Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna, Prof. Matteo Giovanni Rossi

Date: 10-02-2025

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

<u>Exam duration: 2 hours and 30 minutes</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 (up to 13 points)

13 Points if you answer questions 1–4+6
10 Points if you answer questions 1–4+5

We want to realize a system for the acquisition of sequences of data (time series) of different kinds—for example, sequences of temperature readings from a sensor. We want the system to be general enough to handle different kinds of data, and also different strategies for the handling of the data (for example, for certain series it is crucial to handle the readings in strictly sequential order, while for other series data might be read out of order).

Answer the following questions.

1. Define a class `Reading`, which captures the general notion of reading.
   A reading is characterized by a timestamp and by the value that has been read. The timestamp is a value that corresponds to the number of milliseconds elapsed from the first of January 1970 until the time of the reading (i.e., 0 is the midnight of January 1st, 1970, 1 is one millisecond after that, and so on). The reading value is a real number.
   Once created, a reading cannot be modified.
   Class `Reading` has a single constructor, which initializes the timestamp and the value.
   Users of the class must also be able to retrieve the timestamp and the value of the reading.
   A reading can be valid or not. The class provides a function that returns `true` if the reading is valid, `false` otherwise. By default, a reading is valid (though for some kinds of reading the value must obey certain constraints, see the next 2 questions).

2. Define a class `TemperatureReading` that captures temperature readings (measured in Celsius degrees). To be valid, a temperature reading must be greater than or equal to -273 degrees.

3. Define a class `StockVariationReading` that captures readings concerning the variations (in percentage) of the value of a stock. Notice that a variation can be negative. We represent variations (in percentage) as real values between -1 and 1 (i.e., the value of a stock cannot increase or decrease by more than 100%).

4. Define a class `ReadingCollection`, which represents a collection of readings. Depending on the readings stored in the collection, the nature of the collection can vary. The only feature in common to all collections is that they offer an operation, `average`, which returns the average of the values of the readings stored in the collection.

Answer one (**and only one**) of the following two questions. Notice that the maximum total points of the exercise are different if you develop point 5 rather than point 6. In each case, in developing your solution, you should take into consideration the efficiency of operation `addReading`.

5. Define a class, TemperatureCollection, which is a collection of temperature readings. Note that readings could be shared among collections. Also, in the collection there must be a single reading per timestamp.
   The class provides an operation, addReading, which adds a new reading to the collection, and returns true if the addition is successful, false otherwise; temperatures **must** be added in order, according to their timestamps, otherwise the addition is not successful. The class also provides a method, lastNvalues, that takes as input a positive integer n, and returns the last n elements added to the collection (it returns all the elements in the collection if it contains less than n readings). Declare all elements of the class, and define operations addReading and lastNvalues.

6. Define a class, StockVariationCollection, which is a collection of stock variations. Note that readings can be shared among collections. Also, in the collection there must be a single reading per timestamp.
   The class provides an operation, addReading, which adds a new reading to the collection, and returns true if the addition is successful, false otherwise; stock variations can be added out of order.
   The class also provides a method, maximumSubsequence, that returns the sum of the readings of the subsequence of the collection whose sum of the values is maximum. For example, if the readings are (in order of timestamp, which is not reported here for simplicity) $[0.04, 0.1, -0.2, 0.05, -0.02, 0.08, 0.07, 0.08, -0.02, -0.03]$, the sum of the maximum sequence is 0.26, which is the total of the subsequence $[0.05, -0.02, 0.08, 0.07, 0.08]$. If the collection is empty, maximumSubsequence returns 0.
   Declare all elements of the class, and define operations addReading and maximumSubsequence. What is the worst case complexity of your implementation of operation maximumSubsequence?

   NB: The implementation of operation maximumSubsequence does not need to be optimal from the point of view of its complexity.

