Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 20-07-2020

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 (15+2 points)

You have to implement GoodreadX a free platform to share reviews and opinions on books. Books are uniquely identified by their title and have an author (for the sake of simplicity suppose books have a single author and there are no books with the same title). A review is characterized by the title of the book, the text and the rating (an integer between 1 and 5). Below, an example of the data available for a book:

Title: Harry Potter and the Philosopher's stone
Author: J. K. Rowling
Publisher: Bloomsbury
pages: 223
1 stars: 121
2 stars: 4342
3 stars: 80012
4 stars: 199878
5 stars: 109010
Total reviews: 393363
Average: 4.05

The system also **allows to browse each review the book has received.** Your implementation should optimize the worst case complexity and favor as operation a book search. You should also **optimize the computation of the average** of the review scores.
In particular, you have to:

1. provide the declaration of the class `Review`.
2. provide the declaration of the class `GoodReads`.

within the class `GoodReads`:

3. implement the method:
   `void add_book(const string & title, unsigned pagesN, const string &publisher, const string &author);`
   which adds a book and its relevant information to the system. Provide also the method worst case complexity.
4. implement the method:
   `void add_review(const string &bookTitle, const string &text, unsigned int rating);`
   which adds a review to the system. Provide also the method worst case complexity.

5. implement the methods:
   `float get_avg_rating() ;`
   `float get_avg_rating(const string & title);`

   which provide the average ratings for all books and for the book with the specified title. Provide also the methods worst case complexity.

6. implement the method:
   `void search_reviews(const vector<string> & keywords) ;`

   which, prints all the review including all the specified `keywords` (complexity is not required).

7. (**2 extra points**) implement the method:
   `void print_book(const string & title);`

   which prints all the data (including its review) of the specified book (complexity is not required).

Note that, `const` specifers are omitted in the previous methods (**you have to provide in your implementation**). **You can introduce additional classes if needed.**

Moreover, you can rely on the function:

```
vector<string> split(const string &s, char d);
```

which given a string `s` and a delimeter `d` provides as return vector the words in `s` separated by `d` (for the sake of simpicity, assume that **words in reviews are separated by spaces**).

Finally, as a last hint, given two sets `a` and `b` you can test if $a \supseteq b$ by:

```
std::includes(a.cbegin(), a.cend(), b.cbegin(), b.cend())
```

where `std::includes()` is defined in `algorithm` and returns a `bool`.

Remarks
Duplicated code should be avoided: if a functionality needed by a function is already implemented as a method, it should be reused.
Provide the implementation of other required methods or functions, if any.
To cope with error conditions (e.g., a book is not in the system), simply write error messages in `cerr`.

Solution 1

The header file of the class `Review` is the following:

```
#ifndef GOODREADS_REVIEW_H
#define GOODREADS_REVIEW_H

#include <string>
#include <set>
#include <vector>
#include <sstream>


using std::set;
using std::vector;
using std::string;

class Review {
    string book_title;
    string text;
    unsigned rating;
    set<string> words;
```

```
public:
    Review(const string &bookTitle, const string &text, unsigned int rating);
    string to_string() const;

    string get_text() const;
    set<string> get_words() const;

};

vector<string> split(const string &s, char d);


#endif //GOODREADS_REVIEW_H
```

The class includes all the relevant information and the set of words contained in the text of the review. The set is used by the keywords search functionality of the class GoodReads and is filled by the constructor. In this way, the set is created once for all and multiple searches can reuse it.

