SuperCon2023 本選問題

1　最近点対探索

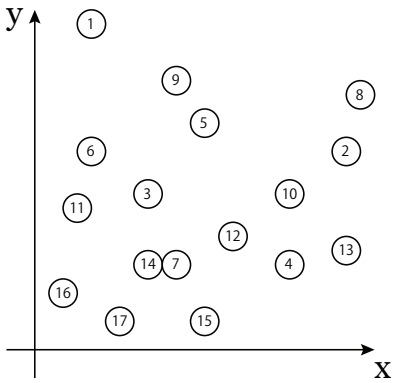

図 1: ２次元空間に分布する点

２次元空間に分布した点の中で最も距離の近いペアを見つける問題を考える。ただし、距離の定義はユークリッド距離とする。点 i の座標を (x_i, y_i)、点 j の座標を (x_j, y_j) とすると、それらの間の距離 d は以下の式で表される。

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{1}$$

図 1 に示すように、それぞれの点には番号がランダムに振られている。この場合は 7 と 14 が最も近いペアとなっている。素朴なアルゴリズムでこのペアを見つけようとすると、N 個の点に対して、N が大変大きくなると、N^2 にほぼ比例するような計算の手間を要する。こういう計算の手間がかかることを「計算量が N^2 に比例する」と呼ぶことがある。分割統治法を用いるとこれを $N \log N$ に比例する計算量に低減できることが知られている。

近傍探索は様々な物理シミュレーションで粒子の相互作用を計算するときに用いられる。例えば、分子同士の相互作用ではファンデルワールス力は近傍の分子の間でしか働かないため、近傍探索が必要になる。クーロン力や重力などのような長距離力の計算でも、近傍の粒子同士の計算は厳密に行い、遠方の粒子同士の計算は近似的に行うため、近傍の粒子を特定する必要がある。

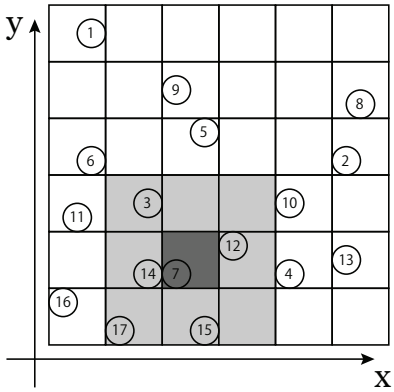

図 2: 空間をブロック分割して近傍を探索する方法

2　ブロック分割

低次元空間の最近点対探索では空間分割が有効である。図 2 のように空間全体を細かい箱に分割し、近傍の 9 つの箱を探索することで最近点対の候補を取得できる。7 の点がある箱の周りには 3, 12, 13, 15, 17 の点があるため、これらの点と 7 の点の距離を計算して最小になるものを求めれば良い。この操作を全ての箱に対して行うことで全ての点に関する最近点対探索が可能となる。図 2 の例では、全ての箱に 1 つずつしか点が存在していないが、もう少し大きな箱に分割して一気に複数の点を探索することもできる。このとき、空でない箱のリストとその中に存在する点のリストがあればよい。ただし、箱の中に存在する点のリストを連結リストなどで作ってしまうとランダムメモリアクセスが大量に発生してしまうため、箱ごとに点をソートしておくと良い。そうすれば、箱の中に存在する点の指標の始点と終点だけ保存しておけば、箱の中の点を全て取得することが可能となる。箱ごとに点をソートするには、箱に番号を振っておき、それをキーとしてバケットソートや基数ソートでソートする方法などが考えられる。空間全体をどのくらい細かい箱に分割するかを考える際に、ソートにかかる計算量と探索にかかる計算量のバランスを気にする必要がある。箱を細かく分割しすぎるとソートのキーの数が増え、箱が大きすぎると探索しないといけない近傍領域が増える。

3　問題

スーパーコン 2023 の本選では、２次元空間に分布した点の最近点対探索を課題とする。上記のような解き方を例示したが、全く違った高速解法も考えられるかもしれない。以下に計算方法の詳細を示す。

