vector<double> const & a, vector<double> const & x)   /
/ column wise access
  190: {
  191:     int const nelem = static_cast<int>(a.size());      // #elements in matrix
  192:     int const mcols = static_cast<int>(x.size());      // #elements in vector <==> #
columns in matrix
```

```
  193:
  194:     assert(nelem % mcols == 0);                        // nelem has to be a multiple
 of mcols (==> #rows)
  195:     int const nrows = nelem/mcols;                     // integer division!
  196:
  197:     vector<double> b(nrows);                 // allocate resulting vector
  198:
  199:     // if we do it directly we have cache issues - not optimal
  200:     // to make the code more efficient we change the two loops and put the b[i] insi
de the inner loop
  201:     // b is not so large compared to a, so higher amount of writing operations to no
t matter that much
  202:     for(size_t j = 0; j < mcols; ++j)
  203:     {
  204:         double xj = x[j];
  205:         for(size_t i = 0; i < nrows; ++i)
  206:         {
  207:             b[i] += a[j*nrows+i] * xj;
  208:         }
  209:     }
  210:
  211:     return b;
  212: }
  213:
  214:
  215: vector<double> MatMatProd(vector<double> const & a, vector<double> const & b, int co
nst & L)
  216: {
  217:     size_t const a_nelem = a.size();
  218:     size_t const b_nelem = b.size();
  219:
  220:     assert(static_cast<int>(a_nelem) % L == 0 && static_cast<int>(b_nelem) % L == 0)
;
  221:
  222:     size_t M = a_nelem/L;
  223:     size_t N = b_nelem/L;
  224:
  225:     vector<double> c(N*M,0);
  226:
  227:     for(size_t i = 0; i < M; ++i)
  228:     {
  229:         for(size_t k = 0; k < L; ++k)
  230:         {
  231:             for(size_t j = 0; j < N; ++j)
  232:             {
  233:                 c[i*M+j] = c[i*M+j] + a[i*L+k]*b[k*N+j];
  234:             }
  235:         }
  236:     }
  237:
  238:     return c;
  239: }
  240:
  241:
  242: vector<double> MatMatProd_par(vector<double> const & a, vector<double> const & b, in
t const & L)
  243: {
  244:     size_t const a_nelem = a.size();
  245:     size_t const b_nelem = b.size();
  246:
  247:     assert(static_cast<int>(a_nelem) % L == 0 && static_cast<int>(b_nelem) % L == 0)
;
  248:
  249:     size_t M = a_nelem/L;
  250:     size_t N = b_nelem/L;
  251:
  252:     vector<double> c(N*M,0);
  253:
```

Try collapse(2)

```
254: #pragma omp parallel for default(none) shared(a, b, c, L, M, N)
255:     for(size_t i = 0; i < M; ++i)
256:     {
257:         for(size_t k = 0; k < L; ++k)
258:         {
259:             for(size_t j = 0; j < N; ++j)
260:             {
261:                 c[i*M+j] = c[i*M+j] + a[i*L+k]*b[k*N+j];
262:             }
263:         }
264:     }
265:
266:     return c;
267: }
268:
269:
270: vector<double> MatMatProd_cblas(vector<double> const & a, vector<double> const & b,
int const & L)
271: {
272:     size_t const a_nelem = a.size();
273:     size_t const b_nelem = b.size();
274:
275:     assert(static_cast<int>(a_nelem) % L == 0 && static_cast<int>(b_nelem) % L == 0)
;
276:
277:     size_t M = a_nelem/L;
278:     size_t N = b_nelem/L;
279:
280:     vector<double> c(N*M,0);
281:
282:     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
283:                                M, N, L,
284:                                1.0, a.data(), L, b.data(), N,
285:                                0.0, c.data(), N);
286:
287:     return c;
288: }
289:
290:
291: vector<double> PolynomEval(vector<double> const & a, vector<double> const & x)
292: {
293:     // we want to use the Horner-scheme
294:     vector<double> sol(x.size(),0);
295:
296:     for(size_t i = 0; i < x.size(); ++i)
297:     {
298:         double tmp = a[a.size()-1];
299:         for(int k = static_cast<int>(a.size())-2; k >= 0; --k)
300:         {
301:             tmp = tmp*x[i] + a[k];
302:         }
303:         sol[i] = tmp;
304:     }
305:
306:     return sol;
307: }
308:
309:
310: vector<double> PolynomEval_par(vector<double> const & a, vector<double> const & x)
311: {
312:     // we want to use the Horner-scheme
313:     vector<double> sol(x.size(),0);
314:
315: #pragma omp parallel for shared(a, x, sol)
316:     for(size_t i = 0; i < x.size(); ++i)
317:     {
318:         double tmp = a[a.size()-1];
319:         for(int k = static_cast<int>(a.size())-2; k >= 0; --k)
```

```
  320:         {
  321:             tmp = tmp*x[i] + a[k];
  322:         }
  323:         sol[i] = tmp;
  324:     }
  325:
  326:     return sol;
  327: }
  328:
  329:
  330: void benchmark_A(int const & N, int const & Nloops)
  331: {
  332:     //#############################################################################
  333:     cout << "\nStart Benchmarking A: scalar product\n";
  334:
  335:     vector<double> x(N), y(N);
  336:     for(size_t k = 0; k < x.size(); ++k)
  337:     {
  338:         x[k] = (k % 219) + 1;
  339:         y[k] = 1.0/x[k];
  340:     }
  341:
  342:     auto t1 = system_clock::now(); // start timer
  343: // Do calculation
  344:     double sk(0.0), ss(0.0);
  345:     for (int i = 0; i < Nloops; ++i)
  346:     {
  347:         sk = scalar(x, y);
  348:         ss += sk;                   // prevents the optimizer from removing unused c
alculation results.
  349:     }
  350:
  351:     auto t2 = system_clock::now();  // stop timer
  352:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  353:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  354:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  355:
  356:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  357:
  358: //#############################################################################
  359: // Check the correct result
  360:     cout << "\n <x,y> = " << sk << endl;
  361:     if (static_cast<unsigned int>(sk) != N)
  362:     {
  363:         cout << "  !!   W R O N G  result   !!\n";
  364:     }
  365:     cout << endl;
  366:
  367: //#############################################################################
  368: // Timings  and Performance
  369:     cout << endl;
  370:     cout.precision(2);
  371:     cout << "N = " << N << endl;
  372:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  373:     cout << "Timing in sec. : " << t_diff << endl;
  374:     cout << "GFLOPS         : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 << endl;
  375:     cout << "GiByte/s       : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 * sizeof(x[
0]) << endl;
  376:     cout << endl << endl;
  377:
  378:     return;
  379: }
  380:
  381:
```