The implementation of the class Review is reported below:

```
#include "Review.h"

Review::Review(const string &bookTitle, const string &t, unsigned int r) :
            book_title(bookTitle), text(t), rating(r)  {
    const vector<string> text_strings(split(t, ' '));
    words =  set<string>(text_strings.cbegin(),text_strings.cend());
}

string  Review::to_string() const {

    return book_title + " " + text + " " +  std::to_string(rating);
}

string  Review::get_text() const {
    return text;
}

set<string> Review::get_words() const {
    return words;
}

vector<string> split(const string &s, char d){
    string word;
    std::vector<string> v;
    std::istringstream text_read (s);
    while (std::getline (text_read, word, d))
    v.push_back (word);
    return v;
}
```

The class GoodReads needs to store the reviews and the data of the books. Since we need to optimize book searches and take into account the worst case complexity, a map is the best container to use, providing $O(log(n))$ complexity (n is the number of books stored in the system). We report below the declaration and implementation of the class BookData which stores all the relevant data of a book. The book title is not included in this class to avoid data replica and is stored and used as key of the map within the class GoodReads.

```
#ifndef GOODREADS_BOOKDATA_H
#define GOODREADS_BOOKDATA_H

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

```
#include <list>

using std::cout;
using std::endl;

using std::string;
using std::vector;
using std::list;


class BookData {
    vector<unsigned> ratings_distr;
    unsigned pages_number;
    string publisher;
    unsigned review_count;
    string author;
    float avg_rating;
    list<unsigned> review_indexes;
public:
    BookData(unsigned int pagesNumber, const string &publisher, const string &author);
    float get_avg_rating() const;
    void add_review(unsigned index, unsigned stars);
    string to_string() const;
    list<unsigned> get_review_indexes()const ;
private:
    float compute_rating();
};


#endif //GOODREADS_BOOKDATA_H
```

```
#include "BookData.h"

BookData::BookData(unsigned pagesNumber, const string &pub, const string &author) :
pages_number(pagesNumber), publisher(pub), author(author) {
    ratings_distr= vector<unsigned>(5,0);
    review_count =0;
    avg_rating =0.0;

}

float BookData::get_avg_rating() const {
    return avg_rating;
}

void BookData::add_review(unsigned int index, unsigned int stars) {
    review_indexes.push_back(index);
    ratings_distr[stars-1]++;
    review_count++;
    avg_rating = compute_rating();
}

float BookData::compute_rating() {
    float average = 0;

    for (size_t i=0; i< ratings_distr.size(); ++i)
        average += (i+1)* ratings_distr[i];

    return average/review_count;
```

```
}

string BookData::to_string() const {

    return author + " " + publisher + " " + std::to_string(pages_number) + " " +
           std::to_string(review_count) + " " + std::to_string(avg_rating);

}

list<unsigned> BookData::get_review_indexes() const {
    return review_indexes;
}
```

BookData stores stars scores by using a single vector, i.e., rating_distr. Moreover, to optimize the average score computation, the average is stored as a class attribute and recomputed anytime a new review is added.

Reviews are stored in a vector in the class GoodReads (this favours memory access for keywords search). As can be noticed by inspecting the code above, the system allows browsing a book reviews by storing in the list BookData::review_indexes the indexes of the reviews stored in the reviews vector.

The declaration and implementation of the class GoodReads are reported below:

```
#ifndef GOODREADS_GOODREADS_H
#define GOODREADS_GOODREADS_H

#include <map>
#include <string>
#include <vector>
#include <iostream>

#include "BookData.h"
#include "Review.h"

using std::map;
using std::string;
using std::vector;

using std::cout;
using std::cerr;
using std::endl;


class GoodReads {
    map<string, BookData> books; // <title, BookData>
    vector<Review> reviews;
public:
    void add_book(const string & title, unsigned pagesNumber, const string &publisher, const string &
     author);
    void add_review(const string &bookTitle, const string &text, unsigned int rating);
    float get_avg_rating() const;
    float get_avg_rating(const string & title) const;
    void search_reviews(const vector<string> & keywords) const;
    void print_book(const string & title) const;
};