3.1　計算方法

1. 与えられたコードを用いて $N = 500{,}000{,}000$ 個の点の座標と番号を計算する
2. 最近点対探索を行い 1 番目、10 番目、100 番目、1000 番目に近い点対を計算し、それらの番号と距離を出力する

3. それぞれの番号と距離が正しければ有効な計算とする
4. そのとき最近点対探索にかかった時間のみを計測し、その時間を競う

3.2 練習と本番環境に関する注意点

- 練習ではこちらで提供するコードを使って初期条件を生成し上記の手順に従い最近点対探索にかかった時間を計測する
- 本番ではこれとは異なる初期条件をこちらで指定した乱数シードを使って生成し上記の手順に従い最近点対探索にかかった時間を計測する

3.3 プログラミング言語

プログラムは C++(C++17) で書く。インラインアセンブリを直接記述すること、出力されたものを改変することは認めない。OpenMP によるスレッド並列化も用いてよい。プログラムの分割はファイル数は特に指定しない。使用できる外部のヘッダはシステムが標準で提供するもののみとし、ライブラリの使用は認めない。なお、標準ヘッダーファイルは改変してはならない。これらの注意に反するプログラムは失格とする。

3.4 ヘッダファイルとテンプレートファイル

付録 A に示すヘッダファイル `sc_header.hpp` を用いて、以下のように main 関数を記述すること。ただし、main 関数の内部において `sc::input(argc, argv);` より前と `sc::output();` より後の記述を改変してはならない。`argv[1]` と `argv[2]` が `N` と `seed` に予約されていて `argv[3]` 以降が自由に使えるようになっている。`sc::genrand()` の終わりから `sc::output()` の始めまでの時間を計算時間とする。回答の提出は `sc::pairs` に値を格納し `sc::output()` を呼び出すことで行う。

```
#include <sc_header.hpp>

int main(int argc, char *argv[]){
  sc::input(argc, argv);
  /* sc::N の情報を用いてメモリ確保を行っても良い */
  sc::genrand();
  /* ここに自由にプログラムを記述（main 関数の外で定義した関数を呼出しても良い） */
  sc::output();
  sc::finalize();
  return 0;
}
```

3.5 コンパイルの方法

本選の実行環境は「富岳」を用いる。コンパイラは「富岳」で標準的に提供されている FCC 4.9.0 を用いる。
プログラム名を main.cpp として、MPI と OpenMP を使用する場合、
`mpiFCC -Nclang -Kfast,openmp -stdlib=libc++ --std=c++17 main.cpp`
MPI を使用せず OpenMP だけ使用する場合、
`FCC -Nclang -Kfast,openmp -stdlib=libc++ --std=c++17 main.cpp`
でコンパイルする。

3.6 ジョブスクリプト

今回のコンテストでは OpenMP のスレッド数や MPI の並列数を実行時に指定する必要があるので、ジョブスクリプトも提出してもらう。MPI を使う場合のジョブスクリプトは以下のようになる。elapse は実行時間制限 10 分の指定である。C++ 言語かによって、適切なコンパイルコマンドを残す。指定しなくてはならないのは 5 行目の MPI 並列数 (この例では 4) と 8 行目の OpenMP スレッド数 (この例では 12) である。実行時には標準入力から問題を読み、標準出力に結果を出す。

```
#!/bin/bash
#PJM -L "node=1"
#PJM -L "elapse=0:10:00"
#PJM --mpi "proc=4"
#PJM -j
#PJM --rsc-list "freq=2200"
#PJM --rsc-list "retention_state=0"
#PJM -S
export OMP_NUM_THREADS=12
export PLE_MPI_STD_EMPTYFILE=off
mpiFCC -Nclang -Kfast,openmp -stdlib=libc++ --std=c++17 main.cpp
mpirun -np 4 ./a.out 500000000 $SC_SEED
```

また、MPI を使わない場合にはコンパイルと実行の仕方が異なり、以下のよう になる。OpenMP のスレッド数は 1 ノードの最大である 48 としてある。

```
#!/bin/bash
#PJM -L "node=1"
#PJM -L "elapse=0:10:00"
#PJM -j
#PJM --rsc-list "freq=2200"
```

```
#PJM --rsc-list "retention_state=0"
#PJM -S
export OMP_NUM_THREADS=48
export PLE_MPI_STD_EMPTYFILE=off
FCC -Nclang -Kfast,openmp -stdlib=libc++ --std=c++17 main.cpp
./a.out 500000000 $SC_SEED
```

