```
    1: #include "bsp_5_3_lib.h"
    2: #include "mayer_primes.h"
    3: #include "algorithm"
    4: #include <iostream>
    5: #ifdef _OPENMP
    6: #include <omp.h>
    7: #endif
    8:
    9: using namespace std;
   10:
   11: int single_goldbach(const int &k)
   12: {
   13:     vector<int> primes = get_primes(k);
   14:     int amount = 0;
   15:     for(size_t it = 0; primes[it]<=k/2.0; ++it) //für Primzahl größer als k/2 haben
wir bereits Zerlegung gezählt: 3+7 = 7+3
   16:     {
   17:        for(size_t j = it; j<primes.size(); ++j)
   18:        {
   19:            if(primes[it] + primes[j] == k)
   20:            {
   21:                amount += 1;
   22:            }
   23:        }
   24:     }
   25:
   26:     return amount;
   27: }
   28:
   29:
   30: int single_goldbach_par(const int &k)
   31: {
   32:     vector<int> primes = get_primes(k);
   33:     int amount = 0;
   34:     size_t it_max = 0;
   35:     // to get a barrier for the loop; omp needs a fix upper bound
   36:         while (it_max < primes.size() && primes[it_max] <= k/2)
   37:     {
   38:                 ++it_max;
   39:         }
   40:
   41: #pragma omp parallel for default(none) shared(primes, k, it_max) reduction(+:amount)
   42:     for(size_t it = 0; it<it_max; ++it) //für Primzahl größer als k/2 haben wir bere
its Zerlegung gezählt: 3+7 = 7+3
   43:     {
   44:        for(size_t j = it; j<primes.size(); ++j)
   45:        {
   46:            if(primes[it] + primes[j] == k)
   47:            {
   48:                amount += 1;
   49:            }
   50:        }
   51:     }
   52:
   53:     return amount;
   54: }
   55:
   56:
   57: vector<int> count_goldbach(const int &n)
   58: {
   59:     vector<int> count_vec((n-4)/2+1,0);
   60:     vector<int> primes = get_primes(n);
   61:     for(size_t k=0; k < primes.size() && primes[k]<=n/2; ++k)
   62:     {
   63:         for(size_t i=k; i<primes.size(); ++i)
   64:         {
   65:             int sum = primes[k] + primes[i];
   66:             if(sum<=n)
```

use stel::lower_bound(..) and distance(...) for it_max =

✓

```
    67:             {
    68:                 count_vec[(sum-4)/2] += 1;
    69:             }
    70:         }
    71:     }
    72:
    73:     return count_vec;
    74: }
    75:
    76:
    77: vector<int> count_goldbach_par(const int &n)
    78: {
    79:     vector<int> count_vec((n-4)/2+1,0);
    80:     vector<int> primes = get_primes(n);
    81:     size_t it_max = 0;
    82:     while (it_max < primes.size() && primes[it_max] <= n/2)
    83:     {
    84:                 ++it_max;
    85:         }
    86:
    87: #pragma omp parallel for schedule(dynamic) shared(primes, n, it_max) reduction(VecAd
d:count_vec)
    88:     for(size_t k=0; k<it_max; ++k)
    89:     {
    90:         for(size_t i=k; i<primes.size(); ++i)
    91:         {
    92:             int sum = primes[k] + primes[i];
    93:             if(sum<=n)
    94:             {
    95:                 count_vec[(sum-4)/2] += 1;
    96:             }
    97:         }
    98:     }
    99:
   100:     return count_vec;
   101: }
```

```
    1: #pragma once
    2:
    3: #include <cassert>
    4: #include <omp.h>
    5: #include <vector>
    6:
    7: /** \brief Zaehlt fuer eine gegebene Zahl @k die Anzahl der moeglichen Zerlegungen a
ls Summe zweier Primzahlen
    8:  *
    9:  * \param[in]   k   fuer diese Zahl wird die Anzahl der Zerlegungen berechnet
   10:  * \return  Anzahl der Zerlegungen
   11:  *
   12:  */
   13: int single_goldbach(const int &k);
   14:
   15:
   16: /** \brief Zaehlt fuer eine gegebene Zahl @k die Anzahl der moeglichen Zerlegungen a
ls Summe zweier Primzahlen [parallel]
   17:  *
   18:  * \param[in]   k   fuer diese Zahl wird die Anzahl der Zerlegungen berechnet
   19:  * \return  Anzahl der Zerlegungen
   20:  *
   21:  */
   22: int single_goldbach_par(const int &k);
   23:
   24:
   25: /** \brief Zaehlt die Anzahl der Dekomposition für alle geraden Zahlen in [4,n]
   26:  *
   27:  * \param[in]   n   obere Intervallgrenze (in diesem Bereich werden die Dekompositio
nen berechnet
   28:  * \return Vektor mit den Anzahl der Dekompositionen (ad Adressierung: x[0] = (n=4),
 x[1] = (n=6), x[2] = (n=8), Umrechnung *2 + 4
   29:  *
   30:  */
   31: std::vector<int> count_goldbach(const int &n);
   32:
   33:
   34: /** \brief Zaehlt die Anzahl der Dekomposition für alle geraden Zahlen in [4,n] - [p
arallel]
   35:  *
   36:  * \param[in]   n   obere Intervallgrenze (in diesem Bereich werden die Dekompositio
nen berechnet
   37:  * \return Vektor mit den Anzahl der Dekompositionen (ad Adressierung: x[0] = (n=4),
 x[1] = (n=6), x[2] = (n=8), Umrechnung *2 + 4
   38:  *
   39:  */
   40: std::vector<int> count_goldbach_par(const int &n);
   41:
   42:
   43: /**      Vector @p b adds its elements to vector @p a .