```
  382: void benchmark_A_cblas(int const & N, int const & Nloops)
  383: {
  384:     //#########################################################################
  385:     cout << "\nStart Benchmarking A: scalar product with cblas\n";
  386:
  387:     vector<double> x(N), y(N);
  388:     for(size_t k = 0; k < x.size(); ++k)
  389:     {
  390:         x[k] = (k % 219) + 1;
  391:         y[k] = 1.0/x[k];
  392:     }
  393:
  394:     auto t1 = system_clock::now(); // start timer
  395: // Do calculation
  396:     double sk(0.0), ss(0.0);
  397:     for (int i = 0; i < Nloops; ++i)
  398:     {
  399:         sk = scalar_cblas(x, y);
  400:         ss += sk;                    // prevents the optimizer from removing unused c
alculation results.
  401:     }
  402:
  403:     auto t2 = system_clock::now();  // stop timer
  404:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  405:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  406:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  407:
  408:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  409:
  410: //#########################################################################
  411: // Check the correct result
  412:     cout << "\n <x,y> = " << sk << endl;
  413:     if (static_cast<unsigned int>(sk) != N)
  414:     {
  415:         cout << "  !!   W R O N G  result   !!\n";
  416:     }
  417:     cout << endl;
  418:
  419: //#########################################################################
  420: // Timings  and Performance
  421:     cout << endl;
  422:     cout.precision(2);
  423:     cout << "N = " << N << endl;
  424:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  425:     cout << "Timing in sec. : " << t_diff << endl;
  426:     cout << "GFLOPS         : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 << endl;
  427:     cout << "GiByte/s       : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 * sizeof(x[
0]) << endl;
  428:     cout << endl << endl;
  429:
  430:     return;
  431: }
  432:
  433:
  434: void benchmark_A_kahan(int const & N, int const & Nloops)
  435: {
  436:     //#########################################################################
  437:     cout << "\nStart Benchmarking A: scalar product with Kahan summation\n";
  438:
  439:     vector<double> x(N), y(N);
  440:     for(size_t k = 0; k < x.size(); ++k)
  441:     {
  442:         x[k] = (k % 219) + 1;
  443:         y[k] = 1.0/x[k];
```

```
  444:     }
  445:
  446:     auto t1 = system_clock::now(); // start timer
  447: // Do calculation
  448:     double sk(0.0), ss(0.0);
  449:     for (int i = 0; i < Nloops; ++i)
  450:     {
  451:         sk = scalar(x, y);
  452:         ss += sk;                   // prevents the optimizer from removing unused c
alculation results.
  453:     }
  454:
  455:     auto t2 = system_clock::now();  // stop timer
  456:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  457:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  458:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  459:
  460:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  461:
  462: //#############################################################################
  463: // Check the correct result
  464:     cout << "\n <x,y> = " << sk << endl;
  465:     if (static_cast<unsigned int>(sk) != N)
  466:     {
  467:         cout << "  !!   W R O N G  result   !!\n";
  468:     }
  469:     cout << endl;
  470:
  471: //#############################################################################
  472: // Timings  and Performance
  473:     cout << endl;
  474:     cout.precision(2);
  475:     cout << "N = " << N << endl;
  476:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  477:     cout << "Timing in sec. : " << t_diff << endl;
  478:     //cout << "GFLOPS         : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 << endl;
  479:     //cout << "GiByte/s       : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 * sizeof(
x[0]) << endl;
  480:     cout << endl << endl;
  481:
  482:     return;
  483: }
  484:
  485:
  486: void benchmark_A_norm(int const & N, int const & Nloops)
  487: {
  488:     //#############################################################################
  489:     cout << "\nStart Benchmarking A_norm: euclidean norm\n";
  490:
  491:     vector<double> x(N,1.0);
  492:
  493:     auto t1 = system_clock::now(); // start timer
  494: // Do calculation
  495:     double sk(0.0), ss(0.0);
  496:     for (int i = 0; i < Nloops; ++i)
  497:     {
  498:         sk = norm_eucl(x);
  499:         ss += sk;                   // prevents the optimizer from removing unused c
alculation results.
  500:     }
  501:
  502:     auto t2 = system_clock::now();  // stop timer
  503:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
```

