Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 31-8-2021

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

<u>**Exam duration: 2 hours**</u>

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 have to develop the `TimeSeries` class to manage efficiently time series, i.e., **series of data points indexed in time order**. In particular, your goal is to store sequences of `float` values and to optimize the **worst case complexity** of all methods.

The time is denoted by a two level granularity, i.e., by specifying a date and a time stamp. Both are represented as strings in the format *"yyyy/mm/dd"* and *"hh:mm:ss"*, respectively. Note that in this way, by relying on the order relation of strings, the order relations for dates and times are preserved (time goes from "00:00:00" to "23:59:59").

You have to:

1. provide the declaration of the `TimeSeries` class (you can focus only on the data structures and any additional methods you possibly will introduce but you can omit from the declaration the methods listed below).

2. Implement the method

   `void add_registration(const string & date, const string & time, float value);`

   which adds a value to the time series.

3. `float get_registration(const string & date, const string & time) const;`

   which returns the time series value at `date` and `time` or NaN in case the value was not previously added in the time series.

4. provide alternatively the implementation of

   `float get_registration_by_date(const string & date) const;`

   or

   `float get_registration_by_time(const string & time) const;`

   which compute the average across all the values stored at a given date or time stamp (note that, even if it is requested to implement one method, **the data structures you choose have to optimize the worst case complexity of both**).

5. `vector<float> get_registration_range(const string & date1, const string & time1, const string & date2) const;`

   which returns the set of values stored starting from date1 and time1 up to date2. date2 data should not be included in the result. If date2<date1 or date1 is not included in the time series, the vector returned as result is empty.

6. TimeSeries intersect_time_series(const TimeSeries & ts) const;

   which returns the intersection of the time series with the one specified as parameter. The values of the TimeSeries returned as result are obtained as mean of the corresponding values of the two initial time series. For example, if the two intial time series stores the registrations ("2021/08/31", "11:30:00", 1) and ("2021/08/31", "11:30:00", 2) the registration ("2021/08/31", "11:30:00", 1.5) is stored in the result.

7. Finally, provide the **worst case complexity** of all the methods you have implemented (with the exception of the intersect method).

Note that you can include any additional helper methods if needed, but in that case you have to provide also their implementation and complexity.

Solution 1

The declarations of TimeSeries, class is reported below:

```
class TimeSeries {

   std::map<std::string, std::map<std::string, float > > registration_by_date_time; //r[date][time]= value
   std::map<std::string, std::map<std::string, float > > registration_by_time_date; //r[time][date]= value

   bool date_exist(const std::string & date) const;
   bool time_exist(const std::string & time) const;
   bool date_time_exist(const std::string & date, const std::string & time) const;
public:
   void add_registration(const std::string & date, const std::string & time, float value);
   float get_registration(const std::string & date, const std::string & time) const;
   float get_registration_by_date(const std::string & date) const;
   float get_registration_by_time(const std::string & time) const;
   std::vector<float> get_registration_range(const std::string & date1, const std::string & time1, const std::
     string & date2) const;
   TimeSeries intersect_time_series(const TimeSeries & ts) const;
   void print() const;

};
```

