Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 15-01-2021

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 have to implement a simplified Piazza site to support students Q&A on University courses topics. 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.

Students (uniquely identified by their `login`) can add posts and answers to the questions of their colleagues. The course instructor reads posts and is notified by the system every time she/he logs in with the new unread posts added by her/his students. Posts are stored in a `vector` within the `PiazzaSite` class and are uniquely identified by an `id` (note that `ids` start from 1).

The site provides also statistics on the students activities. In particular, student's views are stored, **also**, within the set `stud_views` which basic elements are of type `StudentViews`, a struct for which also the `operator<` is provided and defined as follows:

```
bool operator<(const StudentViews &view1, const StudentViews &view2) {
   return (view1.views < view2.views)
      or ( (view1.views == view2.views ) and
           (view1.stud_login < view2.stud_login ));
}
```

In particular, provide the declaration of:

1. `Instructor::posts_unread` data structure which stores the `ids` of the unread posts. Note that an instructor can read posts in any order.
2. `PiazzaSite::studs` data structure, whose purpose is to store the students registered to the site.

Moreover, provide the implementation of the following methods:

3. `void Instructor::add_unread_post(size_t post_id);`

   which adds an unread post `id` to `Instructor::posts_unread`. You can neglect error conditions (e.g., the post `id` does not exist) for the method implementation.

4. `void Instructor::read_post(size_t post_id)`

   which removes the `id` of a post read by the instructor from `Instructor::posts_unread`. You can neglect error conditions (e.g., the post `id` does not exist) for the method implementation.

5. `void PiazzaSite::register_student(const string &stud_login, const string &stud_pwd)`

   which adds a new student to the site. You can neglect error conditions (e.g., the student is already registered) for the method implementation.

Figure 1: Piazza Site Class Diagram

6. `void PiazzaSite::add_post(const string &stud_login, const string &date, const string &content)`

   which adds a new post by `stud_login`. You can neglect error conditions (e.g., the student does not exist) for the method implementation.

7. `void PiazzaSite::read_post_student(const std::string &stud_login, size_t post_id))`

   which updates `stud_login` views since the post identified by `post_id` has been read. **You have to manage the error conditions** (e.g., the student or the post do not exist) for the method implementation writing an error message.

8. `void PiazzaSite::read_post_instructor(size_t post_id)`

   which updates `Instructor::posts_unread` data structure since the `post_id` has been read by the Instructor. **You have to manage the error conditions** (e.g., the post does not exist) for the method implementation writing an error message.

9. `void PiazzaSite::top_k_students(unsigned int k)`

   which prints the login of the k students characterized by the largest number of views.

Finally, provide:

10. the **average complexity** of the methods implemented at points 3-9.

Solution 1

In order to optimize the average case complexity, students are stored in an unordered_map, while instructors' unread posts are stored in a list (this is the best data structure, since it optimizes the delete of posts in any position). The declaration of the data structures are reported below:

```
class PiazzaSite {
        ...
        unordered_map<string, Student> studs;
        ...
}


class Instructor {
        ...
        list<size_t> posts_unread;
        ...
}
```