```
  504:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  505:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  506:
  507:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  508:
  509: //############################################################################
  510: // Check the correct result
  511:     cout << "\n ||x|| = " << sk << endl;
  512:     if (sk - sqrt(N) > 1e-7)
  513:     {
  514:         cout << "  !!   W R O N G  result   !!\n";
  515:     }
  516:     cout << endl;
  517:
  518: //############################################################################
  519: // Timings  and Performance
  520:     cout << endl;
  521:     cout.precision(2);
  522:     cout << "N = " << N << endl;
  523:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  524:     cout << "Timing in sec. : " << t_diff << endl;
  525:     cout << "GFLOPS         : " << 2.0 * N / t_diff / 1024 / 1024 / 1024 << endl;
  526:     cout << "GiByte/s       : " << N / t_diff / 1024 / 1024 / 1024 * sizeof(x[0]) <<
 endl;
  527:     cout << endl << endl;
  528:
  529:     return;
  530: }
  531:
  532:
  533: void benchmark_B(int const & N, int const & M, int const & Nloops)
  534: {
  535:     //############################################################################
  536:     cout << "\nStart Benchmarking B: Matrix-Vector Product (row wise access)\n";
  537:
  538:     vector<double> x(N), b(M), a(N*M);
  539:     // initialize data
  540:     for(size_t i = 0; i < M; ++i)
  541:     {
  542:         for(size_t j = 0; j < N; ++j)
  543:         {
  544:             a[i*N+j] = (i+j) % 219 + 1;
  545:         }
  546:     }
  547:     for(size_t i = 0; i < N; ++i)
  548:     {
  549:         x[i] = 1.0/a[17*N+i];
  550:     }
  551:
  552:     auto t1 = system_clock::now(); // start timer
  553: // Do calculation
  554:     double ss(0.0);
  555:     for (int i = 0; i < Nloops; ++i)
  556:     {
  557:         b = MatVec(a,x);
  558:         ss += b[0];                    // prevents the optimizer from removing unused
 calculation results.
  559:     }
  560:
  561:     auto t2 = system_clock::now();  // stop timer
  562:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  563:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  564:     t_diff = t_diff/Nloops;                                      // duration per loo
```

```
p seconds
  565:
  566:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  567:
  568: //#########################################################################
  569: // Check the correct result
  570:     cout << "\n <A[17,.],x> = " << b[17] << endl;
  571:     if (static_cast<unsigned int>(b[17]) != N)
  572:     {
  573:         cout << "  !!   W R O N G  result   !!\n";
  574:     }
  575:     cout << endl;
  576:
  577: //#########################################################################
  578: // Timings  and Performance
  579:     cout << endl;
  580:     cout.precision(2);
  581:     cout << "N = " << N << "\t M = " << M << endl;
  582:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  583:     cout << "Timing in sec. : " << t_diff << endl;
  584:     cout << "GFLOPS         : " << 2.0 * N * M / t_diff / 1024 / 1024 / 1024 << endl
;
  585:     cout << "GiByte/s       : " << (2.0 * N * M + M) / t_diff / 1024 / 1024 / 1024 *
 sizeof(x[0]) << endl;
  586:     cout << endl << endl;
  587:
  588:     return;
  589: }
  590:
  591:
  592: void benchmark_B_par(int const & N, int const & M, int const & Nloops)
  593: {
  594:     //#########################################################################
  595:     cout << "\nStart Benchmarking B: Matrix-Vector Product (row wise access) [parall
el]\n";
  596:
  597:     vector<double> x(N), b(M), a(N*M);
  598:     // initialize data
  599:     for(size_t i = 0; i < M; ++i)
  600:     {
  601:         for(size_t j = 0; j < N; ++j)
  602:         {
  603:             a[i*N+j] = (i+j) % 219 + 1;
  604:         }
  605:     }
  606:     for(size_t i = 0; i < N; ++i)
  607:     {
  608:         x[i] = 1.0/a[17*N+i];
  609:     }
  610:
  611:     auto t1 = omp_get_wtime(); // start timer
  612: // Do calculation
  613:     double ss(0.0);
  614:     for (int i = 0; i < Nloops; ++i)
  615:     {
  616:         b = MatVec_par(a,x);
  617:         ss += b[0];                    // prevents the optimizer from removing unused
 calculation results.
  618:     }
  619:
  620:         auto t_diff = omp_get_wtime() - t1;
  621:     t_diff = t_diff/Nloops;                                    // duration per loo
p seconds
  622:
  623:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  624:
```

```
  625: //#########################################################################
  626: // Check the correct result
  627:     cout << "\n <A[17,.],x> = " << b[17] << endl;
  628:     if (static_cast<unsigned int>(b[17]) != N)
  629:     {
  630:         cout << "  !!   W R O N G  result   !!\n";
  631:     }
  632:     cout << endl;
  633:
  634: //#########################################################################
  635: // Timings  and Performance
  636:     cout << endl;
  637:     cout.precision(2);
  638:     cout << "N = " << N << "\t M = " << M << endl;
  639:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  640:     cout << "Timing in sec. : " << t_diff << endl;
  641:     cout << "GFLOPS         : " << 2.0 * N * M / t_diff / 1024 / 1024 / 1024 << endl
;
  642:     cout << "GiByte/s       : " << (2.0 * N * M + M) / t_diff / 1024 / 1024 / 1024 *
 sizeof(x[0]) << endl;
  643:     cout << endl << endl;
  644:
  645:     return;
  646: }
  647:
  648:
  649: void benchmark_B_cblas(int const & N, int const & M, int const & Nloops)
  650: {
  651:     //#####################################################################
  652:     cout << "\nStart Benchmarking B: Matrix-Vector Product with cblas (row wise acce
ss)\n";
  653:
  654:     vector<double> x(N), b(M), a(N*M);
  655:     // initialize data
  656:     for(size_t i = 0; i < M; ++i)
  657:     {
  658:         for(size_t j = 0; j < N; ++j)
  659:         {
  660:             a[i*N+j] = (i+j) % 219 + 1;
  661:         }
  662:     }
  663:     for(size_t i = 0; i < N; ++i)
  664:     {
  665:         x[i] = 1.0/a[17*N+i];
  666:     }
  667:
  668:     auto t1 = system_clock::now(); // start timer
  669: // Do calculation
  670:     double ss(0.0);
  671:     for (int i = 0; i < Nloops; ++i)
  672:     {
  673:         b = MatVec_cblas(a,x);
  674:         ss += b[0];                    // prevents the optimizer from removing unused
 calculation results.
  675:     }
  676:
  677:     auto t2 = system_clock::now();  // stop timer
  678:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  679:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  680:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  681:
  682:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  683:
  684: //#########################################################################
```