#endif //GOODREADS_GOODREADS_H

#include "GoodReads.h"
```

```
void GoodReads::add_book(const string &title, unsigned int pages_number, const string &publisher, const
    string &author) {
   BookData bd(pages_number, publisher, author);
   if (books.find(title) == books.cend())// the book is not already in the collection
      books.insert(std::make_pair(title, bd));
   else
      cerr << "The book is already in the books collection" <<endl;
}

void GoodReads::add_review(const string &book_title, const string &text, unsigned int rating) {
   const auto it = books.find(book_title);
   if (it == books.cend())// the book is not in the collection
      cerr << "The book is not in the books collection" <<endl;
   else{
      Review r(book_title,text,rating);
      reviews.push_back(r);
      it->second.add_review(reviews.size()-1,rating);
   }
}

float GoodReads::get_avg_rating() const {
   float avg= 0;
   size_t count = 0;
   for (auto it = books.cbegin(); it != books.cend(); ++it){
      avg += it->second.get_avg_rating();
      count++;
   }

   return avg/count;
}

float GoodReads::get_avg_rating(const string & title) const {
   const auto it = books.find(title);
   if (it == books.cend()){// the book is not in the collection
      cerr << "The book is not in the books collection" <<endl;
      return 0;
   }
   else
      return it->second.get_avg_rating();
}

void GoodReads::search_reviews(const vector<string> & keywords) const {
   const set<string> keywords_set(keywords.cbegin(),keywords.cend());
   for (auto it = reviews.cbegin(); it != reviews.cend(); ++it){
      const set<string> & words = it->get_words();
      if(std::includes(words.cbegin(), words.cend(), keywords_set.cbegin(), keywords_set.cend()) )
         cout << it->to_string()<<endl;
   }

}

void GoodReads::print_book(const string &title) const {
   const auto it = books.find(title);
      if (it == books.cend())// the book is not in the collection
         cerr << "The book is not in the books collection" <<endl;
      else{
      // print book data
      cout << it->first<<endl;
```

```
        cout << it->second.to_string()<<endl;
        auto rev_list = it->second.get_review_indexes();
        auto rev_it = rev_list.cbegin();
        while ( rev_it != rev_list.cend()){
            cout << reviews[*rev_it].to_string() << endl;
            rev_it++;
        }
    }
}
```

For what concerns methods complexity:

1. `void add_book()` has $O(log(n))$ complexity (n is the number of books stored in the system), due to the `find` and `insert` in the map data structure.
2. `void add_review()` has $O(log(n)+m)$ complexity where n is the number of books stored in the system and m is the number of reviews. The log term is due by the `find` in the map while the linear term by the `push_back` in the review vector. Note that adding a review to the book has a constant complexity, since recomputing the average is indipendent on the size of the system (5 stars are always considered despite the number of books and reviews) and also the `push_back` in a list is constant.
3. `float get_avg_rating()` is linear in the number of books while `float get_avg_rating(const string & title)` has $O(log(n)$ complexity due again by the `find` operation.

Exercise 2 (8 points)

Given the class `Author` (which includes two attributes, i.e., `name` and `nation` and provides the corresponding getters and setters) and the following definition:

```
using authors = vector<Author>;
```

you have to develop **the parallel function**:

```
size_t author_nation_count(const string & nation, const authors & auts);
```

which returns the number of authors from the given `nation`.

Assume that the vector of authors is available in all available processes while `nation` is available only in rank 0. Moreover, for the sake of simplicity, you can assume that the size of the authors vector is a multiple of the number of available processes.

Solution 2

The implementation of the function `author_nation_count` is reported below:

```
size_t author_nation_count(const string & nation, const authors & auts){
 int rank, size;

 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 MPI_Comm_size (MPI_COMM_WORLD, &size);

 std::string nation_msg = nation;
 unsigned length = nation.size();

 if (rank == 0)
     {
     MPI_Bcast (&length,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
     MPI_Bcast (&nation_msg[0],length,MPI_CHAR,0,MPI_COMM_WORLD);
     }
 else
```

```
    {

    MPI_Bcast (&length,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
    nation_msg.resize(length);
    MPI_Bcast (&nation_msg[0],length,MPI_CHAR,0,MPI_COMM_WORLD);
    }
 unsigned count =0;

 for (size_t i=rank; i< auts.size(); i+=size)
    if (auts[i].get_nation() == nation_msg)
       count++;