The code implementing Instructor::posts_unread updates and the source file of the class PiazzaSite are reported below.

```
   /* From Instructor.cpp */
void Instructor::add_unread_post(size_t post_id) {
   posts_unread.push_back(post_id);
}

void Instructor::read_post(size_t post_id) {

   posts_unread.remove(post_id);

}
/* PiazzaSite.cpp */
void PiazzaSite::register_student(const std::string &stud_login, const std::string &stud_pwd) {

   Student s(stud_login, stud_pwd);

   studs[stud_login] = s ;

   StudentViews stud_view = StudentViews(stud_login,0);

   stud_views.insert(stud_view);
}

void PiazzaSite::add_post(const std::string &std_login, const std::string &date, const std::string &content)
    {
   const Post post(date,content);

   posts.push_back(post);
   instr.add_unread_post(post.get_id());

}

void PiazzaSite::add_answer(size_t post_id, const std::string &answer) {
   posts[post_id-1].add_answer(answer);
}

void PiazzaSite::read_post_student(const std::string &stud_login, size_t post_id) {

   auto student_it = studs.find(stud_login);
```

```
    if (student_it != studs.end()){

        Student & s = student_it->second;

        StudentViews stud_view(stud_login, s.get_views());

        auto stud_view_it = stud_views.find(stud_view);

        // If the stud_login and the post exist
        if (stud_view_it!= stud_views.end() and post_id <= posts.size()){
            stud_views.erase(stud_view_it);
            stud_view.views++;
            stud_views.insert(stud_view);
            s.inc_views();
        }
        else // print error
            std::cerr << "post " << post_id << " not found" << std::endl;
    }
    else // print error
    std::cerr << stud_login << " not found" << std::endl;

}

void PiazzaSite::read_post_instructor(size_t post_id) {
    if (post_id<= posts.size())
        instr.read_post(post_id);
    else // print error
        std::cerr << "post " << post_id << " not found" << std::endl;
}

void PiazzaSite::top_k_students(unsigned int k) const {
    size_t count =1;
    for (auto it = stud_views.crbegin(); it != stud_views.crend() and count <= k; ++it) {
        cout << it->stud_login << " " << it->views << endl;
        count++;
    }
}
```

A new unread post in the `Instructor` data is added simply through a `push_back` and removed through a `remove`. The average complexities are $O(1)$ and $O(P)$, respectively, where P is the number of unread posts.

`PiazzaSite::register_student` simply adds the student data in the `unordered_map` and inserts also the student with zero views in the `set stud_views`. Hence, the average complexity is $O(1+log(S)) = O(log(S))$, where S is the number of students registered to the site. `PiazzaSite::add_post` is obtained through a `push_back` in the `vector` of posts and by also updating the instructor's unread posts. Hence, the average complexity is $O(1)$.

`PiazzaSite::read_post_student` initially checks if the student is in the `unordered_map studs` and then updates the student's views by, first, removing the student data in the `set stud_views` and then inserting data again after updating the views counter. This is necessary, since `set` objects are implicitly `const`. Hence, the complexity is $O(log(S))$ due to the operation on the `set` data structure.

`PiazzaSite::read_post_instructor` relies on the corresponding operation provided by the `Instructor` class with complexity $O(P)$. Finally, the method `PiazzaSite::top_k_students` has complexity $O(k)$ and is implemented through a reverse scan of the `set stud_views`.

Exercise 2 (11 points)

You have to implement a **parallel function** that computes the Frobenius norm of a given matrix A. Recall that the Frobenius norm of a matrix $A \in \mathbb{R}^{m \times n}$ is given by

$$||A||_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2} . \tag{1}$$

The matrix A is read in the program from a file whose **name is passed through the command-line** and it must be defined as an object of class dense_matrix. **The file is available only at rank 0.**

In particular, you have to:

- complete the implementation of the main function provided below to read the name of the file from command-line.
- implement the function
  double frobenius_norm (**const** std::string & file_name);
  It receives as input the name of the file where the matrix is stored and returns the corresponding Frobenius norm. Notice that the matrix has m rows and n columns; you can assume that the number of rows m is multiple of the number of available processes.

The partial implementation of the **main** function and of the dense_matrix class are reported below:

*** main.cpp

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

  int rank (0), size (0);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  /* YOUR CODE GOES HERE */

  double norm = frobenius_norm(file_name);

  if (rank == 0)
    std::cout << norm << std::endl;

  MPI_Finalize ();
  return 0;
}
```

*** dense_matrix.hpp

```
#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;
};
```

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

  void
  swap (dense_matrix &, dense_matrix &);
}

#endif // DENSE_MATRIX_HH
```

