Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 01-09-2020

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

Exam duration: 2 hours and 15 minutes (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 (12 points)

You have to implement an on demand video system. The main functionality of your application, apart the video streaming service, is the movie search. Each movie is uniquely identified by its title and it is characterized by its director, the list of its actors and its genre.

You should favor the movie search according to the **worst case complexity**. You have to implement the search by movie title and also by keywords (i.e., a subset of the words included in movie titles).
In particular, you have **to provide the declaration of the class `OnDemandVideosSystem`** and implement its following methods:

1. `void addMovie(const string &p_title, const string &p_director, const vector<string> &p_actors, const string &p_genre);`

   which adds a movie and its relevant information to the system. Provide also the method worst case complexity.

2. `void searchMovie(const string &search_title);`

   which prints the movie details given its complete title (for this you can rely on the `void print()` method which implementation is not requested). Provide also `searchMovie()` worst case complexity.

3. `void searchMovie(const string &search_string, const string &criterion);`

   which searches a movie starting from a string including the keywords separated by spaces and a criterion (which possible values are "OR" ,"AND"). The method prints all the movie titles which satisfy the search criterion. "AND" means that a title satisfies the search condition if all keywords are included in the title, "OR" means that a title satisfies the search criterion if at least one of the keywords is included in the title.

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

As an example, if your system includes the movies ***Star wars The return of the Jedi*** and ***Star wars The Force Awakens***, the method call:

```
system.searchMovie("Force Jedi", "AND");
```

does not print any movie title, while the call:

```
system.searchMovie("Force Jedi", "OR");
```

prints:

Movies found
Star wars The Force Awakens
Star wars The return of the Jedi

You can rely on the folowing functions available within the utilities.h header file:

```
// Extract the set of keywords included in the string s
set<string> computeKeywords(const string &s);

// Returns the words in s separated by d
set<string> splitWords(const string &s, char d = ' ');

// Returns the intersection between set1 and set2
set<string> setIntersect(const set<string> &set1, const set<string> &set2);

// Returns the union between set1 and set2
set<string> setUnion(const set<string> &set1, const set<string> &set2);
```

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 movie is already present in the system when it is added), simply write error messages in cerr.

Solution 1

The header and the source files of the class OnDemandVideosSystem are the following:

```
/* OnDemandVideosSystem.h */

#ifndef ONDEMANDVIDEOS_ONDEMANDVIDEOSSYSTEM_H
#define ONDEMANDVIDEOS_ONDEMANDVIDEOSSYSTEM_H

#include "MovieData.h"
#include "utilities.h"

#include <set>
#include <map>
#include <vector>
#include <iostream>


using std::set;
using std::map;
using std::vector;
using std::cout;
using std::cerr;
using std::endl;


class OnDemandVideosSystem {
   map<string,MovieData> movies;     // store movies data
   map<string,set<string>> keywords; // store for each keywords
                                     // the set of movies including that keyword

public:

   void searchMovie(const string &search_title) const;
   void searchMovie(const string &search_string, const string &criterion) const;
```

```
    void addMovie(const string &p_title, const string &p_director,
                  const vector<string> &p_actors,
                  const string &p_genre);
};

#endif //ONDEMANDVIDEOS_ONDEMANDVIDEOSSYSTEM_H


/* OnDemandVideosSystem.cpp */
#include "OnDemandVideosSystem.h"


void OnDemandVideosSystem::searchMovie(const string &search_string, const string &criterion) const{
    // for the sake of simplicity assume that the keywords extracted from the
    // search_string are already stored in the system

    set<string> search_keywords_set = splitWords(search_string);
    // Final set of titles including the keywords
    set<string> titles;

    auto it = search_keywords_set.begin();
    titles = keywords.at(*it); // initialize titles with the first keyword

    it++; // move the iterator to consider the next keyword;


    std::function<set<string>(const set<string> &, const set<string> &)> criterion_function;

    if (criterion == "OR"){ // for OR criterion we need to use the union of
                            //  the movies title sets
        criterion_function = setUnion;
    }
    else if (criterion == "AND"){ // for AND criterion we need to use the intersection of
                                  //  the movies title sets
        criterion_function = setIntersect;
    }
    else {
        // Wrong search criterion, print error message and return
        cerr<< "Wrong search criterion" << endl;
        return;
    }
    // for each remaining keyword compute the union or intersection of
    // titles, given the selected search criterion
    for(; it!=search_keywords_set.cend(); ++it){
        auto current_titles = titles;
        titles = criterion_function(current_titles, keywords.at(*it));

    }
    cout << "Movies found" <<endl;
    for (auto it2 = titles.cbegin(); it2 !=titles.cend(); ++it2)
        cout << *it2 <<endl;

}

void OnDemandVideosSystem::searchMovie(const string &search_title) const{
```

```
    // search the movie title in the movies map data structure
    if (movies.find(search_title)!=movies.cend()){
        cout << search_title << endl;
        movies.at(search_title).print();
    }

    else
        cout << search_title << " not present in the videos system" << endl;
}
void OnDemandVideosSystem::addMovie(const string &p_title, const string &p_director,
            const vector<string> &p_actors,
            const string &p_genre){

    if (movies.find(p_title)!=movies.cend())
        // check if the movie is already stored and print error message
        cerr << p_title << " already in the system" << endl;
    else{
        // store the movie in the movies map data structure
        MovieData movieData(p_title,p_director,p_actors,p_genre);
        movies[p_title] = movieData;

        set<string> k_words = computeKeywords(p_title);
        // For each keyword, add the movie to the keyword set
        for (const string& k: k_words)
            keywords[k].insert(p_title);

        cout << p_title << " added to the videos system" << endl;

    }



}
```

