Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 05-02-2021

Last Name: .....
First Name: .....
Student ID: .....
Signature: .....

Exam duration: 2 hours (Online version)

Students can use a pen or a pencil for answering questions.

Students are NOT permitted to use books, course notes, calculators, mobile phones, and similar connected devices.

Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.

Writing on the cheat sheet is NOT allowed.

Exercise 1: _____ Exercise 2: _____ Exercise 3: _____

Exercise 1 (14 points)

You work for NetFlick a video on demand company and you have to develop the recommender system that suggests to users the movies to watch. For all of the required functionalities, you have to optimize the **average case complexity**. The class diagram of the reference implementation is reported in Figure 1.

Movies are characterized by a title and by their cast and are stored in a vector. Users are also stored in a vector and are characterized by their name and by the set of indices of the movies they watched. The core of your system are **two sparse matrices**, `ratings` that stores the rating (in the range [1,5]) given by user i to movie j and `similarity_matrix` that stores the similarity s_{ij} of users i and j according to the formula:

$$s_{ij} = \frac{1}{|S|} \sum_{m \in S} (max\_rate - |r_i(m) - r_j(m)|) \quad (1)$$

where $S = Movies_i \cap Movies_j$ is the intersection of the sets of movies watched by both user i and j and r_i denotes the rating given by user i to movie m.

You have to:

1. provide the declaration of the matrices `ratings` and `similarity_matrix`.

Moreover, provide the implementation of the following methods:

2. `void add_user (const string & name);`
   which adds a user to the system.
3. `void add_movie(const std::string & title, const std::string & cast);`
   which adds a movie to the system.
4. `void add_rating(size_t user_index, size_t movie_index, unsigned short rate);`
   which **updates the system data structures** according to the new `rate` by the user characterized by `user_index` for the movie at index `movie_index`.
5. `void compute_similarity_matrix();`
   which updates the `similarity_matrix` according to the current ratings.
6. `void movies_similar(size_t user_index, float min_rate) const;`
   which, for the specified user, prints the titles of the movies watched by all users j such that:
   `similarity_matrix[user_index][j] >= min_rate`.

Figure 1: NetFlick recommender system Class Diagram

Finally, provide:

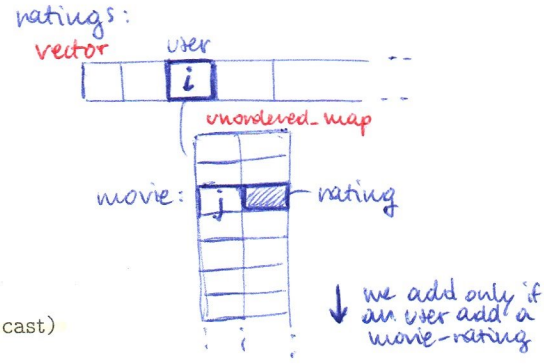
7. the **average complexity** of all the methods you have implemented.

Note that you can rely on the private method:
`std::set<size_t> NetFlick::intersect_movies(size_t user_index1 , size_t user_index2) const;`
which returns the indices of the movies watched by the users of indices `user_index1` and `user_index2`. Consider that its complexity is $O(M'log(M'))$, where M' is the average number of movies watched by individual users.

Solution 1

In order to optimize the average case complexity and to implement also sparse matrices, both the rating and similarity matrices are stored as vectors of unordered maps. The declaration of the data structures is reported below:

1.
```
class NetFlick {
    ...
    vector<unordered_map<size_t, unsigned short>> ratings;
    vector<unordered_map<size_t, float>> similarity_matrix;
    ...
}
```

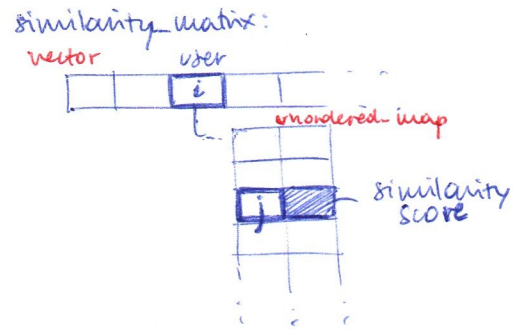


The code implementing the required methods is reported below.

```
/* From NetFlick.cpp */
```