```
  685: // Check the correct result
  686:     cout << "\n <A[17,.],x> = " << b[17] << endl;
  687:     if (static_cast<unsigned int>(b[17]) != N)
  688:     {
  689:         cout << "  !!   W R O N G  result   !!\n";
  690:     }
  691:     cout << endl;
  692:
  693: //#########################################################################
  694: // Timings  and Performance
  695:     cout << endl;
  696:     cout.precision(2);
  697:     cout << "N = " << N << "\t M = " << M << endl;
  698:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  699:     cout << "Timing in sec. : " << t_diff << endl;
  700:     cout << "GFLOPS         : " << 2.0 * N * M / t_diff / 1024 / 1024 / 1024 << endl
;
  701:     cout << "GiByte/s       : " << (2.0 * N * M + M) / t_diff / 1024 / 1024 / 1024 *
 sizeof(x[0]) << endl;
  702:     cout << endl << endl;
  703:
  704:     return;
  705: }
  706:
  707:
  708: void benchmark_B_column(int const & N, int const & M, int const & Nloops)
  709: {
  710:     //#########################################################################
  711:     cout << "\nStart Benchmarking B: Matrix-Vector Product (column wise access)\n";
  712:
  713:     vector<double> x(N), b(M), a(N*M);
  714:     // initialize data
  715:     for(size_t i = 0; i < M; ++i)
  716:     {
  717:         for(size_t j = 0; j < N; ++j)
  718:         {
  719:             a[i*N+j] = (i+j) % 219 + 1;
  720:         }
  721:     }
  722:     for(size_t i = 0; i < N; ++i)
  723:     {
  724:         x[i] = 1.0/a[17*N+i];
  725:     }
  726:
  727:     auto t1 = system_clock::now(); // start timer
  728: // Do calculation
  729:     double ss(0.0);
  730:     for (int i = 0; i < Nloops; ++i)
  731:     {
  732:         b = MatVec_column(a,x);
  733:         ss += b[0];                    // prevents the optimizer from removing unused
 calculation results.
  734:     }
  735:
  736:     auto t2 = system_clock::now();  // stop timer
  737:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  738:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  739:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  740:
  741:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  742:
  743: //#########################################################################
  744: // Check the correct result
  745:     cout << "\n <A[17,.],x> = " << b[17] << endl;
```

```
  746:     if (static_cast<unsigned int>(b[17]) != N)
  747:     {
  748:         cout << "  !!   W R O N G  result   !!\n";
  749:     }
  750:     cout << endl;
  751:
  752: //#########################################################################
  753: // Timings  and Performance
  754:     cout << endl;
  755:     cout.precision(2);
  756:     cout << "N = " << N << "\t M = " << M << endl;
  757:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  758:     cout << "Timing in sec. : " << t_diff << endl;
  759:     cout << "GFLOPS         : " << 2.0 * N * M / t_diff / 1024 / 1024 / 1024 << endl
;
  760:     cout << "GiByte/s       : " << (2.0 * N * M + M) / t_diff / 1024 / 1024 / 1024 *
 sizeof(x[0]) << endl;
  761:     cout << endl << endl;
  762:
  763:     return;
  764: }
  765:
  766:
  767: void benchmark_C(int const & N, int const & M, int const & L, int const & Nloops)
  768: {
  769:     //#########################################################################
  770:     cout << "\nStart Benchmarking C: Matrix-Matrix Product\n";
  771:
  772:     vector<double> a(M*L,1.0), b(L*N,1.0), c(N*M);
  773:     // with this data we get C[i,j] = L for all i and j
  774:
  775:     auto t1 = system_clock::now(); // start timer
  776: // Do calculation
  777:     double ss(0.0);
  778:     for (int i = 0; i < Nloops; ++i)
  779:     {
  780:         c = MatMatProd(a,b,L);
  781:         ss += c[0];                    // prevents the optimizer from removing unused
 calculation results.
  782:     }
  783:
  784:     auto t2 = system_clock::now();  // stop timer
  785:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  786:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  787:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  788:
  789:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  790:
  791: //#########################################################################
  792: // Check the correct result
  793:     cout << "\n C[10,15] = " << c[10*N+15] << endl;
  794:     if (static_cast<unsigned int>(c[10*N+15]) != L)
  795:     {
  796:         cout << "  !!   W R O N G  result   !!\n";
  797:     }
  798:     cout << endl;
  799:
  800: //#########################################################################
  801: // Timings  and Performance
  802:     cout << endl;
  803:     cout.precision(2);
  804:     cout << "N = " << N << "\t M = " << M << "\t L = " << L << endl;
  805:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  806:     cout << "Timing in sec. : " << t_diff << endl;
```