While the implementation of the methods are reported below:

```
bool TimeSeries::date_exist(const std::string & date) const {

   const std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin =
     registration_by_date_time.find(date);
   if (cit_begin == registration_by_date_time.cend())
      return false;
   else
      return true;
}

bool TimeSeries::time_exist(const std::string & time) const {

   const std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin =
     registration_by_time_date.find(time);
   if (cit_begin == registration_by_time_date.cend())
      return false;
   else
      return true;
}

bool TimeSeries::date_time_exist(const std::string & date, const std::string & time) const{
   if (!date_exist(date))
```

```
        return false;

    const std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin =
      registration_by_date_time.find(date);
    // check if timestamp exists
    const std::map<std::string, float>::const_iterator cinner_it_begin = cit_begin->second.find(time);
    if (cinner_it_begin == cit_begin->second.cend())
        return false;

    return true;
}

void TimeSeries::add_registration(const std::string & date, const std::string & time, float value) {
    registration_by_date_time[date][time]= value;
    registration_by_time_date[time][date]= value;
}

float TimeSeries::get_registration(const std::string & date, const std::string & time) const {
    // check if date exists
    if (!date_exist(date))
        return std::numeric_limits<float>::quiet_NaN();

    const std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin =
      registration_by_date_time.find(date);

    // check if timestamp exists
    const std::map<std::string, float>::const_iterator cinner_it_begin = cit_begin->second.find(time);
    // if not return NaN
    if (cinner_it_begin == cit_begin->second.cend())
        return std::numeric_limits<float>::quiet_NaN();
    // else return corresponding value
    return cinner_it_begin->second;
}

float TimeSeries::get_registration_by_date(const std::string & date) const {

    // check if date exists
    if (!date_exist(date))
        return std::numeric_limits<float>::quiet_NaN();

    const std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin =
      registration_by_date_time.find(date);

    std::map<std::string, float>::const_iterator cinner_it_begin = cit_begin->second.cbegin();
    // read sequentially internal map and compute mean
    float sum = 0;
    size_t count =0;
    for (;cinner_it_begin != cit_begin->second.cend(); ++cinner_it_begin){
        sum += cinner_it_begin->second;
        ++count;
    }

    // We are sure that at least one element for the date exists (we cannot add a value
    // associated with a date without specifying a time stamp)
    return sum/count;


}
```

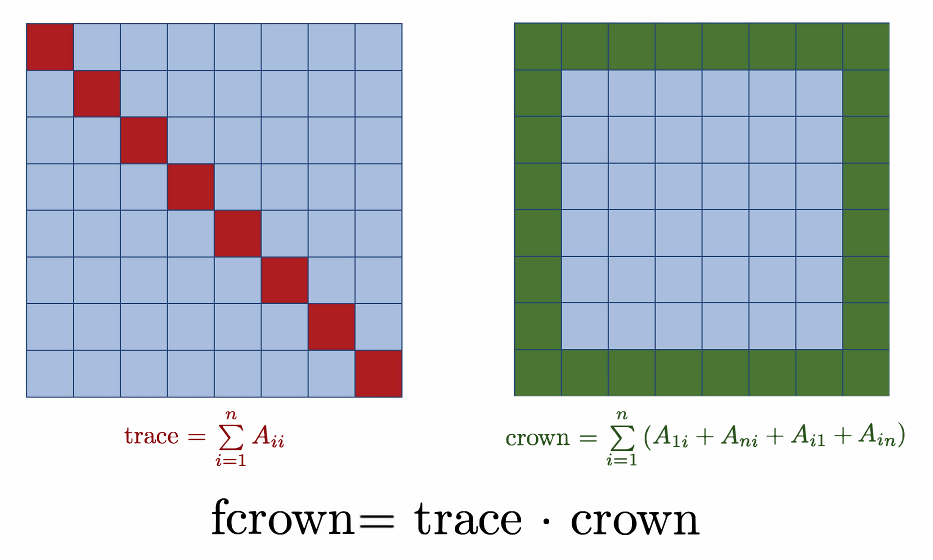
```
float TimeSeries::get_registration_by_time(const std::string & time) const {

   // check if time exists
   if (!time_exist(time))
      return std::numeric_limits<float>::quiet_NaN();

   const std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin =
     registration_by_time_date.find(time);

   std::map<std::string, float>::const_iterator cinner_it_begin = cit_begin->second.cbegin();
   // read sequentially internal map and compute mean
   float sum = 0;
   size_t count =0;
   for (;cinner_it_begin != cit_begin->second.cend(); ++cinner_it_begin){
      sum += cinner_it_begin->second;
      ++count;
   }

   // We are sure that at least one element for the time exists (we cannot add a value
   // associated with a time without specifying a date)
   return sum/count;

}

std::vector<float>
TimeSeries::get_registration_range(const std::string & date1, const std::string & time1, const std::string &
     date2) const {
   std::vector<float> result;

   if (!date_exist(date1) || date2 < date1)
      return result;

   std::map<std::string,std::map<std::string, float>>::const_iterator cit_begin = registration_by_date_time.
     find(date1);

   while (cit_begin->first < date2 && cit_begin != registration_by_date_time.cend()){
      std::map<std::string, float>::const_iterator cinner_it_begin = cit_begin->second.find(time1);
      while (cinner_it_begin != cit_begin->second.cend()){
         result.push_back(cinner_it_begin->second); // store result
         ++cinner_it_begin; // move to the next time slot if any
      }
      // move to the next day
      ++cit_begin;

   }
   return result;
}

TimeSeries TimeSeries::intersect_time_series(const TimeSeries &ts) const{
   TimeSeries result;

   std::map<std::string,std::map<std::string, float>>::const_iterator cit = ts.registration_by_date_time.
     cbegin();
   // move to the higher date the iterator
   if (registration_by_date_time.cbegin()->first > cit->first)
      cit = registration_by_date_time.cbegin();
   else
```

```
        cit = registration__by__date__time.lower__bound (cit->first);

    // From now on, cit is the iterator to the first level map of the range of interest

    std::map<std::string,std::map<std::string, float>>::const__iterator cit__end = ts.registration__by__date__time
      .cend();
    --cit__end; // this way cit__end points to the last element of the second time series

    while (cit != registration__by__date__time.cend() and cit->first <= cit__end->first){
        std::map<std::string, float>::const__iterator cinner__it= cit->second.cbegin();
        // cinner__it is the iterator of the second level map
        // cit-> first is the date of the current element
        // cinner__it->first is the time of the current element
        // cinner__it->second is the value of the current element

        // process the current day
        while (cinner__it != cit->second.cend()){
            // if the current element is included also in ts
            if (ts.date__time__exist(cit->first, cinner__it->first))
                // add to the result the mean of the two registrations
                result.add__registration(cit->first, cinner__it->first, (cinner__it->second + ts.get__registration(cit->
      first, cinner__it->first))/2);
            ++cinner__it; // move to the next registration
        }

        ++cit; // move to the next day
    }

    return result;
}

void TimeSeries::print() const {

    std::map<std::string,std::map<std::string, float>>::const__iterator cit__begin = registration__by__date__time.
      cbegin();

    while (cit__begin != registration__by__date__time.cend()){
        std::map<std::string, float>::const__iterator cinner__it__begin = cit__begin->second.cbegin();
        while (cinner__it__begin != cit__begin->second.cend()){
            std::cout << cit__begin->first << " " <<   cinner__it__begin->first << " " << cinner__it__begin->second
      << std::endl;
            ++cinner__it__begin; // move to the next time slot if any
        }
        // move to the next day
        ++cit__begin;

    }
}
```