3.
```
void NetFlick::add_movie(const std::string & title, const std::string & cast)
{
    movies.emplace_back(title,cast);
}
```
or: `movies.push_back(Movie(title,cast));`

5.
```
void NetFlick::compute_similarity_matrix()
{
    for (size_t i = 0; i < users.size() - 1; ++i)
    {
        for(size_t j = i + 1; j < users.size(); ++j)
```



```
        {
            std::set<size_t> shared_movies_indexes = intersect_movies(i, j);
            if (shared_movies_indexes.size() > 0)
            {
                float similarity =0;
                for (size_t movies_index : shared_movies_indexes)
                    similarity += (max_rate - std::abs(ratings[i][movies_index] - ratings[j][movies_index]));
                similarity_matrix[i][j] =similarity / shared_movies_indexes.size();
                similarity_matrix[j][i] = similarity_matrix[i][j];
            }
        }
    }
}
```

2.
```
void NetFlick::add_user(const std::string &name)
{
    users.emplace_back(name);
    ratings.push_back({});
    similarity_matrix.push_back({});
}
```
users.push_back(User(name));
we just added an user, we don't have movies yet

4.
```
void NetFlick::add_rating(size_t user_index, size_t movie_index, unsigned short rate)
{
    ratings[user_index][movie_index] = rate;
    users[user_index].add_movie_index(movie_index);
}
```
similarity?

6.
```
void NetFlick::movies_similar(size_t user_index, float min_rate) const
{
    std::vector<size_t> above_threshold_users;

    // select users satisfying the threshold
    for (auto it = similarity_matrix[user_index].cbegin(); it != similarity_matrix[user_index].cend(); ++it)
        if (it->second >= min_rate)
            above_threshold_users.push_back(it->first);

    std::set<size_t> selected_movies_indexes;
    // pick all movies indexes
    for (size_t i = 0; i < above_threshold_users.size(); ++i)
    {
        const auto & add_movies_IDs = users[above_threshold_users[i]].get_movies_indexes();
        selected_movies_indexes.insert(add_movies_IDs.cbegin(),add_movies_IDs.cend());
    }

    // print movies satisfying criteria
    for (auto movie_index : selected_movies_indexes)
        movies[movie_index].print();

}

void NetFlick::print_users() const
{
    cout << "### Users ###" << endl;
    for (auto it = users.cbegin(); it!= users.cend();++it)
        it->print();
}

void NetFlick::print_movies() const
```

```
{
   cout << "### Movies ###" << endl;
   for (auto it = movies.cbegin(); it!= movies.cend();++it)
      it->print();
}

void NetFlick::print_ratings() const
{
   cout << "### Ratings ###" << endl;
   for (size_t i=0; i< ratings.size(); ++i)
   {
      cout << "User: " << i <<": ";
      for (auto it = ratings[i].cbegin(); it != ratings[i].cend(); ++it)
         cout << it->first << ": " << it->second << ", ";
      cout <<endl;
   }
}

void NetFlick::print_similarity_matrix() const
{
   cout << "### Similarity Matrix ###" << endl;
   for (size_t i = 0; i < similarity_matrix.size(); ++i)
   {
      cout << "User: " << i <<": ";
      for (auto it = similarity_matrix[i].cbegin(); it != similarity_matrix[i].cend(); ++it)
         cout << it->first << ": " << it->second << ", ";
      cout <<endl;
   }
}

void NetFlick::print() const
{
   print_users();
   print_movies();
   print_ratings();
   print_similarity_matrix();
}

std::set<size_t> NetFlick::intersect_movies(size_t user_index1, size_t user_index2) const
{
   std::vector<size_t> mv1 =users[user_index1].get_movies_indexes();
   std::vector<size_t> mv2 =users[user_index2].get_movies_indexes();


   std::vector<size_t> v(mv1.size() + mv2.size()); //result vector
   std::vector<size_t>::iterator it;

   std::sort (mv1.begin(),mv1.end());
   std::sort (mv2.begin(),mv2.end());

   it=std::set_intersection (mv1.cbegin(), mv1.cend(), mv2.cbegin(),mv2.cend(), v.begin());
   v.resize(it-v.begin());

   return std::set<size_t>(v.cbegin(), v.cend());
}
```

The inserts in the vectors are straightforward. The point of attention is on the need to keep the data structures coherent. For this reason add_user prepares also the matrices by pushing back empty unordered maps.