```
  807:     cout << "GFLOPS          : " << 2.0 * N * M * L / t_diff / 1024 / 1024 / 1024 <<
endl;
  808:     cout << "GiByte/s        : " << (L*(N+M) + M*N) / t_diff / 1024 / 1024 / 1024 * s
izeof(a[0]) << endl;
  809:     cout << endl << endl;
  810:
  811:     return;
  812: }
  813:
  814:
  815: void benchmark_C_par(int const & N, int const & M, int const & L, int const & Nloops
)
  816: {
  817:     //#########################################################################
  818:     cout << "\nStart Benchmarking C: Matrix-Matrix Product [parallel]\n";
  819:
  820:     vector<double> a(M*L,1.0), b(L*N,1.0), c(N*M);
  821:     // with this data we get C[i,j] = L for all i and j
  822:
  823:     auto t1 = omp_get_wtime(); // start timer
  824: // Do calculation
  825:     double ss(0.0);
  826:     for (int i = 0; i < Nloops; ++i)
  827:     {
  828:         c = MatMatProd_par(a,b,L);
  829:         ss += c[0];                    // prevents the optimizer from removing unused
 calculation results.
  830:     }
  831:
  832:     auto t_diff = omp_get_wtime() - t1;  // stop timer
  833:     t_diff = t_diff/Nloops;                                          // duration per loo
p seconds
  834:
  835:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  836:
  837: //#########################################################################
  838: // Check the correct result
  839:     cout << "\n C[10,15] = " << c[10*N+15] << endl;
  840:     if (static_cast<unsigned int>(c[10*N+15]) != L)
  841:     {
  842:         cout << "  !!   W R O N G  result   !!\n";
  843:     }
  844:     cout << endl;
  845:
  846: //#########################################################################
  847: // Timings  and Performance
  848:     cout << endl;
  849:     cout.precision(2);
  850:     cout << "N = " << N << "\t M = " << M << "\t L = " << L << endl;
  851:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  852:     cout << "Timing in sec. : " << t_diff << endl;
  853:     cout << "GFLOPS          : " << 2.0 * N * M * L / t_diff / 1024 / 1024 / 1024 <<
endl;
  854:     cout << "GiByte/s        : " << (L*(N+M) + M*N) / t_diff / 1024 / 1024 / 1024 * s
izeof(a[0]) << endl;
  855:     cout << endl << endl;
  856:
  857:     return;
  858: }
  859:
  860:
  861: void benchmark_C_cblas(int const & N, int const & M, int const & L, int const & Nloo
ps)
  862: {
  863:     //#########################################################################
  864:     cout << "\nStart Benchmarking C: Matrix-Matrix Product mit cblas\n";
  865:
```

```
  866:     vector<double> a(M*L,1.0), b(L*N,1.0), c(N*M);
  867:     // with this data we get C[i,j] = L for all i and j
  868:
  869:     auto t1 = system_clock::now(); // start timer
  870: // Do calculation
  871:     double ss(0.0);
  872:     for (int i = 0; i < Nloops; ++i)
  873:     {
  874:         c = MatMatProd_cblas(a,b,L);
  875:         ss += c[0];                    // prevents the optimizer from removing unused
 calculation results.
  876:     }
  877:
  878:     auto t2 = system_clock::now();  // stop timer
  879:     auto duration = duration_cast<microseconds>(t2 - t1);         // duration in micr
oseconds
  880:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  881:     t_diff = t_diff/Nloops;                                       // duration per loo
p seconds
  882:
  883:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  884:
  885: //#########################################################################
  886: // Check the correct result
  887:     cout << "\n C[10,15] = " << c[10*N+15] << endl;
  888:     if (static_cast<unsigned int>(c[10*N+15]) != L)
  889:     {
  890:         cout << "  !!   W R O N G  result   !!\n";
  891:     }
  892:     cout << endl;
  893:
  894: //#########################################################################
  895: // Timings  and Performance
  896:     cout << endl;
  897:     cout.precision(2);
  898:     cout << "N = " << N << "\t M = " << M << "\t L = " << L << endl;
  899:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  900:     cout << "Timing in sec. : " << t_diff << endl;
  901:     cout << "GFLOPS         : " << 2.0 * N * M * L / t_diff / 1024 / 1024 / 1024 <<
endl;
  902:     cout << "GiByte/s       : " << (L*(N+M) + M*N) / t_diff / 1024 / 1024 / 1024 * s
izeof(a[0]) << endl;
  903:     cout << endl << endl;
  904:
  905:     return;
  906: }
  907:
  908:
  909: void benchmark_D(int const & p, int const & N, int const & Nloops)
  910: {
  911:     //#########################################################################
  912:     cout << "\nStart Benchmarking D: polynomial evaluation\n";
  913:
  914:     vector<double> x(N,1), sol(N), a(p+1);
  915:     for(size_t i = 0; i < a.size(); ++i)
  916:     {
  917:         a[i] = pow(-1.0,i);      // 1-x+x^2-x^3+x^4...
  918:     }
  919:     a[0] = 1;
  920:
  921:     auto t1 = system_clock::now(); // start timer
  922: // Do calculation
  923:     double ss(0.0);
  924:     for (int i = 0; i < Nloops; ++i)
  925:     {
  926:         sol = PolynomEval(a,x);
```

```
  927:         ss += sol[0];                    // prevents the optimizer from removing unus
ed calculation results.
  928:     }
  929:
  930:     auto t2 = system_clock::now();  // stop timer
  931:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  932:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  933:     t_diff = t_diff/Nloops;                                      // duration per loo
p seconds
  934:
  935:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
on "=="
  936:
  937: //#############################################################################
  938: // Check the correct result
  939:     cout << "\n p(x[0]) = " << sol[0] << endl;
  940:     if (static_cast<unsigned int>(sol[0]) != (static_cast<int>(a.size()) % 2))
  941:     {
  942:         cout << "  !!   W R O N G  result   !!\n";
  943:     }
  944:     cout << endl;
  945:
  946: //#############################################################################
  947: // Timings  and Performance
  948:     cout << endl;
  949:     cout.precision(2);
  950:     cout << "p = " << p << "\t N = " << N << endl;
  951:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
  952:     cout << "Timing in sec. : " << t_diff << endl;
  953:     cout << "GFLOPS         : " << 2.0*(p+1)*N / t_diff / 1024 / 1024 / 1024 << endl
;
  954:     cout << "GiByte/s       : " << N*(3+2*p) / t_diff / 1024 / 1024 / 1024 * sizeof(
a[0]) << endl;
  955:     cout << endl << endl;
  956:
  957:     return;
  958: }
  959:
  960:
  961: void benchmark_D_par(int const & p, int const & N, int const & Nloops)
  962: {
  963:     //#############################################################################
  964:     cout << "\nStart Benchmarking D: polynomial evaluation [parallel]\n";
  965:
  966:     vector<double> x(N,1), sol(N), a(p+1);
  967:     for(size_t i = 0; i < a.size(); ++i)
  968:     {
  969:         a[i] = pow(-1.0,i);      // 1-x+x^2-x^3+x^4...
  970:     }
  971:     a[0] = 1;
  972:
  973:     auto t1 = omp_get_wtime(); // start timer
  974: // Do calculation
  975:     double ss(0.0);
  976:     for (int i = 0; i < Nloops; ++i)
  977:     {
  978:         sol = PolynomEval(a,x);
  979:         ss += sol[0];                    // prevents the optimizer from removing unus
ed calculation results.
  980:     }
  981:
  982:     auto t_diff = omp_get_wtime() - t1;  // stop timer
  983:         t_diff = t_diff/Nloops;                                      // duration per
 loop seconds
  984:
  985:     //assert(std::abs(ss/NLOOPS-sk)<1e-5);  // avoids unsafe floating point comparis
```