Additionally, the class MovieData is introduced:

```
    /* MovieData.h */
#ifndef ONDEMANDVIDEOS_MOVIEDATA_H
#define ONDEMANDVIDEOS_MOVIEDATA_H

#include <string>
#include <vector>
#include <set>
#include <iostream>

#include "utilities.h"

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

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

class MovieData {
    string director;
    set<string> title_keywords;
    vector<string> actors;
    string genre;
```

```
public:
   const vector<string> &getActors() const;
   const set<string> &getTitleKeywords() const;


public:
   MovieData() = default;
   MovieData(const string &p_title, const string &p_director,
           const vector<string> &p_actors,
           const string &p_genre);

   void print() const;


};


#endif //ONDEMANDVIDEOS_MOVIEDATA_H
```

```
/* MovieData.cpp */
```

```
#include "MovieData.h"

const vector<string> &MovieData::getActors() const {
   return actors;
}

const set<string> &MovieData::getTitleKeywords() const {
   return title_keywords;
}

MovieData::MovieData(const string &p_title, const string &p_director,
                const vector<string> &p_actors,
                const string &p_genre) : director(p_director),
                title_keywords(computeKeywords(p_title)),
                actors(p_actors),
                genre(p_genre)
                {}

void MovieData::print() const{
   cout << "Director" << endl;
   cout <<  director << endl;
   cout << "Starring" << endl;
   for (auto it = actors.cbegin(); it != actors.cend();++it)
      cout << *it << endl;
   cout << "Genre " << genre << endl;

}
```

The class MovieData includes all the movie relevant information and the set of keywords included in the movie title. This set is used by the keywords search functionality of the class OnDemandVideosSystem and is filled by the constructor. In this way, the set is created once for all and multiple searches can reuse it.

The class OnDemandVideosSystem, to optimize the keywords search, needs to store the movies data and it introduces also an additional map which associates to each keyword the set of titles that include such keyword. Since we need to optimize the worst case complexity, a map is the best container to use for storing both movies and keywords, providing $O(log(n))$ complexity for searching a movie by title and to add a new movie (n is the number of movies stored in the system). Note that, in particular the addMovie method complexity is $O(log(n) + k_2 \cdot (log(k_1) + log(n_2))$ where k_1 is the number of keywords in the system, k_2 is the number of keywords in the search string and n_2 is the maximum number of movies associated with a keyword. The first term is due to check if a movie is already in the system and to add the movie data. The second term is due to access the keywords map

and to add in the associated set the new title, which is performed for every keyword in the search string (hence the very worst case complexity of addMovie is $O(k_1 \cdot (log(k_1) + log(n)))$.

Exercise 2 (12 points)

You have to implement a **parallel function** that receives as input the extrema of an interval and returns all the prime numbers in the given range. The prototype of the function is the following:

```
std::vector<unsigned> get_prime_numbers (unsigned, unsigned);
```

Moreover, you have to complete the implementation of the **main** function, considering that the two extrema of the range are given by the user through the **command-line**. The partial implementation of the **main** function is reported below:

```
int main (int argc, char *argv[])
{
  // init
  MPI_Init(&argc, &argv);

  // initialize rank and size
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // read parameters from command-line
  unsigned min = // Your code goes here
  unsigned max = // Your code goes here

  // get the vector of prime numbers
  std::vector<unsigned> prime_numbers = get_prime_numbers(min, max);

  // rank 0 prints the prime numbers
  if (rank == 0)
  {
    std::cout << "prime numbers between " << min << " and " << max
              << " are:" << std::endl;
    for (unsigned n : prime_numbers)
      std::cout << n << " ";
    std::cout << std::endl;
  }

  // finalize
  MPI_Finalize();

  return 0;
}
```

In your implementation you can rely on the function:

```
bool is_prime (unsigned n);
```

which returns **true** if n is a prime number and **false** otherwise.