More interesting methods are compute_similariy_matrix and movies_similar. The former exploits the fact

that the similarity matrix is symmetric and performs only a scan of half of the possible couples of users (i,j). The set of movies of interest is obtained by relying on the `intersect_movies` method.

The `movies_similar` method first of all selects the set of users j whose similarity is above the given threshold. Then, it identifies the indices of the movies that need to be printed and finally prints the movies data. The use of sets allow to easily remove movies replica. The average complexity of `add_` methods is $O(1)$. The average complexity of `compute_similariy_matrix` is $O(N^2(M + M' \log(M')) = O(N^2 M' \log(M'))$, where N is the number of users, M is the average number of movies watched by two users while M' is the average number of movies watched by individual users (Note that, $O(M) = O(M')$). The term $O(M' \log(M'))$ is due by the `intersect_movies` method which sorts the two initial vectors in a way the intersection can then be computed with $O(M')$ complexity. The `movies_similar` has $O(K+N'M' \log(M')+N'M')$ where K is the number of elements in the row `similarity_matrix[user_index]`, N' is the average number of users with similarity above the threshold and M' is the average number of movies watched by any user. In particular, we obtain the term $O(K)$ from the first **for** loop (remember that the average complexity of `push_back` is $O(1)$). Then, we loop over the N' users in `above_threshold_users` and, at each iteration, we insert the M' movies in `selected_movies_indexes`, obtaining an overall complexity of $O(N'M' \log(M'))$. Finally, we loop over all movies in `selected_movies_indexes`, which are $N'M'$ on average, to print them.

Exercise 2 (11 points)

You have to implement a **parallel function** with the following prototype:

```
la::dense_matrix evaluate_by_column(const la::dense_matrix& A, const function_t& f);
```

It receive as first parameter a matrix $A \in \mathbb{R}^{m \times n}$ ()you can **assume that the number of columns n is multiple of the number of available processes**). The second parameter is an object of type `function_t`, which represents a function $f : \mathbb{R}^m \to \mathbb{R}$ and is defined as:

```
typedef std::function<double(const la::dense_matrix&)> function_t;
```

The parameter of the function f is a **column vector** $c \in \mathbb{R}^{m \times 1}$ that **must be represented as a `dense_matrix`**. Both the matrix A and the function f are **assumed to be known by all processes**.

The function `evaluate_by_column` returns a **row vector** $r \in \mathbb{R}^{1 \times n}$, **represented as a `dense_matrix`**, which should be computed as follows: if we denote by A_j the j-th column of A and by r_j each component of the solution vector r,

$$r_j = f(A_j) \quad \forall j \in \{1, ..., n\}. \tag{2}$$

As an example, suppose that the matrix A is used to store elements coming from a dataframe. The rows of the dataframe represent different students, while the columns store the grade obtained by these students in different exercises and the final grade of the exam (the orange part in the table below).

Student	Exercise1	Exercise2	Exercise3	FinalGrade
Alice	11.5	10	8	29.5
Bob	10	9.5	6	25.5
Carol	6	5	8	19

If you have defined the function f to compute the average, you will obtain as output the following row vector r:

$$[9.17,\ 8.17,\ 7.3,\ 24.7],$$

which are the averages for Exercise1, Exercise2, Exercise3 and FinalGrade, respectively.