Solution 2

In order to compute the Frobenius norm of a matrix A in parallel, we rely on a block partitioning schema. In particular, rank 0 reads the given matrix from file and scatters its rows among the available processes. The corresponding implementation of the function frobenius_norm is reported below:

```
double
frobenius_norm (const std::string & file_name)
{
  int rank (0), size (0);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  // initialize variables for rows and columns
  unsigned m (0);
  unsigned n (0);

  la::dense_matrix full_A;

  // rank 0 reads the matrix from file and updates m and n accordingly
  if (rank == 0)
  {
    std::ifstream f_stream (file_name);
    full_A.read (f_stream);
    m = full_A.rows ();
    n = full_A.columns ();
  }

  // rank 0 broadcasts m and n to all cores
  MPI_Bcast (&m, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
  MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

  // compute local_m (hyp: the number of rows is multiple of the number of
  // available cores)
  const unsigned local_m = m / size;

  // initialize local matrix
  la::dense_matrix local_A(local_m, n);

  // scatter data
  MPI_Scatter (
    full_A.data (), local_m * n, MPI_DOUBLE,
    local_A.data (), local_m * n, MPI_DOUBLE,
    0, MPI_COMM_WORLD);

  // compute the norm in all cores
  double norm = frobenius_norm (local_A);
```

```
  return norm;
}
```

Notice that both the number of rows and the number of columns m and n must be broadcasted in order to determine the size of the new local matrix.

Given the matrix local_A over all processes, this is passed to an overloaded function frobenius_norm, that is implemented as follows in the corresponding source file:

```
double
frobenius_norm(const la::dense_matrix & local_A)
{
  const unsigned local_m = local_A.rows ();
  const unsigned n = local_A.columns ();

  double norm (0.0);

  for (std::size_t i = 0; i < local_m; ++i)
    for (std::size_t j = 0; j < n; ++j)
      norm += (local_A(i,j) * local_A(i,j));

  /*
    int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  */
  MPI_Allreduce(MPI_IN_PLACE, &norm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  return std::sqrt(norm);
}
```

Each process computes its partial sum by looping over all elements of the matrix local_A. The value of the norm is finally obtained by summing all partial results by relying on the method MPI_Allreduce (passing as first parameter MPI_IN_PLACE to avoid declaring a new variable).

Exercise 3 (8 points)

The provided code implements a class called SweetsBox. This class represent a box which can be filled with two kinds of sweets: candies and cookies, and provides methods for refilling and consuming these items. The size of the box (i.e., how many sweets can be stored in it) is fixed for each object, and stored in the capacity attribute. The implementation of a second class called SweetsSock, which is derived from SweetsBox and can be filled with different kinds of sweets, is also provided.

After carefully reading the code, you have to answer to the following questions in the provided web form.

1. The value returned by box1.eatContent(3) at line 15
2. The value of the attribute box1.cookies after the execution of box1.print() at line 16
3. The value of the attribute sock1.coal after the execution of sock1.refill() at line 18
4. The value of the attribute sock1.candies after the execution of sock1.print() at line 20
5. The value returned by the **first execution** of b.eatContent(1) at line 34
6. The value returned by the **second execution** of b.eatContent(1) at line 34

Provided source code:

- **SweetsBox.h**

```
#include <iostream>

class SweetsBox {
```

```
private:
    unsigned cookies = 0;

protected:
    unsigned const capacity;
    unsigned candies = 0;

public:
    SweetsBox(unsigned c);

    virtual unsigned eatContent(int t);

    void refill();

    void print() const;

    virtual ~SweetsBox(){};
};
```

- **SweetsBox.cpp**

```
#include "SweetsBox.h"

SweetsBox::SweetsBox(unsigned c) : capacity(c) {}

unsigned SweetsBox::eatContent(int t) {
    int i = 0;
    while (i < t && candies > 0 && cookies > 1) {
        ++i;
        --candies;
        cookies = cookies - 2;
    }

    return  candies + cookies;
}

void SweetsBox::refill() {
    candies = capacity/2;
    cookies = capacity - candies;
    std::cout << "Box gets " << candies + cookies << " items" << std::endl;
}

void SweetsBox::print() const {
    std::cout << "Box: " << candies << " - " << cookies << std::endl;
}
```