```
on "=="
  986:
  987: //#########################################################################
  988: // Check the correct result
  989:     cout << "\n p(x[0]) = " << sol[0] << endl;
  990:     if (static_cast<unsigned int>(sol[0]) != (static_cast<int>(a.size()) % 2))
  991:     {
  992:         cout << "  !!   W R O N G  result   !!\n";
  993:     }
  994:     cout << endl;
  995:
  996: //#########################################################################
  997: // Timings  and Performance
  998:     cout << endl;
  999:     cout.precision(2);
 1000:     cout << "p = " << p << "\t N = " << N << endl;
 1001:     cout << "Time for Nloops: " << t_diff*Nloops << endl;
 1002:     cout << "Timing in sec. : " << t_diff << endl;
 1003:     cout << "GFLOPS         : " << 2.0*(p+1)*N / t_diff / 1024 / 1024 / 1024 << endl
;
 1004:     cout << "GiByte/s       : " << N*(3+2*p) / t_diff / 1024 / 1024 / 1024 * sizeof(
a[0]) << endl;
 1005:     cout << endl << endl;
 1006:
 1007:     return;
 1008: }
 1009:
 1010:
 1011: void solver(int const & N, int const & nrhs)
 1012: {
 1013:         // generating the system matrix a
 1014:         vector<double> a(N*N);
 1015:         for(int i=0; i<N; ++i)
 1016:         {
 1017:                 for(int j=0; j<N; ++j)
 1018:                 {
 1019:                         if(i==j)
 1020:                         {
 1021:                                 a[i*N+j] = 4;
 1022:                         }
 1023:                         else
 1024:                         {
 1025:                                 a[i*N+j] = 1.0/(i-j)/(i-j);
 1026:                         }
 1027:                 }
 1028:         }
 1029:         vector<double> a_old = a;
 1030:
 1031:         // generating rhs
 1032:         vector<double> rhs(N*nrhs);
 1033:         for(int i=0; i<nrhs; ++i)
 1034:         {
 1035:                 for(int j=0; j<N; ++j)
 1036:                 {
 1037:                         rhs[i*nrhs + j] = i+j-1;
 1038:                 }
 1039:         }
 1040:         vector<double> b_old = rhs;
 1041:
 1042:         vector<int> ipiv(N);
 1043:         // factorization
 1044:         LAPACKE_dgetrf(LAPACK_ROW_MAJOR, N, N, a.data(), N, ipiv.data());
 1045:
 1046:         auto t1 = system_clock::now(); // start timer
 1047:         // Calculation of rhs -> b gets overwritten and contains the solution
 1048:         LAPACKE_dgetrs(LAPACK_ROW_MAJOR, 'N', N, nrhs, a.data(), N, ipiv.data(), rhs
.data(), nrhs);
 1049:         auto t2 = system_clock::now();  // stop timer
```

```
1050:     auto duration = duration_cast<microseconds>(t2 - t1);         // duration in micr
oseconds
1051:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
1052:     cout.precision(2);
1053:         cout << "Timing in sec. : " << t_diff << "      for nrhs= " << nrhs << endl;
1054:
1055:         // Check for correct result
1056:         vector<double> ax = MatMatProd_cblas(a_old, rhs, N);
1057:         vector<double> diff(N*nrhs);
1058:         for(int i=0; i<N*nrhs; ++i)
1059:         {
1060:                 diff[i] = ax[i] - b_old[i];
1061:         }
1062:         double diffnorm = norm_eucl(diff);
1063:         if(diffnorm > 1e-4)
1064:         {
1065:                 cout << "  !!   W R O N G  result   !!\n";
1066:         }
1067:
1068:         return;
1069: }
```

```
    1: #pragma once
    2:
    3: #include <omp.h>
    4: #include <vector>
    5:
    6: /**     Inner product
    7:         @param[in] x    vector
    8:         @param[in] y    vector
    9:         @return             resulting Euclidean inner product <x,y>
   10: */
   11: double scalar(std::vector<double> const &x, std::vector<double> const &y);
   12:
   13: /**     Inner product - parallel
   14:         @param[in] x    vector
   15:         @param[in] y    vector
   16:         @return             resulting Euclidean inner product <x,y>
   17: */
   18: double scalar_par(std::vector<double> const &x, std::vector<double> const &y);
   19:
   20:
   21: /**     Inner product with cblas
   22:         @param[in] x    vector
   23:         @param[in] y    vector
   24:         @return             resulting Euclidean inner product <x,y>
   25: */
   26: double scalar_cblas(std::vector<double> const &x, std::vector<double> const &y);
   27:
   28:
   29: /**     Inner product with Kahan summation
   30:         @param[in] x    vector
   31:         @param[in] y    vector
   32:         @return             resulting Euclidean inner product <x,y>
   33: */
   34: double scalar_kahan(std::vector<double> const &x, std::vector<double> const &y);
   35:
   36:
   37: /**     euclidean norm
   38:         @param[in] x    vector
   39:         @return             resulting Euclidean norm
   40: */
   41: double norm_eucl(std::vector<double> const &x);
   42:
   43:
   44: /**     sum of vector
   45:         @param[in] x    vector
   46:         @return             sum of the vector elements
   47: */
   48: double sum(std::vector<double> const &x);
   49:
   50:
   51: /**     sum of vector - parallel
   52:         @param[in] x    vector
   53:         @return             sum of the vector elements
   54: */
   55: double sum_par(std::vector<double> const &x);
   56:
   57:
   58: /** \brief Matrix-Vektor-Multiplikation (row-wise access)
   59:  *
   60:  * \param[in]   a   Matrix with row wise access
   61:  * \param[in]   x   vector which gets multiplied
   62:  * \return          resulting product a*x (vector)
   63:  *
   64:  */
   65: std::vector<double> MatVec(std::vector<double> const & a, std::vector<double> const
& x);
   66:
   67:
```