   44:         @param[in] a    vector
   45:         @param[in] b    vector
   46:         @return             a+=b componentwise
   47: */
   48: template<class T>
   49: std::vector<T> &operator+=(std::vector<T> &a, std::vector<T> const &b)
   50: {
   51:     assert(a.size()==b.size());
   52:     for (size_t k = 0; k < a.size(); ++k) {
   53:         a[k] += b[k];
   54:     }
   55:     return a;
   56: }
   57:
   58: #pragma omp declare reduction(VecAdd : std::vector<int>  : omp_out += omp_in) \
   59:  initializer (omp_priv=omp_orig)
```

```cpp
#include "bsp_5_3_lib.h"
#include "mayer_primes.h"
#include <iostream>
#include <algorithm>
#include <chrono>
// BSP 5_3 Goldbach conjunction
using namespace std;

int main()
{
        omp_set_num_threads(8);

        cout << "\nChecking for correct result\n";
        cout << "694 has " << single_goldbach_par(694) << " decompositions" << endl;

    // Auswertung für n=100000 bzw herausfinden der Zahl, welche die meisten Dekompositionen hat
    vector<int> v = count_goldbach_par(1e5);
    auto ip = max_element(v.begin(), v.end());
    cout << "number in [4,1e5] with most decompositions: " << distance(v.begin(), ip)*2+4 << " has " << *ip << " decompositions" << endl;

    cout << "\nTiming for parallel" << endl;
    vector<int> nvec{static_cast<int>(1e4), static_cast<int>(1e5), static_cast<int>(4*1e5), static_cast<int>(1e6), static_cast<int>(2*1e6)}; // Vektor für n
    for(size_t k=0; k<nvec.size(); ++k)
    {
        auto timestart = omp_get_wtime();
        vector<int> vall = count_goldbach_par(nvec[k]);
        auto time = omp_get_wtime() - timestart;
        cout << "n = " << nvec[k] << "\ttime in s: " << time << endl;
    }

    cout << "\nTiming for serial" << endl;
    for(size_t k=0; k<nvec.size(); ++k)
    {
        auto timestart = omp_get_wtime();
        vector<int> vall = count_goldbach(nvec[k]);
        auto time = omp_get_wtime() - timestart;
        cout << "n = " << nvec[k] << "\ttime in s: " << time << endl;
    }

    // for large n the parallel version is slower than the sequential
    // the reason is the reduction(VecAdd) which is slow especially for large vectors

//    cout << single_goldbach(694) << endl;
//    cout << count_goldbach(10000)[690/2] << endl;
    return 0;
}
```

```
    1: #pragma once
    2:
    3: #include <cstring> //memset
    4: #include <vector>
    5: //using namespace std;
    6:
    7: /** \brief Determines all prime numbers in interval [2, @p max].
    8:  *
    9:  *  The sieve of Eratosthenes is used.
   10:  *
   11:  *  The implementation originates from <a href="http://code.activestate.com/recipes/
576559-fast-prime-generator/">Florian Mayer</a>.
   12:  *
   13:  * \param[in]   max end of interval for the prime number search.
   14:  * \return  vector of prime numbers @f$2,3,5, ..., p<=max @f$.
   15:  *
   16:  * \copyright
   17:  * Copyright (c) 2008 Florian Mayer          (adapted by Gundolf Haase 2018)
   18:  *
   19:  * Permission is hereby granted, free of charge, to any person obtaining a copy
   20:  * of this software and associated documentation files (the "Software"), to deal
   21:  * in the Software without restriction, including without limitation the rights
   22:  * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
   23:  * copies of the Software, and to permit persons to whom the Software is
   24:  * furnished to do so, subject to the following conditions:
   25:  *
   26:  * The above copyright notice and this permission notice shall be included in
   27:  * all copies or substantial portions of the Software.
   28:  *
   29:  *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLI
ED,
   30:  * INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   31:  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
   32:  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
   33:  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
   34:  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOF
TWARE.
   35:  *
   36:  */
   37: template <class T>
   38: std::vector<T> get_primes(T max)
   39: {
   40:     std::vector<T> primes;
   41:     char *sieve;
   42:     sieve = new char[max / 8 + 1];
   43:     // Fill sieve with 1
   44:     memset(sieve, 0xFF, (max / 8 + 1) * sizeof(char));
   45:     for (T x = 2; x <= max; x++)
   46:     {
   47:         if (sieve[x / 8] & (0x01 << (x % 8))) {
   48:             primes.push_back(x);
   49:             // Is prime. Mark multiplicates.
   50:             for (T j = 2 * x; j <= max; j += x)
   51:             {
   52:                 sieve[j / 8] &= ~(0x01 << (j % 8));
   53:                         }
   54:         }
   55:         }
   56:     delete[] sieve;
   57:     return primes;
   58: }
   59:
   60: //----------------------------------------------------------------
   61: //int main()        // by  Florian Mayer
   62: //{g++ -O3  -std=c++14 -fopenmp main.cpp && ./a.out
   63: //    vector<unsigned long> primes;
   64: //    primes = get_primes(10000000);
   65: //    // return 0;
```

```
66: //    // Print out result.
67: //    vector<unsigned long>::iterator it;
68: //    for(it=primes.begin(); it < primes.end(); it++)
69: //        cout << *it << " ";
70: //
71: //    cout << endl;
72: //    return 0;
73: //}
74:
```