Note: the prime numbers returned by the function get_prime_numbers should not necessarily be ordered.

Solution 2

To complete the **main** function in order to read the two parameters , it is enough to add the following lines:

```
// read parameters from command-line
unsigned min = std::stoi(argv[1]);
```

```
unsigned max = std::stoi(argv[2]);
```

Since the two extrema are read from command-line, they are known to all processors, thus there is no need to communicate them before performing the computation.

The implementation of the function get_prime_numbers is reported below:

```
std::vector<unsigned> get_prime_numbers (unsigned min, unsigned max)
{
 // initialize rank and size
 int rank, size;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 // initialize vector of prime numbers in each rank
 std::vector<unsigned> my_prime_numbers;

 // loop over the given range (cyclic partitioning)
 for (unsigned n = min + rank; n <= max; n += size)
 {
  // if the current number is prime, insert it in the corresponding vector
  if (is_prime(n))
   my_prime_numbers.push_back(n);
 }

 // collect all partial results in a single vector
 std::vector<unsigned> all_prime_numbers;
 for (int r = 0; r < size; ++r)
 {
  if (r == rank)
  {
   // get and communicate how many prime numbers the current core has found
   unsigned n_primes = my_prime_numbers.size();
   MPI_Bcast(&n_primes, 1, MPI_UNSIGNED, rank, MPI_COMM_WORLD);

   // insert the new values at the end of the vector of prime numbers and
   // communicate them to all the other cores
   unsigned old_n_primes = all_prime_numbers.size();
   all_prime_numbers.insert(all_prime_numbers.end(),
                            my_prime_numbers.cbegin(),
                            my_prime_numbers.cend());

   MPI_Bcast(all_prime_numbers.data() + old_n_primes, n_primes,
          MPI_UNSIGNED, rank, MPI_COMM_WORLD);
  }
  else
  {
   // get the new number of primes
   unsigned n_primes;
   MPI_Bcast(&n_primes, 1, MPI_UNSIGNED, r, MPI_COMM_WORLD);

   // resize the vector of prime numbers and get the new values
   unsigned old_n_primes = all_prime_numbers.size();
   all_prime_numbers.resize(old_n_primes + n_primes);
   MPI_Bcast(all_prime_numbers.data() + old_n_primes, n_primes,
          MPI_UNSIGNED, r, MPI_COMM_WORLD);
  }
 }

 return all_prime_numbers;
}
```

The two parameters min and max are known to all processors, as well as, of course, all numbers in the range [min, max]. Therefore, no communication is required and the interval can be split among the different cores by using, for instance, a cyclic partitioning schema. This has the advantage that the case in which the length of the interval is not a multiple of the number of available processors is automatically managed without need of additional computations.

Each core relies on the function is_prime and stores in the vector my_prime_numbers the prime numbers that it finds. All these partial results must be collected in the single vector all_prime_numbers before being returned, so that all cores return the full set of prime numbers in the considered interval.

Note that to collect all partial results in all_prime_numbers we cannot use a method as AllGather. Indeed, all vectors my_prime_numbers can have, in principle, different dimension. To deal with this, each core has to communicate, first of all, the size of its my_prime_numbers and then all the prime numbers that it has found, that are appended at the end of all_prime_numbers.

Note: if you still want to use AllGather, a possible method would be the following: first of all, the maximum number of primes found by the processors should be communicated through an AllReduce, by using MPI_MAX to compare the partial results obtained by all processors. Having this value, all cores can resize their vector my_prime_numbers, adding at the end some padding (for instance, zeros) that has to be removed at the end from the global result, after the AllGather.

Exercise 3 (8 points)

The provided code implements a data structure called DocumentStore that stores and manages Documents. A Document is characterized by some text (implemented as a std::string) and an id (implemented as size_t). DocumentStore contains two arrays of Document: one for completed documents (instance variable docs) and one for drafts (instance variable docsDraft). The size of the array docs is set in the DocumentStore constructor and can be configured by the users. The size of docsDraft is fixed and equal to 10 (const DRAFT_SIZE). Documents can be added to the arrays through methods addDocument and saveAsDraft respectively. In both cases, if the array is already full no actions are taken. All the documents stored in a DocumentStore can be visualized through method print.

The header and implementation files of class Document are provided along with the header file of DocumentStore while DocumentStore.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 DocumentStore** in order to implement a ***like-a-value*** behavior. **Note that while completed documents must be copied between DocumentStore, drafts must not be transferred during the operation but just emptied.** For example, after statement `a = b`, where a and b are DocumentStores, a must store the completed documents of b and no draft.
2. **List and motivate** the program **output** considering the main file provided.