 MPI_Allreduce (MPI_IN_PLACE, &count, 1, MPI_UNSIGNED,
                    MPI_SUM, MPI_COMM_WORLD);

 return count;
}
```

Rank 0 initially sends the nation string to all processes. This requires to send the nation lenght first, in a way processes can prepare the receiving buffers. Both the string and its lenght are shared by a MPI_Bcast operation. Next, since the authors vector is already available in all ranks, the local count is performed locally through a cyclic partitioning scheme. Finally, the total count is computed through a MPI_Allreduce operation and returned by all processess.

Exercise 3 (8 points)

The goal of the source code provided is to implement a simple calendar. Class Calendar stores Events that are characterized by a name (implemented as a std::string) and a date (implemented as time_t). Stored events can be visualized through method print of the class Calendar. The header and implementation files of Events are provided along with the header file of Calendar while Calendar.cpp is not complete since the implementation of some methods is missing.

After carefully reading the code, you have to:

1. **Complete the implementation of Calendar** in order to implement a ***like-a-pointer* behavior.**
2. **List and motivate** the program **output** considering the main file provided.

Provided source code:

- **Event.h**

```
class Event {
private:
    time_t date;
    string name;
public:
    Event(time_t d, const string& n): date(d), name(n){};
    time_t getTime() const;
    string getName() const;
};
```

- **Event.cpp**

```
time_t Event::getTime() const {
    return date;
};
string Event::getName() const {
    return name;
};
```

- **Calendar.h**

```
class Calendar {
private:
    vector<Event> *events;
    size_t *count;
public:
    Calendar(): events(new vector<Event>()), count(new size_t(1)) {};
    Calendar(const Calendar&);
    Calendar& operator=(const Calendar&);
    ~Calendar();
    void addEvent(const Event&);
    void print() const;
};
```

- **Calendar.cpp**

```
// MISSING IMPLEMENTATION

void Calendar::addEvent(const Event &event){
    events->push_back(event);
}

void Calendar::print() const {
    for (const auto &e : *events){
        cout << e.getName() << endl;
    }
}
```

- **main.cpp**

```
int main() {
    Calendar c0;
    Event e0(time(0), "Important Meeting");
    c0.addEvent(e0);

    Calendar c1;
    Event e1(time(0), "Andrew's Birthday");
    c1.addEvent(e1);
    c1 = c0;

    Calendar c2(c0);
    Event e2(time(0), "Trip to Cairo");
    c2.addEvent(e2);

    c2 = Calendar();
    Event e3(time(0), "Visit to Museum");
    c2.addEvent(e3);

    cout << "c0:" << endl;
    c0.print();
    cout << "c1:" << endl;
    c1.print();
    cout << "c2:" << endl;
    c2.print();

    return 0;
}
```

Solution 3

1. In order to implement a *like-a-pointer* behavior, we have to exploit instance variable `count` to enumerate the instances pointing to the same data. In order to share the data, `count` and `events` can be simply copied using assignment because they are pointers to dynamic memory. `count` must be incremented and decremented when an instance is copied or de-allocated respectively. When count reaches 0 the dynamic memory must be also freed.

```
Calendar::Calendar(const Calendar& other): events(other.events), count(other.count) {
    ++*count;
}

Calendar& Calendar::operator=(const Calendar& other){
  if (this != &other){
      if (--*count == 0){
          delete count;
          delete events;
      }
      count = other.count;
      events = other.events;
      ++*count;
   }
   return *this;
}


Calendar::~Calendar(){
    if (--*count == 0){
```

```
        delete events;
        delete count;
    }
}
```

2. c0 and c1 share the same data because of the assignment c1 = c0. c2 adds an event to the data originally pointed by c0 since it is created using the copy constructor. After adding e2, c2 is reinitialized and starts pointing to different data locations.

   Output:

   c0:
   Important Meeting
   Trip to Cairo
   c1:
   Important Meeting
   Trip to Cairo
   c2:
   Visit to Museum