Solution 1

1. Declaration of class Reading:

```
class Reading {
public:
    Reading(double v, unsigned ts) : value(v), timestamp(ts) {}
    virtual bool is_valid() const { return true; }

    double get_value() const { return value; }
    unsigned get_timestamp() const { return timestamp; }

protected:
    const double value;
    const unsigned timestamp;

};
```

   Since a reading, after being created, cannot be modified, the fields of the class are all const. Hence, they must be both initialized at construction time. They are declared with visibility protected to allow derived classes to directly access them. Notice that an object of type Reading behaves in a similar manner as a base type (such as int or char): it can be exchanged (copied, passed as parameter, etc.) freely, as if it were a simple value, because it is unmutable.
   The getter methods, get_value and get_timestamp have been introduced to allow classe other than derived ones to read the values of the fields. The behavior of method is_valid depends on the nature of the reading, so its definition will be overridden by derived classes, hence it is declared virtual. Without further information about its nature, the reading is considered to be valid, hence the definition of is_valid in the base class of the hierarchy of readings simply returns true.

2. Declaration of class TemperatureReading:

```
class TemperatureReading : public Reading {
public:
    TemperatureReading(double v, unsigned ts) : Reading(v, ts) {}
    bool is_valid() const override { return value > -273; }
};
```

A TemperatureReading is a special case of Reading, hence it inherits from the latter. The only difference between a Reading and a TemperatureReading is that a temperature must be greater than -273 to be valid, hence method is_valid overrides the one defined in the base class and is defined accordingly.

3. Declaration of class StockVariationReading:

```
class StockVariationReading : public Reading {
public:
    StockVariationReading(double v, unsigned ts) : Reading(v, ts) {}
    bool is_valid() const override { return -1 <= value && value <= 1; }
};
```

Similarly to TemperatureReading, StockVariationReading inherits from Reading and overrides the definition of is_valid; in this case, a StockVariationReading is valid when is belong to interval $[-1, 1]$.

4. Declaration of class ReadingCollection:

```
class ReadingCollection {
public:
    virtual double average() const = 0;
};
```

We do not know anything about a ReadingCollection except that it provides a method, average. However, since we do not have any information about the intended behavior of a collection, we declare avergae as a *pure virtual* function.
Notice that ReadingCollection is, essentially, what is known as an *interface*: it provides the signature of operations (in this case, average), but nothing else. Any class implementing ReadingCollection will have to provide not only the definition of method average, but also the definition of the attributes.

5. Declaration of class TemperatureCollection:

```
class TemperatureCollection : public ReadingCollection {
private:
    std::vector<std::shared_ptr<TemperatureReading>> readings;
public:
    bool addReading(const TemperatureReading &tr) {
        if (readings.empty() || readings.back()->get_timestamp() < tr.get_timestamp()) {
            readings.push_back(std::make_shared<TemperatureReading>(tr));
            return true;
        } else
            return false;
    };

    std::vector<std::shared_ptr<TemperatureReading>> lastNvalues(unsigned n)  const{
        std::vector<std::shared_ptr<TemperatureReading>> res;
        if (readings.size() <= n)
            res = readings;
        else {
            for (unsigned i = readings.size() - n; i < readings.size(); i++) {
                res.push_back(readings[i]);
            }
        }
        return res;
    }

    double average() const override;

};
```

Class TemperatureCollection implements interface ReadingCollection. As such, it defines how the collection is actually stored in memory. In the case of a collection of temperature readings, we know that, by construction, data must be inserted in the collection in strictly sequential order. Hence, we need a data structure for which appending element at the end of the sequence is efficient. For this reason, a TemperatureCollection relies on a vector to store the readings. Indeed, the push_back operation is, on average, very efficient for vectors—except when a reallocation of the sequence is necessary, which, however typically occurs infrequently. In addition, since we know that readings are shared among instances of the class, to avoid wasting memory we store them on the *heap*, and we use *shared pointers* (rather than regular pointers) so that we do not need to concern ourselves with the details of the handling of the allocated memory (in particular, its deallocation). Notice that since readings are immutable by construction, the use of shared pointers to access the data does not pose any concerns regarding side effects and unwanted uncontrolled changes, which are simply impossible.