The declaration of the `dense_matrix` class is reported below:

```
#ifndef DENSE_MATRIX_HH
#define DENSE_MATRIX_HH

#include <istream>
#include <vector>

namespace la // Linear Algebra
{
 class dense_matrix final
```

```
{
  typedef std::vector<double> container_type;

public:
  typedef container_type::value_type value_type;
  typedef container_type::size_type size_type;
  typedef container_type::pointer pointer;
  typedef container_type::const_pointer const_pointer;
  typedef container_type::reference reference;
  typedef container_type::const_reference const_reference;

private:
  size_type m_rows, m_columns;
  container_type m_data;

  size_type
  sub2ind (size_type i, size_type j) const;

public:
  dense_matrix (void) = default;

  dense_matrix (size_type rows, size_type columns,
                const_reference value = 0.0);

  explicit dense_matrix (std::istream &);

  void
  read (std::istream &);

  void
  swap (dense_matrix &);

  reference
  operator () (size_type i, size_type j);
  const_reference
  operator () (size_type i, size_type j) const;

  size_type
  rows (void) const;
  size_type
  columns (void) const;

  dense_matrix
  transposed (void) const;

  pointer
  data (void);
  const_pointer
  data (void) const;

  void
  print (std::ostream& os) const;
};

dense_matrix
operator * (dense_matrix const &, dense_matrix const &);

void
```

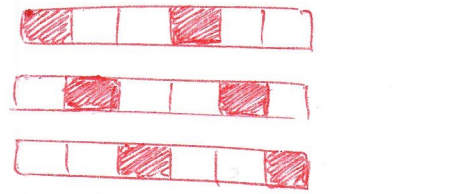
```
  swap (dense_matrix &, dense_matrix &);
}

#endif // DENSE_MATRIX_HH
```

Solution 2

The parallel function evaluate_by_column can be implemented according to two different strategies: using *cyclic partitioning*, or using *block partitioning*. In both cases, **we do not need to perform any communication at the beginning, since the matrix is already known to all processes**. The two alternative implementations are reported below (remember that we have to add **using** la::dense_matrix or to report explicitly the namespace in the code):

*** cyclic partitioning

```
dense_matrix evaluate_by_column(const dense_matrix& A, const function_t& f)
{
  int rank(0), size(0);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  unsigned nrows = A.rows();
  unsigned ncols = A.columns();

  // result is a row vector
  dense_matrix result(1,ncols);
  for (unsigned j = rank; j < ncols; j+=size)
  {
    // extract column
    dense_matrix column(nrows,1);
    for (unsigned i = 0; i < nrows; ++i)
      column(i,0) = A(i,j);

    // compute result
    result(0,j) = f(column);
  }

  // loop over all elements
  for (unsigned j = 0; j < ncols; ++j)
  {
    // broadcast each element from the proper root
    int root = j % size;
    MPI_Bcast(&(result(0,j)), 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
  }

  return result;
}
```



*** block partitioning

```
dense_matrix evaluate_by_column(const dense_matrix& A, const function_t& f)
{
  int rank(0), size(0);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  unsigned nrows = A.rows();
  unsigned ncols = A.columns();
```

```
unsigned n_local_cols = ncols / size;

// result is a row vector
dense_matrix local_result(1,n_local_cols);
for (unsigned j = 0; j < n_local_cols; ++j)
{
  // extract column
  dense_matrix column(nrows,1);
  for (unsigned i = 0; i < nrows; ++i)
    column(i,0) = A(i, j + rank * n_local_cols);

  // compute result
  local_result(0,j) = f(column);
}

// collect result
dense_matrix result(1,ncols);
MPI_Allgather(local_result.data(), n_local_cols, MPI_DOUBLE, result.data(),
          n_local_cols, MPI_DOUBLE, MPI_COMM_WORLD);

return result;
}
```

The two versions of the function have the same structure: in both cases, at each iteration of the external loop we have to extract the correct column from the matrix A in order to give it as input to function f. Since **we do not have methods that allow us to extract a specific column in a matrix** (something similar to the instruction $A[:, j]$ that we could rely on in Matlab, for instance), we have to **copy the column in a temporary** `dense_matrix` with the correct number of rows.

The first difference among the two strategies is that in the ***cyclic partitioning*** case we can use directly a row vector `result`, initialized with the number of columns of the matrix A. Indeed, each process will loop over the elements that it has to compute and fill the corresponding values of `result`. In the ***block partitioning*** case, instead, we define a local row vector `local_result`, having a number of columns equal to `A.columns()/size`. In this case, we have to be careful when accessing the elements of A to build the column, since we have to compute the correct index as `j + rank * n_local_cols`.