To optimize both the get__registration__by__date() and get__registration__by__time() methods two nested maps are introduced that are accessed through date and time and vice versa, respectively. In this way the methods have a complexity which is logarithmic in the number of the accessed keys plus a linear term due to the number of elements which mean is computed. The methods date__exist(), time__exist(), date__time__exist() are introduced to check if the accessed elements exist and have complexity $O(log(D))$, $O(log(T))$ and $O(log(D) + log(T))$ where D is the number of different dates and T is the total number of registrations in a day.

The add__registration() method add elements in the two nested maps and has complexity $O(log(D) + log(T))$. The get__registration() method has complexity $O(log(D) + log(T))$ where the first term is due to the call to date__exist() and the second to the acess to the inner map. The method get__registration__by__date() has complexity $O(log(D) + T)$ where the first term is due to the call to date__exist() and to access the external map, while the second term is due, as discussed previously, to the sequential scan of the registration of the specific date. Similarly, the method `get_registration_by_time()` has complexity $O(log(T) + D)$. Finally, the method `get_registration_range()` has complexity $O(log(D) + K) = O(N)$ in the very worst case, where K is the number of registrations between the specified time (date1, time1) and date2 which, in the very worst case equals to the number of elements in the time series N.

Exercise 2 (11 points)



trace $= \sum_{i=1}^{n} A_{ii}$
crown $= \sum_{i=1}^{n} (A_{1i} + A_{ni} + A_{i1} + A_{in})$
fcrown= trace · crown

Figure 1: *fcrown* property computation

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

```
double fcrown_property (const std::string& filename);
```

It computes the *fcrown property* of a **squared matrix** `A`, which should be **read from the file** whose name is passed as parameter and represented through the class `la::dense_matrix` (header file reported below).