The addReading method makes sure that readings are inserted in ascending order of timestamp: if the timestamp of the element to be added is not greater than the biggest timestamp appearing in the collection (which, by construction, is the one of the last reading of the collection), the element is not added (and false is returned). The argument of function addREading is declared as a reference to const for clarity's sake; however, the object is by construction immutable, so it cannot be modified anyway; in addition, the object is a small one (it includes only 2 attributes, both of simple types) so, according to the guidelines seen in class, it could have been passed also by value, rather than by reference.
Function lastNvalues simply copies the pointers to the last N elements of the vector in a new vector, and returns the latter.

6. Declaration of class StockVariationCollection:

```
class StockVariationCollection : public ReadingCollection {
private:
   std::map<unsigned, std::shared_ptr<StockVariationReading>> readings;
public:
   bool addReading(const StockVariationReading &svr) {
      auto res = readings.insert(std::pair<unsigned, std::shared_ptr<StockVariationReading>>(svr.
    get_timestamp(), std::make_shared<StockVariationReading>(svr)));
      return res.second;
   };

   // This is a very simple implementation of the function.
   // It performs 2 nested loops, and in this way it checks, one by one, all
   // subsequences of the collection (if the sequence is made of elements [e1, e2, e3], for example,
   // it checks [e1], then [e1, e2], then [e1, e2, e3], then [e2], then [e2, e3], then [e3]
   // for each sequence it computes the sum of its elements, and it keeps track of the
   // maximum sum.
   double maximumSubsequence()  const{
      if (readings.empty())
         return 0;
      double max = readings.cbegin()->second->get_value();
      for (auto it = readings.cbegin(); it != readings.end(); it++) {
         double sum = it->second->get_value();
         if (sum > max)
            max = sum;

         auto it2 = it;
         for (it2++; it2 != readings.end(); it2++) {
            sum += it2->second->get_value();
            if (sum > max)
               max = sum;
         }
      }
      return max;
   }

   double average() const override;

};
```

In the case of StockVariationReadingCollection, since we know that elements can be inserted out of order of timestamp, a vector is no longer the best solution. In fact, the ordering of the elements according to their timestamps is a crucial feature of the collection (for example, to compute the maximum subsequence); in principle, one could store them out of order and then sort them whenever needed, but a better solution is just to store them in order, al long as the storing is efficient. Since inserting an element in order in a sequence typically requires shifting $O(n)$ elements of the sequence (with n the number of stored elements), using a vector is not the best solution. An ordered map (where the key is simply the timestamp of the reading), instead, fulfills all our needs. As in the case of a TemperatureCollection, we store the readings on the heap and use shared pointers to ease memory management.

An ordered map already guarantees that stored keys are all distinct, so method addReading simply needs to return the value returned by insert (more precisely, the second element of the pair returned by insert). The argument of function addREading is declared as a reference to const for clarity's sake; however, the object is by construction immutable, so it cannot be modified anyway; in addition, the object is a small one (it includes only 2 attributes, both of simple types) so, according to the guidelines seen in class, it could have been passed also by value, rather than by reference.

As mentioned in the comments before method maximumSubsequence, the function essentially goes through the subsequences of the collection one by one (notice that, since we are using a map indexed on the timestamps, the iterators return the elements of the collection in order of timestamp).
The complexity of the implementation of function maximumSubsequence is $O(n^2)$, with n number of elements in the collection. Indeed, the i-th execution of the internal for loop (with $i = 1, 2, \ldots, n$) performs $n - i + 1$ iterations, so the total number of subsequences checked is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, which is $O(n^2)$ (notice that the check of each subsequence, which is performed by lines 27-29, takes constant time).

Notice that function maximumSubsequence can be implemented in a much more efficient way. Indeed, the following implementation has, as the name suggests, linear complexity (i.e., it has complexity $O(n)$):

```
// This alternative implementation performs a single pass, so it has linear complexity.
// More precisely, it keeps track of the current sum of the elements.
// The moment when the sum becomes negative it means that, if we continue adding
// values to the sum, in any case the single value of the next element will
// be greater than the addition of the current sum with the next element, hence
// we need to make sure that we "restart" computing the sum; for this reason variable sum
// is reset to zero, because when the next element is added to sum, the new value
// of sum will simply be the value of the next element.
double maximumSubsequence_linear() const{
   if (readings.empty())
      return 0;
   auto it = readings.cbegin();
   double sum = it->second->get_value();
   double max = sum;
   for (it++; it != readings.end(); it++) {
      sum += it->second->get_value();
      if (sum > max)
         max = sum;
      if (sum < 0)
         sum = 0;
   }
   return max;
}
```

It is clear that maximumSubsequence_linear has linear complexity, since it performs a single loop through the elements of the collection.