The second difference is in how we proceed to collect the final result (**on all processes, since we are returning from a function!**): in the ***block partitioning*** case, we rely on `MPI_Allgather`, having defined a global row vector `result` with the correct number of columns. In the ***cyclic partitioning*** case, instead, given that each process worked directly on the global vector `result`, we have to loop over it and, at each iteration, perform an `MPI_Bcast`, carefully determining which process is the root of the communication.

Exercise 3 (8 points)

The provided code implements two classes: `Book` is used to represent books, that are uniquely identified by an `ID` and are characterized by a `Title` and a `rating`, computed according to the number of reviews (`num_reviews`). The class provides a method to add a new rating to the book (updating the number of reviews accordingly).

A second class called `Library` represents a collection of books which are owned by a reader. Library owners can rate their books, and also store in their library notes about the books they own.

The complete implementation of both classes is reported below.

After carefully reading the code, you have to answer the following questions in the provided web form. Note that each answer requires you to type a single number!

1. How many books are stored in `l1` after the execution of `l1.print` at line 24?
2. How many books are stored in `l2` after the execution of `l2.print` at line 28?
3. How many unique owner names have been printed overall?
4. How many notes are printed during the execution of `l3.print` at line 33?
5. How many reviews did "The Great Gatsby" get overall?

6. Which is the average rating for "The Hobbit" when line 40 is reached?

Provided source code:

- **Book.h**

```
#include <string>
#include <iostream>

class Book {
   std::string Title;
   unsigned ID;
   unsigned num_reviews = 0;
   double rating = 0;

public:
   Book(const std::string& t, unsigned &id) : Title(t), ID(id) {++id;}

   void print() const;

   void addRating (unsigned r);

   std::string getTitle() const;

   unsigned getID() const;
};
```

- **Book.cpp**

```
#include "Book.h"

void Book::print() const
{
   std::cout << std::endl;
   std::cout << "# Title: " << Title << std::endl;

   std::cout << "Rating ";
   for (unsigned i = 0; i < rating; ++i)
      std::cout << "*";

   std::cout << " based on " << num_reviews << " reviews" << std::endl;
}

void Book::addRating (unsigned r)
{
   rating = (rating * num_reviews + r) / (num_reviews + 1);
   ++num_reviews;
}

std::string Book::getTitle() const
{
   return Title;
}

unsigned Book::getID() const {
   return ID;
}
```

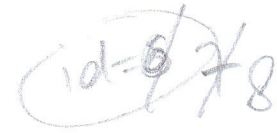
- **Library.h**

```
#include <unordered_map>
#include <vector>
#include <memory>
#include "Book.h"

typedef std::vector<Book> shelfType;

class Library {
   std::string owner;
   std::shared_ptr<shelfType> shelf;
   std::unordered_map<unsigned, std::string> notes;

   bool search (const Book& b) const;

public:
   explicit Library (const std::string& o);

   Library (const Library& lib);

   void addBook (const Book& b);

   void rateBook (Book& b, unsigned rating);

   void rateBook (Book& b, unsigned rating, const std::string& note);

   void print() const;

   Library& operator= (const Library&);
};
```