```
   68: /** \brief Matrix-Vektor-Multiplikation (row-wise access) - parallel
   69:  *
   70:  * \param[in]   a   Matrix with row wise access
   71:  * \param[in]   x   vector which gets multiplied
   72:  * \return          resulting product a*x (vector)
   73:  *
   74:  */
   75: std::vector<double> MatVec_par(std::vector<double> const & a, std::vector<double> co
nst & x);
   76:
   77:
   78: /** \brief Matrix-Vektor-Multiplikation mit cblas (row-wise access)
   79:  *
   80:  * \param[in]   a   Matrix with row wise access
   81:  * \param[in]   x   vector which gets multiplied
   82:  * \return          resulting product a*x (vector)
   83:  *
   84:  */
   85: std::vector<double> MatVec_cblas(std::vector<double> const & a, std::vector<double>
const & x);
   86:
   87:
   88: /** \brief Matrix-Vektor-Multiplikation (column-wise access)
   89:  *
   90:  * \param[in]   a   Matrix with row wise access
   91:  * \param[in]   x   vector which gets multiplied
   92:  * \return          resulting product a*x (vector)
   93:  *
   94:  */
   95: std::vector<double> MatVec_column(std::vector<double> const & a, std::vector<double>
 const & x);
   96:
   97:
   98: /** \brief Matrix-Matrix-Multiplikation (row-wise access)
   99:  *
  100:  * \param[in]   a   matrix with row wise access (M*L)
  101:  * \param[in]   b   matrix with row wise access (L*N)
  102:  * \param[in]   L   inner dimension of the matrix product
  103:  * \return          resulting product a*b
  104:  *
  105:  */
  106: std::vector<double> MatMatProd(std::vector<double> const & a, std::vector<double> co
nst & b, int const & L);
  107:
  108:
  109: /** \brief Matrix-Matrix-Multiplikation (row-wise access) - parallel
  110:  *
  111:  * \param[in]   a   matrix with row wise access (M*L)
  112:  * \param[in]   b   matrix with row wise access (L*N)
  113:  * \param[in]   L   inner dimension of the matrix product
  114:  * \return          resulting product a*b
  115:  *
  116:  */
  117: std::vector<double> MatMatProd_par(std::vector<double> const & a, std::vector<double
> const & b, int const & L);
  118:
  119:
  120: /** \brief Matrix-Matrix-Multiplikation mit cblas (row-wise access)
  121:  *
  122:  * \param[in]   a   matrix with row wise access (M*L)
  123:  * \param[in]   b   matrix with row wise access (L*N)
  124:  * \param[in]   L   inner dimension of the matrix product
  125:  * \return          resulting product a*b
  126:  *
  127:  */
  128: std::vector<double> MatMatProd_cblas(std::vector<double> const & a, std::vector<doub
le> const & b, int const & L);
  129:
```

```
  130:
  131: /** \brief Polynomauswertung an Stelle x
  132:  *
  133:  * \param[in]   a   Vekor mit den Koeffizienten des Polynoms a=[a0,a1,a2,...]
  134:  * \param[in]   x   Vektor, für welchen das Polynom ausgewertet werden soll
  135:  * \return          resulting vector p(x)
  136:  *
  137:  */
  138: std::vector<double> PolynomEval(std::vector<double> const & a, std::vector<double> c
onst & x);
  139:
  140:
  141: /** \brief Polynomauswertung an Stelle x - parallel
  142:  *
  143:  * \param[in]   a   Vekor mit den Koeffizienten des Polynoms a=[a0,a1,a2,...]
  144:  * \param[in]   x   Vektor, für welchen das Polynom ausgewertet werden soll
  145:  * \return          resulting vector p(x)
  146:  *
  147:  */
  148: std::vector<double> PolynomEval_par(std::vector<double> const & a, std::vector<doubl
e> const & x);
  149:
  150:
  151: /** \brief Benchmarking A - the scalar product
  152:  *
  153:  * \param   N   size of the vector
  154:  * \param   Nloops  number of iterations we want to do for the measuring
  155:  *
  156:  */
  157: void benchmark_A(int const & N, int const & Nloops);
  158:
  159:
  160: /** \brief Benchmarking A - the scalar product with cblas
  161:  *
  162:  * \param   N   size of the vector
  163:  * \param   Nloops  number of iterations we want to do for the measuring
  164:  *
  165:  */
  166: void benchmark_A_cblas(int const & N, int const & Nloops);
  167:
  168:
  169:
  170: /** \brief Benchmarking A - the scalar product with Kahan summation
  171:  *
  172:  * \param   N   size of the vector
  173:  * \param   Nloops  number of iterations we want to do for the measuring
  174:  *
  175:  */
  176: void benchmark_A_kahan(int const & N, int const & Nloops);
  177:
  178:
  179: /** \brief Benchmarking A - norm
  180:  *
  181:  * \param   N   size of the vector
  182:  * \param   Nloops  number of iterations we want to do for the measuring
  183:  *
  184:  */
  185: void benchmark_A_norm(int const & N, int const & Nloops);
  186:
  187:
  188: /** \brief Benchmarking B - matrix-vector product Ax=b (row wise access)
  189:  *
  190:  * \param   N   size of vector x
  191:  * \param   M   size of vector b (=> A: M*N)
  192:  * \param   Nloops  number of iterations we want to do for the measuring
  193:  *
  194:  */
  195: void benchmark_B(int const & N, int const & M, int const & Nloops);
```