The *fcrown property* of matrix `A` is defined as:

$$\text{fcrown(A)} = \text{trace(A)} \cdot \text{crown(A)},$$

where

$$\text{trace(A)} = \sum_{i=1}^{n} A_{ii}$$

and

$$\text{crown(A)} = \sum_{i=1}^{n} (A_{1i} + A_{ni} + A_{i1} + A_{in})$$

as reported in Figure 1. Note that, when computing crown(A), **the corner elements are considered twice**. For instance, element A_{11} is included both when summing the elements on the first row and when summing the elements in the first column (other than when computing the trace of the matrix).

For the computation, you can assume that **the number of rows in the matrix is multiple of the number of available cores**.

The header file of the `dense_matrix` class is provided below.

***** dense_matrix.hpp**

```
#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 &);
}
```

Solution 2

```
  double fcrown_property (const std::string& filename)
  {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // initialize matrix and number of rows/columns
    la::dense_matrix A;
    unsigned n = 0;

    // read matrix from rank 0
    if (rank == 0)
    {
      std::ifstream ifs(filename);
      A.read(ifs);

      n = A.rows();

      A.print(std::cout);
    }

    // broadcast number of rows
    MPI_Bcast(&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

    // prepare local matrices
    unsigned local_n = n / size;
    la::dense_matrix local_A(local_n, n);

    // scatter
    MPI_Scatter(A.data(), local_n * n, MPI_DOUBLE,
                local_A.data(), local_n * n, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    // sum values on the diagonal and on first and last columns
    double trace = 0.;
    double crown = 0.;
    for (unsigned i = 0; i < local_n; ++i)
    {
      trace += local_A(i, i + rank*local_n);
      crown += (local_A(i,0) + local_A(i,n-1));
    }

    // for the first and last ranks, sum values on the first and last row
    if (rank == 0)
    {
      for (unsigned j = 0; j < n; ++j)
        crown += local_A(0,j);
    }
    if (rank == size - 1)
    {
```

```
    for (unsigned j = 0; j < n; ++j)
      crown += local_A(local_n-1,j);
  }

  // allreduce
  MPI_Allreduce(MPI_IN_PLACE, &trace, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE, &crown, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  return trace * crown;
}
```

Exercise 3 (8 points)

The code provided below models the interactions between a mobile phone and the tariffs applied by its carrier on operations like sending messages and making calls.

Each Carrier sets a cost per sms sent, as well as a cost per minute spent in a call. Such costs can be retrieved through dedicated methods, while the updatePrices() method allows a carrier to change the costs applied to users.

The class Phone contains the information relative to a single phone, that include a list of contacts and an available credit together with a pointer to the current carrier. The method textToContact() will send a message to a contact and charge the user for the message only if the phone has enough credit and the contact is registered in the phone. The method callContact() will call a contact for the requested duration, unless the available credit does not allow for a call of such length. The method textAllContacts() will attempt to broadcast the same text to all registered contacts, and finally the method changeCarrier() allows a phone owner to change carrier to the phone. 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 if you are attending the online version of the exam). Notice that each answer requires you to type a single number!** Moreover, **it is mandatory for you to develop your solution motivating the results you achieved on the sheets** (that will be possibly uploaded). **Only providing the final answers is not enough for obtaining points in this section.**

1. What is the credit of the phone after the exection of textToContact() at line 10?
2. What is the value stored in callTime after the execution of callContact() at line 11?
3. What is the value stored in messageSent after the execution of textToContact() at line 14? (1 for true, 0 for false)
4. Depending on the last answer, either the instruction callContact() at line 18 or textAllContacts() is executed. What is the value returned by the executed instruction?