- **Library.cpp**

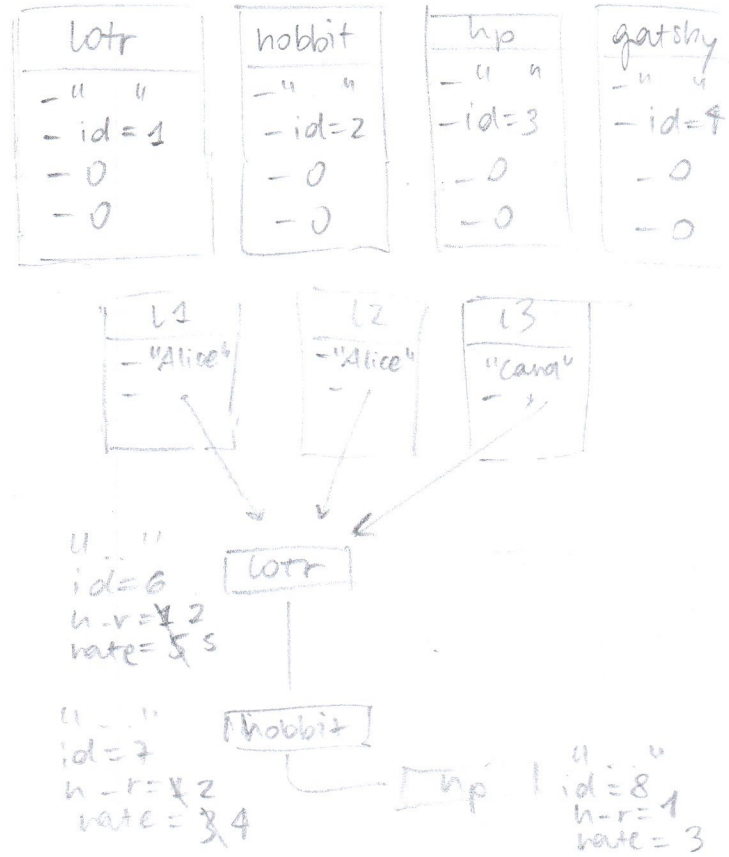
```
#include "Library.h"

bool Library::search (const Book& b) const
{
   for (const Book& owned : *shelf)
   {
      if (b.getID() == owned.getID())
         return true;
   }
   return false;
}

Library::Library (const std::string& o) : owner(o), shelf(std::make_shared<shelfType>()) {}

Library::Library (const Library& lib)
{
   owner = lib.owner;
   shelf = lib.shelf;
}

void Library::addBook (const Book& b)
{
   shelf->push_back(b);
}

void Library::rateBook (Book& b, unsigned rating)
{
```

```
    if (search(b))
        b.addRating(rating);
}

void Library::rateBook (Book& b, unsigned rating, const std::string& note)
{
    if (search(b))
    {
        rateBook(b, rating);
        notes.insert({b.getID(), note});
    }
}

void Library::print() const
{
    std::cout << std::endl;
    std::cout << "# " << owner << "'s Library:" << std::endl;
    for (const Book& p : *shelf)
    {
        std::cout << p.getTitle();

        if(notes.find(p.getID()) != notes.end())
            std::cout << " -- " << notes.at(p.getID());

        std::cout << std::endl;
    }
}

Library& Library::operator= (const Library& l)
{
    shelf = l.shelf;
    notes.clear();
    return *this;
}
```



- **main.cpp**

```
#include "Library.h"

int main()
{
    unsigned id = 0;
    Book lotr("The Lord of the Rings", id);
    Book hobbit("The Hobbit", id);
    Book hp("Harry Potter", id);
    Book gatsby("The Great Gatsby", id);

    Library l1("Alice");
    Library l2(l1);
    Library l3("Carol");

    l3.addBook(gatsby);
    l3.rateBook(gatsby, 4, "A Modern classic");

    l3 = l1;
    l1.addBook(lotr);
    l1.rateBook(lotr, 5, "A milestone in literature");
    l1.addBook(hobbit);
    l1.rateBook(hobbit, 3);
```



```
23      l1.rateBook(gatsby, 4);
24      l1.print();
25
26      l2.addBook(hp);
27      l2.rateBook(hp, 3, "Great for kids");
28      l2.print();
29
30      l3.rateBook(gatsby, 4);
31      l3.rateBook(hobbit, 5, "There and Back Again");
32      l3.rateBook(lotr, 5);
33      l3.print();
34
35      lotr.print();
36      hobbit.print();
37      hp.print();
38      gatsby.print();
39
40      return 0;
41  }
```

Solution 3

1. `Library` l1 contains only the two books added at lines 19 and 21, therefore the answer is 2.
2. `Library` l2 was created through a copy constructor from l1, therefore it contains all books inserted in l1 or l2 up to this point. The answer is 3.
3. Since l2 is created through a copy constructor from l1, they will share the same owner. Instead, l3 is created with a different owner which is not changed by the assignment operator in line 18. The answer is 2.
4. Given that notes stored in a `Library` are deleted whenever the assignment operator is applied to it, the only notes stored – and printed – by line 33 are those inserted in l3 after line 18. The answer is 1.
5. Despite the fact that the `main` contains multiple attempts at rating the book, it is only possible to successfully rate a book if it is owned by the relative library. The only valid rating is the one performed at line 16, and the answer is 1.
6. Both ratings for the book are successful since both libraries own them when they are called. The average rating is therefore 4.