Provided source code:

- **Document.h**

```
class Document {
private:
    string text;
    size_t id;
public:
    Document(){};
    Document(const string& text, size_t id): text(text), id(id){};
    const string& getText() const;
    size_t getId() const;
};
```

- **Document.cpp**

```
const string& Document::getText() const {
    return text;
};
size_t Document::getId() const {
    return id;
};
```

- **DocumentStore.h**

```
const int DRAFT_SIZE = 10;

class DocumentStore {
private:
    Document *docs;
    Document *docsDraft;
    size_t size;
    size_t curr;
    size_t currDraft;
public:
    DocumentStore(int);
    DocumentStore(const DocumentStore&);
    DocumentStore& operator=(const DocumentStore&);
    ~DocumentStore();
    void addDocument(const Document&);
    void saveAsDraft(const Document&);
    void print() const;
};
```

- **DocumentStore.cpp**

```
DocumentStore::DocumentStore(int sz): size(sz), curr(0), currDraft(0){
    docs = new Document[sz];
    docsDraft = new Document[DRAFT_SIZE];
}

// MISSING IMPLEMENTATION

void DocumentStore::addDocument(const Document &doc) {
    if (curr < size) {
        docs[curr++] = doc;
    }
}

void DocumentStore::saveAsDraft(const Document &doc) {
    if (curr < DRAFT_SIZE) {
        docsDraft[currDraft++] = doc;
    }
}

void DocumentStore::print() const {
    cout << "Documents:" << endl;
    for (size_t i = 0; i < curr; i++){
        cout << docs[i].getText() << endl;
    }
    cout << "Drafts:" << endl;
    for (size_t i = 0; i < currDraft; i++){
        cout << docs[i].getId() << endl;
    }
}
```

- **main.cpp**

```
int main() {
    size_t id = 945;
    Document d0("Apple", id++);
    Document d1("Orange", id++);
    Document d2("Melon", id++);
```

```
    Document d3("Peach", id++);
    Document d4("Strawberry", id++);

    DocumentStore ds0(3);
    ds0.addDocument(d0);
    ds0.saveAsDraft(d1);

    DocumentStore ds1(ds0);
    ds1.addDocument(d2);
    ds1.addDocument(d3);

    DocumentStore ds2(2);
    ds2.addDocument(d1);
    ds2.addDocument(d4);
    ds2.saveAsDraft(d2);
    ds2.saveAsDraft(d3);

    DocumentStore ds3(3);
    ds2.addDocument(d0);
    ds2.addDocument(d1);
    ds3 = ds2;
    ds3.addDocument(d3);

    cout << "- ds0 -" << endl;
    ds0.print();
    cout << "- ds1 -" << endl;
    ds1.print();
    cout << "- ds2 -" << endl;
    ds2.print();
    cout << "- ds3 -" << endl;
    ds3.print();

    return 0;
}
```

Solution 3

1. In order to implement a *like-a-value* behavior, we simply have to copy the completed documents of the target DocumentStore in newly created arrays. Because the drafts must not be copied but simply deleted we can assign index currDraft to 0 without changing the actual array. Note that in the copy operator, in case the two DocumentStores have the same size, there is no need to create a new array but we can simply reuse the existing one and refresh the indexes.

```
DocumentStore::DocumentStore(const DocumentStore& other) {
    size = other.size;
    curr = other.curr;
    currDraft = 0;
    docs = new Document[other.size];
    for (size_t i = 0; i < curr; ++i){
        docs[i] = other.docs[i];
    }
    docsDraft = new Document[DRAFT_SIZE];
}

DocumentStore& DocumentStore::operator=(const DocumentStore& other){
    if (size != other.size){
        delete[] docs;
        docs = new Document[other.size];
```

```
        size = other.size;
    }
    curr = other.curr;
    currDraft = 0;
    for (size_t i = 0; i < curr; ++i){
        docs[i] = other.docs[i];
    }
    return *this;
}

DocumentStore::~DocumentStore(){
    delete[] docs;
    delete[] docsDraft;
}
```

2. Underlying arrays are not shared among different DocumentStores. Therefore, after creating $ds1$ using $ds0$, arrays *docs* of the two instances contain the same documents but they are completely separated in memory. In fact, after adding new documents to $ds1$, no changes are made to $ds0$. After adding $d3$ to $ds3$ nothing happens since, after copying $ds2$ its size is equal to 2.

   Output:

```
- ds0 -
Documents:
Apple
Drafts:
945
- ds1 -
Documents:
Apple
Melon
Peach
Drafts:
- ds2 -
Documents:
Orange
Strawberry
Drafts:
946
949
- ds3 -
Documents:
Orange
Strawberry
Drafts:
```