3.7　採点方法

1. 1 番目、10 番目、100 番目、1000 番目の点対の番号がひとつでも間違った場合は最下位
2. それぞれの距離は丸め誤差のみ許容（相対誤差が 10^{-14} 以内）
3. 時間計測はこちらで 5 回行い中央値を用いる
4. マクロ変数やヘッダファイルの改ざんが発覚した場合は失格

付録 A

A.1 ヘッダファイル

座標 X, Y を格納するための配列と点対を記述するための構造体及びその 1000 番目までのエントリーを格納する配列は以下のヘッダファイルで定義されている。関数 `input()` では乱数生成器を用いて X,Y の配列を初期化している。関数 `output()` では、`pairs` に格納された点対の情報を用いて出力するため、プログラムでは必ずこの `pairs` に最近傍点対の上位 1000 個を格納しなくてはならない。時間は関数 `gettimeofday()` を用いて `input()` の最後から `output()` の最初までの時間を計測するようになっている。

```
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <cassert>
#include <random>
#include <sys/time.h>
#include <omp.h>

#define SC23_MAGIC 551 // 秘密の整数、本番では別の値をを使う

namespace sc {
  constexpr int maxThreads = 48;
```

```
  constexpr int ranking = 1000;
  double TIME0;
  double *X, *Y;
  long N = 1024;
  long Nloc = N; // MPI のときのそのプロセスの粒子数
  long seed = 1;
  int mpi_size  = 1;
  int mpi_rank  = 0;
  int mpi_idx_off = 0;

  template <size_t key> double &time0(){
          static double pre[key % 256]; // アドレスのランダム化
          static double t;
          static double post[255 - key % 256];
          return t;
  }

  struct Pair {
    int i, j;
    double dist2;

    bool operator<(const Pair &rhs) const {
      return dist2 < rhs.dist2;
    }
  };
  Pair *pairs;

  double get_time() {
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return (double)(tv.tv_sec+tv.tv_usec*1e-6);
  }

  void input(int &argc, char **&argv) {
#ifdef MPI_VERSION
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
#endif
    if(argc > 1)  N    = atol(argv[1]);
    if(argc > 2)  seed = atol(argv[2]);
```

```
    int pbeg = (mpi_rank + 0) * maxThreads / mpi_size;
    int pend = (mpi_rank + 1) * maxThreads / mpi_size;
    long ibeg = pbeg * N / maxThreads;
    long iend = pend * N / maxThreads;
    Nloc = iend - ibeg;
    mpi_idx_off = ibeg;
    X = new double [Nloc];
    Y = new double [Nloc];
    pairs = new Pair [ranking];
  }

  void genrand(){
    std::uniform_real_distribution<double> dis(0.0, 1.0);
    int pbeg = (mpi_rank + 0) * maxThreads / mpi_size;
    int pend = (mpi_rank + 1) * maxThreads / mpi_size;
#pragma omp parallel for
    for (int ib=pbeg; ib<pend; ib++) {
      std::mt19937_64 generator((1+ib) * seed * SC23_MAGIC);
      int begin = (ib + 0) * N / maxThreads;
      int end   = (ib + 1) * N / maxThreads;
      if(ib == maxThreads-1) end = N > end ? N : end;
      for (int i=begin; i<end; i++) {
        X[i - mpi_idx_off] = dis(generator);
        Y[i - mpi_idx_off] = dis(generator);
      }
    }
    time0<SC23_MAGIC>() = get_time();
  }

  void output() {
#ifdef MPI_VERSION
    int mpi_rank  = 0;
        MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
        if(mpi_rank > 0) return;
#endif
    printf("!%d: %e s\n", SC23_MAGIC, get_time() - time0<SC23_MAGIC>());
    for(size_t k : {0, 9, 99, 999}) {
      const Pair &pp = pairs[k];
      int i = pp.i;
      int j = pp.j;
```

```
      if(i > j) std::swap(i, j);
      printf("!%d: #%4zu : %10.15e (%d,%d)\n", SC23_MAGIC, k+1, sqrt(pp.dist2), i, j);
    }
  }

  void finalize() {
    delete[] X;
    delete[] Y;
    delete[] pairs;
#ifdef MPI_VERSION
    MPI_Finalize();
#endif
  }
};

#undef SC23_MAGIC
```