```
  196:
  197:
  198: /** \brief Benchmarking B - matrix-vector product Ax=b (row wise access) [parallel]
  199:  *
  200:  * \param   N   size of vector x
  201:  * \param   M   size of vector b (=> A: M*N)
  202:  * \param   Nloops  number of iterations we want to do for the measuring
  203:  *
  204:  */
  205: void benchmark_B_par(int const & N, int const & M, int const & Nloops);
  206:
  207:
  208: /** \brief Benchmarking B - matrix-vector product Ax=b with cblas (row wise access)
  209:  *
  210:  * \param   N   size of vector x
  211:  * \param   M   size of vector b (=> A: M*N)
  212:  * \param   Nloops  number of iterations we want to do for the measuring
  213:  *
  214:  */
  215: void benchmark_B_cblas(int const & N, int const & M, int const & Nloops);
  216:
  217:
  218: /** \brief Benchmarking B - matrix-vector product Ax=b (column wise access)
  219:  *
  220:  * \param   N   size of vector x
  221:  * \param   M   size of vector b (=> A: M*N)
  222:  * \param   Nloops  number of iterations we want to do for the measuring
  223:  *
  224:  */
  225: void benchmark_B_column(int const & N, int const & M, int const & Nloops);
  226:
  227:
  228: /** \brief Benchmarking C - Matrix-Matrix product C=A*B A_M*L, B_L*N
  229:  *
  230:  * \param   N
  231:  * \param   M
  232:  * \param   L
  233:  * \param   Nloops  number of iterations we want to do for the measuring
  234:  *
  235:  */
  236: void benchmark_C(int const & N, int const & M, int const & L, int const & Nloops);
  237:
  238:
  239: /** \brief Benchmarking C - Matrix-Matrix product C=A*B A_M*L, B_L*N [parallel]
  240:  *
  241:  * \param   N
  242:  * \param   M
  243:  * \param   L
  244:  * \param   Nloops  number of iterations we want to do for the measuring
  245:  *
  246:  */
  247: void benchmark_C_par(int const & N, int const & M, int const & L, int const & Nloops
);
  248:
  249:
  250: /** \brief Benchmarking C - Matrix-Matrix product with cblas; C=A*B A_M*L, B_L*N
  251:  *
  252:  * \param   N
  253:  * \param   M
  254:  * \param   L
  255:  * \param   Nloops  number of iterations we want to do for the measuring
  256:  *
  257:  */
  258: void benchmark_C_cblas(int const & N, int const & M, int const & L, int const & Nloo
ps);
  259:
  260:
  261: /** \brief Benchmarking D - polynomial evaluation
```

```
262:  *
263:  * \param   p   the degree of the polynomial
264:  * \param   N   size of the input vector x where p(x)
265:  * \param   Nloops  number of iterations we want to do for the measuring
266:  *
267:  */
268: void benchmark_D(int const & p, int const & N, int const & Nloops);
269:
270:
271: /** \brief Benchmarking D - polynomial evaluation [parallel]
272:  *
273:  * \param   p   the degree of the polynomial
274:  * \param   N   size of the input vector x where p(x)
275:  * \param   Nloops  number of iterations we want to do for the measuring
276:  *
277:  */
278: void benchmark_D_par(int const & p, int const & N, int const & Nloops);
279:
280:
281: /** \brief solving system of linear equations with time measurement
282:  *
283:  * \param   N   size of the system Ax=b with A(nxn)
284:  * \param       nrhs    number of right hand sides b(nxnrhs)
285:  *
286:  */
287: void solver(int const & N, int const & nrhs);
288:
```

```cpp
#include "bsp_3_lib_bench_par.h"
#include <cmath>
#include <iostream>

using namespace std;

int main()
{
    benchmark_B(8000,8000,400);
    benchmark_B_par(8000,8000,400);
    benchmark_C(4000,4000,4000,1);
    benchmark_C_par(4000,4000,4000,1);
    benchmark_D(1e4,1e5,15);
    benchmark_D_par(1e4,1e5,15);

    // comparing the time for sum and inner product with and without parallelization
    cout << "\nComparing the runtime (in sec) for inner product and sum with and without parallelization\n\n";
    for(int k=3; k<=8; ++k)
    {

                int N = pow(10,k);
                cout << "k = " << k << "        N = " << N << endl;
                vector<double> v1(N,1.0/N);
                vector<double> v2(N,N);

                auto tstart = omp_get_wtime();
                double s = scalar(v1, v2);
                auto t_diff_scalar = omp_get_wtime() - tstart;

                tstart = omp_get_wtime();
                double sp = scalar_par(v1, v2);
                auto t_diff_scalar_par = omp_get_wtime() - tstart;

                tstart = omp_get_wtime();
                double su = sum(v1);
                auto t_diff_sum = omp_get_wtime() - tstart;

                tstart = omp_get_wtime();
                double sup = sum_par(v1);
                auto t_diff_sum_par = omp_get_wtime() - tstart;

                cout << "sum    " << t_diff_sum << "    inner_prod      " << t_diff_scalar << endl;
                cout << "sum_par        " << t_diff_sum_par << "        inner_prod_par      " << t_diff_scalar_par << endl << endl;;
        }

    return 0;
}
```