Provided source code:

- **Product.h**

```
#include <string>

class Carrier {
private:
    unsigned smsCost;
    unsigned callCostPerMinute;

public:
    Carrier(unsigned a, unsigned b);

    unsigned billSms();
    unsigned billPhoneCall(unsigned minutes);

    void updatePrices(unsigned sms, unsigned call = 0);

    void sendMessage(std::string msg);
};
```

- **Product.cpp**

```
#include "Carrier.h"

Carrier::Carrier(unsigned a, unsigned b) : smsCost(a), callCostPerMinute(b) {}

unsigned Carrier::billSms() {
  return smsCost;
}

unsigned Carrier::billPhoneCall(unsigned minutes) {
  return callCostPerMinute * minutes;
}

void Carrier::updatePrices(unsigned sms, unsigned call) {
  smsCost = sms;
  callCostPerMinute = call ? call : callCostPerMinute;
}

void Carrier::sendMessage(std::string msg) {
  /* Implementation is irrelevant to the exercise */
}
```

- **Shop.h**

```
#include <memory>
#include <unordered_set>
#include "Carrier.h"

class Phone {
private:
    std::unordered_set<std::string> contacts;
    unsigned credit;
    std::shared_ptr<Carrier> carrier;

public:
    Phone(std::unordered_set<std::string> c, unsigned cr,
          std::shared_ptr<Carrier> ca);

    bool textToContact(std::string contact, std::string content);

    unsigned callContact(std::string contact, unsigned mins);

    unsigned textAllContacts(std::string msg);

    void changeCarrier(std::shared_ptr<Carrier> c);
};
```

- **Shop.cpp**

```
#include "Phone.h"

Phone::Phone(std::unordered_set<std::string> c, unsigned cr,
             std::shared_ptr<Carrier> ca) : contacts(c), credit(cr),
                                            carrier(ca) {}

bool Phone::textToContact(std::string contact, std::string content)  {
  if (contacts.count(contact) && credit >= carrier->billSms()) {
    credit -= carrier->billSms();
    carrier->sendMessage(content);
    return true;
  }
  return false;
}

unsigned int Phone::callContact(std::string contact, unsigned int mins) {
  unsigned m = 0;
  if (contacts.count(contact)) {
    while (m < mins && credit > carrier->billPhoneCall(m + 1)) {
      ++m;
    }
    credit -= carrier->billPhoneCall(m);
  }
  return m;
}

unsigned int Phone::textAllContacts(std::string msg) {
  unsigned ret = 0;
  bool canSend = true;
  for (std::string c : contacts) {
    if (canSend) {
      canSend = textToContact(c, msg);
    }
    ret = ret + canSend;
  }
  return ret;
}

void Phone::changeCarrier(std::shared_ptr<Carrier> c) {
  carrier = c;
}
```

- **main.cpp**

```
#include <iostream>
#include "Phone.h"

int main() {

  std::shared_ptr<Carrier> blueCarrier = std::make_shared<Carrier>(3, 4);
  std::shared_ptr<Carrier> redCarrier = std::make_shared<Carrier>(5, 5);
  Phone phone({"Alice", "Betty", "Charles", "Mom"}, 75, blueCarrier);

  phone.textToContact("Robert", "Hi rob, how are you?");
  unsigned callTime = phone.callContact("Mom", 12);
  std::cout << "First call time is " << callTime << std::endl;
  blueCarrier->updatePrices(10);
  bool messageSent = phone.textToContact("Betty", "Please call me at 9:30 pm");
  phone.changeCarrier(redCarrier);
  if (!messageSent) {
    std::cout << "Last call lasted ";
    std::cout << phone.callContact("Alice", 5);
    std::cout << " minutes" << std::endl;
  } else {
    std::cout << "Text sent to ";
    std::cout << phone.textAllContacts("You are invited to my party!!");
    std::cout << " contacts" << std::endl;
  }
  return 0;
}
```

Solution 3

1. This first method has as first argument a string that does not correspond to any registered user in the phone, therefore no message is sent and the overall credit remains 75.
2. This time the call is performed towards a registered user. Since the overall cost of the call is $12 * 4 = 48$, it is possible to perform the call for all the 12 minutes requested.
3. Despite the increased cost for sms, the phone still has enough credit for sending it, so the result is 1 (true).
4. The method to be executed is the one at line 22. Before the execution of the method, the phone still has a credit of 17. Thanks to the carrier change, it is possible to send three messages with the available credit. Since the list of registered users contains four people, the method sends three because it has not enough credit for sending a fourth one to the last contact.