A.2 サンプルコード (素朴な最近点探索法)

以下に、素朴な方法で最近点対探索を行うサンプルコードを示す。

```
// #include <mpi.h> // MPI を使う場合は必ず sc_header より前に
#include "sc_header.hpp"
#include <queue>

int main(int argc, char **argv){
  sc::input(argc, argv);
  using sc::N;
  assert(sc::N == sc::Nloc);
  sc::genrand();

  std::priority_queue<sc::Pair> queue;
  double r2_max = 2.0;
  for(int i=0; i<N; i++){
    for(int j=i+1; j<N; j++){
      double dx = sc::X[j] - sc::X[i];
      double dy = sc::Y[j] - sc::Y[i];
      double r2 = dx*dx;
      r2 += dy * dy;
```

```
      if(r2 < r2_max){
        if(queue.size() < 1000){
          queue.push({i, j, r2});
        }else if (r2 < queue.top().dist2) {
          queue.pop();
          queue.push({i, j, r2});
          r2_max = queue.top().dist2;
        }
      }
    }
  }
  for(int i=0; i<1000; i++){
    sc::pairs[999-i] = queue.top();
    queue.pop();
  }

  sc::output();
  sc::finalize();
  return 0;
};
```

A.3 サンプルコード (素朴な最近点探索法の MPI 版)