- **SweetsSock.h**

```
#include "SweetsBox.h"

class SweetsSock : public SweetsBox {
private:
    unsigned coal = 0;

public:

    SweetsSock(unsigned c);

    unsigned eatContent(int t) override;

    void refill();

    void print() const;

    virtual ~SweetsSock(){};
};
```

- **SweetsSock.cpp**

```
#include "SweetsSock.h"

SweetsSock::SweetsSock(unsigned c) : SweetsBox(c/2) {}

unsigned SweetsSock::eatContent(int t) {
    if (coal != 0) {
        --coal;
    }

    if (candies > t) {
        candies = candies - t;
    } else {
        candies = 0;
    }

    return coal;
}

void SweetsSock::refill() {
    candies = capacity;
    coal = capacity/2;
    std::cout << "Sock gets " << candies + coal << " items" << std::endl;
}

void SweetsSock::print() const {
    std::cout << "Sock: " << candies << " - " << coal << std::endl;
}
```

- **main.cpp**

```
#include <iostream>

#include "SweetsSock.h"

void changeAndPrint(SweetsBox &b);

int main() {
```

```
8
9       SweetsBox box1(8);
10      SweetsSock sock1(8);
11
12      std::cout << "box1, sock1 output" << std::endl;
13
14      box1.refill();
15      box1.eatContent(3);
16      box1.print();
17
18      sock1.refill();
19      sock1.eatContent(3);
20      sock1.print();
21
22      SweetsBox box2(32);
23      SweetsSock sock2(32);
24
25      std::cout << "box2, sock2 output" << std::endl;
26
27      changeAndPrint(box2);
28      changeAndPrint(sock2);
29
30  }
31
32  void changeAndPrint(SweetsBox &b){
33      b.refill();
34      b.eatContent(1);
35      b.print();
36  }
```

Solution 3

1. The object `box1` is created at line 9 with a capacity of 8, therefore the `refill()` method will set the value of both `candies` and `cookies` to 4. After that, since `eatContent(int t)` consumes cookies twice as fast than candies, and this call will attempt to consume 6 cookies while only 4 are present, when `cookies` reaches 0 there will be 2 `candies` left in the box, so the returned value will be 2.

2. As explained in the previous answer, after the execution of `box1.eatContent(3)` at line 9 the values of the attributes `candies` and `cookies` will be 2 and 0 respectively. Since `box1.print()` does not modify any attribute, the value of the attribute `cookies` will be equal to 0 after this method is called.

3. The object `sock1` is created with a capacity of 4, since the `SweetsSock(unsigned c)` constructor calls `SweetsBox(c/2)`. The `sock1.refill()` operation then assigns the values 4 and 2 to the `candies` and `coal` attributes respectively. Therefore, the correct answer is 2.

4. Similarly to the behaviour of the base class `SweetsBox` described in answer 2 above, the `sock1.print()` method call does not modify any attribute in `sock1`, so the value of `sock1.candies` will be determined by the execution of `sock1.eatContent(3)` in the previous line. This method decreases by 1 the value of `coal`, and by 3 the value of `candies`, which was previously 4. As a consequence, the correct answer is 1.

5. The first execution of `changeAndPrint()` is performed on `box2`, which members are called in the same sequence described in previous answers. The execution of `b.eatContent(1)` consumes one candy and two cookies from a `SweetsBox` object with a capacity of 32, therefore the returned value is 29.

6. The second execution of `changeAndPrint()` is performed on the object `sock2`, which is passed as reference to a `SweetsBox` object. Therefore, the call to the method `b.refill()` will result in an execution of `SweetsBox::refill()` on `sock2`. Since `sock2` has been created with a capacity of 16, this will set both `candies` and `cookies` parameters to 8, while `coal` will remain at its default value (0). Next, `b.eatContent(1)` will result in the execution of `SweetsSock::eatContent()` since this method is `virtual` in the base class. As a consequence, this time the returned value will be 0.