~~int main(int argc, char *argv[]){~~
~~MPI_Init(&argc, &argv);~~

~~MPI_Bcast(&n, 1, MPI_DOUBLE, 0, 0);~~

```
        }
    }
    result = result / quantity;
  }
  similarity_matrix[i][j] = result;
  similarity_matrix[j][i] = result;
 }
}
double result;
```

```
for (size_t i=0; i < users.size()-1 ; ++i) {
  for (size_t j=i+1, j < users.size(); ++j) {
    // How many movies they share?
    std::set<size_t> intersect_movies(i,j);
    unlike_movies =
    int quantity = unlike_movies.size();
    if (quantity >1) {
      for (const size_t movie_index : unlike_movies){
        result += std::max(ratings[i][movie_index], ratings[j][
        movie_index]) - std::abs(ratings... - ratings...);
```

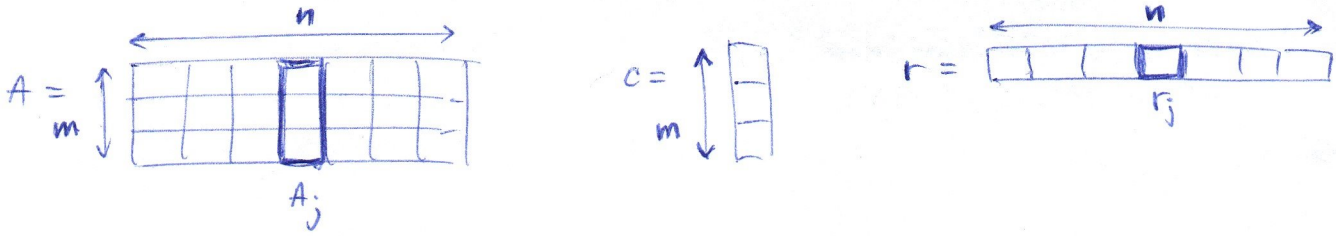
- go to similarity_matrix [user_index]
- vedi nella sua lista chi ha ≥ min_rate
- prendi gli indici di tutti i film di chi ha ≥ min_rate

- alla fine fai unique
- print

```
vector<size_t> movie_index;
for(size_t j=0; j < similarity_matrix[user_index].size() ; ++j){
    if (similarity_matrix[user_index][j] >= min_rate){
        for(size_t k=0; k< users[j].movies_indexes.size() ; ++k){
            movie_index.push_back(users[j].movies_indexes[k]);
        }
    }
}
std::set<size_t> unique_movie_index (movie_index.cbegin(), movie_index.cend());
for(auto movie : unique_movie_index){
    cout << movies[movie].title << endl;
}
```

~~MPI_Recv(&n, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);~~

~~MPI_Send(&n, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);~~



$r_j = f(A_j)$

```
la::dense_matrix  evaluate_by_column ( const la::dense_matrix& A, const function_t& f) {
    int rank, size;
    MPI_Comm_size(..) ;
    MPI_Comm_rank(.) ;
    size_type n = A.m_columns;
    size_type m = A.m_rows;

    la::dense_matrix r(1,n);
    for(size_t j=rank; j < n; j+=size) {
        la::dense_matrix temp(m,1);
        for(size_t i=0; i<m ; ++i) {
            temp(i,1) = A(i,j);
        }
        r(1,j) = f(temp);
    }
    for( size_t k=0; k<n; ++k) {
        int sender = k % size;
        MPI_Bcast(&(r(0,j)),1, MPI_DOUBLE, sender, MPI_COMM_WORLD);
    }
    return r;
```