以下に、素朴な方法で最近点対探索を行うサンプルコードの MPI 版を示す。

```
#include <mpi.h> // MPI を使う場合は必ず sc_header より前に
#include "sc_header.hpp"
#include <queue>

struct Point{
  double x, y;
  int index, iproc;
};

int main(int argc, char **argv){
  sc::input(argc, argv);
  using sc::N;
  using sc::Nloc;
```

```
Point *porg = new Point[sc::Nloc];
const size_t palloc = sc::Nloc * 2.2 + 1000;
Point *ploc = new Point[palloc];
assert(sizeof(Point)==24 && sizeof(long)==8);
MPI_Datatype MPI_POINT;
MPI_Type_contiguous(3, MPI_LONG, &MPI_POINT);
MPI_Type_commit(&MPI_POINT);

sc::genrand();

const int np = sc::mpi_size;
int counts[np], displs[np+1];
for(auto &ic : counts) ic = 0;
// count and...
for(int i=0; i<Nloc; i++){
  int iproc = sc::X[i] * np;
  assert(0 <= iproc && iproc < np);
  counts[iproc]++;
}
for(int i=0, sum=0; i<=np; i++){
  displs[i] = sum;
  sum += counts[i];
}
// then sort
int dstloc[np];
for(int i=0; i<np; i++){
  dstloc[i] = displs[i];
}
for(int i=0; i<Nloc; i++){
  int iproc = sc::X[i] * np;
  Point p = {sc::X[i], sc::Y[i], i + sc::mpi_idx_off, iproc};
  porg[dstloc[iproc]++] = p;
}

// 粒子分割の準備
int rcounts[np], rdispls[np+1];
MPI_Alltoall(counts, 1, MPI_INT, rcounts, 1, MPI_INT, MPI_COMM_WORLD);
for(int i=0, sum=0; i<=np; i++){
  rdispls[i] = sum;
  sum += rcounts[i];
}
```

```
int nloc = rdispls[np]; // new Nloc after all2all
assert(nloc <= palloc);
MPI_Alltoallv(porg, counts,  displs,  MPI_POINT,
              ploc, rcounts, rdispls, MPI_POINT, MPI_COMM_WORLD);

for(int i=0; i<nloc; i++){
  assert(ploc[i].iproc == sc::mpi_rank);
  assert(ploc[i].iproc == (int)(ploc[i].x * np));
}

int nrecv = 0;
if(sc::mpi_rank > 0){
  MPI_Send(&nloc,  1, MPI_INT, sc::mpi_rank-1, 10, MPI_COMM_WORLD);
}
if(sc::mpi_rank < np-1){
  MPI_Recv(&nrecv, 1, MPI_INT, sc::mpi_rank+1, 10, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
}
assert(nloc + nrecv <= palloc);

if(sc::mpi_rank > 0){
  MPI_Send(&ploc[0],    nloc,  MPI_POINT, sc::mpi_rank-1, 11, MPI_COMM_WORLD);
}
if(sc::mpi_rank < np-1){
  MPI_Recv(&ploc[nloc], nrecv, MPI_POINT, sc::mpi_rank+1, 11, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
}

const int nloc_tot = nloc + nrecv;
for(int i=nloc; i<nloc_tot; i++){
  assert(ploc[i].iproc == sc::mpi_rank + 1);
}

std::priority_queue<sc::Pair> queue;
double r2_max = 2.0;
for(int i=0; i<nloc; i++){
  for(int j=i+1; j<nloc_tot; j++){
    double dx = ploc[j].x - ploc[i].x;
    double dy = ploc[j].y - ploc[i].y;
    double r2 = dx*dx;
    r2 += dy * dy;
```

```
    if(r2 < r2_max){
      int ii = std::min(ploc[i].index, ploc[j].index);
      int jj = std::max(ploc[i].index, ploc[j].index);
      if(queue.size() < 1000){
        queue.push({ii, jj, r2});
      }else if (r2 < queue.top().dist2) {
        queue.pop();
        queue.push({ii, jj, r2});
        r2_max = queue.top().dist2;
      }
    }
  }
}

if(0 == sc::mpi_rank){ //受信して queue を merge
  for(int isrc=1; isrc<sc::mpi_size; isrc++){
    MPI_Recv(sc::pairs, 1000*sizeof(sc::Pair), MPI_BYTE, isrc, isrc,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for(int i=0; i<1000; i++){
      if (sc::pairs[i] < queue.top()) {
        queue.pop();
        queue.push(sc::pairs[i]);
      }
    }
  }
  for(int i=0; i<1000; i++){
    sc::pairs[999-i] = queue.top();
    queue.pop();
  }
  sc::output();
}else{ // queue を root に送信
  for(int i=0; i<1000; i++){
    sc::pairs[999-i] = queue.top();
    queue.pop();
  }
  MPI_Send(sc::pairs, 1000*sizeof(sc::Pair), MPI_BYTE, 0,
           sc::mpi_rank, MPI_COMM_WORLD);
}

sc::finalize();
return 0;
```

```
}
```

A.4 サンプルコード (ブロック分割を用いた最近点探索法)

以下に、ブロック分割を用いた最近点対探索を行うサンプルコードを示す。

```
#include "sc_header.hpp"
#include <vector>
#include <algorithm>

int main(int argc, char* argv[]) {
  sc::input(argc, argv);
  using sc::N;

  const int level = argc>3 ? atoi(argv[3]) : log(N)/2;
  const int Nx = 1 << level;
  const int range = Nx * Nx;
  int *key = new int [N];
  int *bucket = new int [range];
  int **bucketPerThread = new int* [sc::maxThreads];
  for (int i=0; i<sc::maxThreads; i++)
    bucketPerThread[i] = new int [range];
  int *offset = new int [range+1];
  int *permutation = new int [N];
  double *X2 = new double [N];
  double *Y2 = new double [N];

  sc::genrand();
#pragma omp parallel for
  for (int i=0; i<N; i++) {
    int ix = sc::X[i] * Nx;
    int iy = sc::Y[i] * Nx;
    int k = 0;
    for (int l=0; l<level; l++) {
       k |= (iy & 1 << l) <<  l;
       k |= (ix & 1 << l) << (l + 1);
    }
    key[i] = k;
  }
```

```
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int threads = omp_get_num_threads();
    for (int i=0; i<range; i++)
      bucketPerThread[tid][i] = 0;
#pragma omp for
    for (int i=0; i<N; i++)
      bucketPerThread[tid][key[i]]++;
#pragma omp single
    for (int t=1; t<threads; t++)
      for (int i=0; i<range; i++)
        bucketPerThread[t][i] += bucketPerThread[t-1][i];
#pragma omp for
    for (int i=0; i<range; i++)
      bucket[i] = bucketPerThread[threads-1][i];
#pragma omp single
    for (int i=1; i<range; i++)
      bucket[i] += bucket[i-1];
    offset[0] = 0;
#pragma omp for
    for (int i=0; i<range; i++)
      offset[i+1] = bucket[i];
#pragma omp for
    for (int i=0; i<N; i++) {
      bucketPerThread[tid][key[i]]--;
      int inew = offset[key[i]] + bucketPerThread[tid][key[i]];
      permutation[inew] = i;
    }
  }

#pragma omp parallel for
  for (int i=0; i<N; i++) {
    X2[i] = sc::X[permutation[i]];
    Y2[i] = sc::Y[permutation[i]];
  }
  int minI[sc::ranking*sc::maxThreads],minJ[sc::ranking*sc::maxThreads];
  double minD[sc::ranking*sc::maxThreads];
  for (int i=0; i<sc::ranking*sc::maxThreads; i++) {
    minI[i] = minJ[i] = 0;
    minD[i] = 1;
```

```
  }
#pragma omp parallel for
  for (int i=0; i<range; i++) {
    int threadOffset = sc::ranking*omp_get_thread_num();
    int ix = 0;
    int iy = 0;
    for (int l=0; l<level; l++) {
      iy |= (i & 1 <<  2 * l)      >>  l;
      ix |= (i & 1 << (2 * l + 1)) >> (l + 1);
    }
    int minjx = 0 > ix-1 ? 0 : ix-1;
    int maxjx = ix+1 < Nx-1 ? ix+1 : Nx-1;
    int minjy = 0 > iy-1 ? 0 : iy-1;
    int maxjy = iy+1 < Nx-1 ? iy+1 : Nx-1;
    for (int jx=minjx; jx<=maxjx; jx++) {
      for (int jy=minjy; jy<=maxjy; jy++) {
        int j = 0;
        for (int l=0; l<level; l++) {
           j |= (jy & 1 << l) <<  l;
           j |= (jx & 1 << l) << (l + 1);
        }
        if(j < i) continue;
        for (int ii=offset[i]; ii<offset[i+1]; ii++) {
          for (int jj=offset[j]; jj<offset[j+1]; jj++) {
            double dd = (X2[ii] - X2[jj]) * (X2[ii] - X2[jj])
                      + (Y2[ii] - Y2[jj]) * (Y2[ii] - Y2[jj]);
            if (minD[sc::ranking-1 + threadOffset] > dd && ii < jj) {
              int k = sc::ranking-1;
              while (k>0 && dd < minD[k-1 + threadOffset]) {
                minI[k + threadOffset] = minI[k-1 + threadOffset];
                minJ[k + threadOffset] = minJ[k-1 + threadOffset];
                minD[k + threadOffset] = minD[k-1 + threadOffset];
                k--;
              }
              minI[k + threadOffset] = ii;
              minJ[k + threadOffset] = jj;
              minD[k + threadOffset] = dd;
            }
          }
        }
      }
```

```
    }
  }

  int threads;
  #pragma omp parallel
  threads = omp_get_num_threads();

  std::vector<int> index(sc::ranking * threads);
  for (int i=0; i<sc::ranking * threads; i++) index[i] = i;
  std::sort(index.begin(), index.end(),
    [&](const int& a, const int& b) { return (minD[a] < minD[b]); });
  for (int i=0; i<sc::ranking; i++) {
    int ii = index[i];
    sc::pairs[i].i = permutation[minI[ii]];
    sc::pairs[i].j = permutation[minJ[ii]];
    if(sc::pairs[i].i > sc::pairs[i].j) std::swap(sc::pairs[i].i, sc::pairs[i].j);
    sc::pairs[i].dist2 = minD[ii];
  }

  delete[] key;
  delete[] bucket;
  for (int i=0; i<sc::maxThreads; i++)
    delete[] bucketPerThread[i];
  delete[] bucketPerThread;
  delete[] offset;
  delete[] permutation;
  delete[] X2;
  delete[] Y2;

  sc::output();
  sc::finalize();
  return 0;
}
```