Exercise 2 (13 points)

You are required to develop a parallel program that determines whether a matrix of integers is diagonal. The input matrix is stored in a file (whose name is provided as a single command-line argument) and is accessible only by the process of rank 0. Ensure that you manage command-line arguments appropriately, raising errors as necessary. We want to exploit a parallel architecture to realize the program:

1. Describe in your own words what strategy you want to use to paralellize the computation (i.e., how you distribute data across processes, what computation each process performs, how you generate the results).

2. Write suitable MPI code that realizes the program above.

   Note 1: you may assume that the input is a square matrix (no input size checking is required). Additionally, the number of rows (columns) is divisible by the number of available processes.

   Note 2: you can rely on the implementation of the `la::dense_matrix` class, whose declaration is provided below. This class can store integer values and is essentially the same class we used during exercise sessions for managing matrices of doubles, with which you should be familiar.

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<int> 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 = 0, m_columns = 0;
    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

1. To compute the desired result in a parallel fashion, we can proceed in the following way:
   - (i) We first determine if a single program argument is provided at the command line and accordingly generate a message.
   - (ii) If the input matrix file name is provided correctly, rank 0 reads the file and shards the matrix by row through a *scatter* operation.
   - (iii) Each process verifies that its matrix portion satisfies the diagonal condition (all non-diagonal elements are zero).
   - (iv) The global result is obtained by employing a *reduce* operation that computes the logical AND across all processes.
   - (v) Finally, rank 0 outputs the result.
2. A possible piece of code implementing the mechanisms above is the following:

```
#include <iostream>
#include <string>
#include <fstream>
#include <mpi.h>
#include "dense_matrix.hpp"

bool isDiagonalMatrix(const la::dense_matrix& matrix);

```

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    unsigned n; // Size of the matrix
    la::dense_matrix matrix;

    // (i)
    // Check arguments number is correct and print instructions for
    // program use
    if (argc != 2) {
      if (rank == 0)
        std::cerr << "Usage:" << argv[0] << "<filename>"<<std::endl;
      MPI_Finalize();
      return -1;
    }

    // (ii-a)
    // Argument number is ok. Let's read the matrix from file and share
    // across all ranks
    if (rank == 0) {
      std::ifstream ifs(argv[1]);
      matrix.read(ifs);
      ifs.close();
      // Print matrix
      std::cout << "Input matrix:" << std::endl;
      matrix.print(std::cout);
    }

    // (ii-b)
    n = matrix.rows();
    // Broadcast the size of the matrix to all processes
    MPI_Bcast(&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

    // Calculate the number of rows each process will handle
    unsigned local_n = n / size;
    la::dense_matrix sub_matrix(local_n,n,0);


    // Scatter the matrix to all processes
    MPI_Scatter(matrix.data(), local_n * n, MPI_INT, sub_matrix.data(),
                local_n * n, MPI_INT, 0, MPI_COMM_WORLD);

    // (iii)
    // Each process checks its part of the matrix
    unsigned local_result = isDiagonalMatrix(sub_matrix);
    std::cout << "local_result, rank: " << rank << " " << local_result << std::endl;

    // (iv)
    if (rank == 0)
      MPI_Reduce(MPI_IN_PLACE, &local_result, 1, MPI_UNSIGNED,  MPI_LAND, 0,
      MPI_COMM_WORLD);
    else
      MPI_Reduce(&local_result, nullptr , 1, MPI_UNSIGNED,  MPI_LAND, 0,
      MPI_COMM_WORLD);

65     // (v)
66     // Rank 0 print the result
67     if (rank == 0) {
68         if (local_result)
69           std::cout << "The input matrix is a diagonal matrix" << std::endl;
70         else
71           std::cout << "The input matrix is NOT a diagonal matrix" << std::endl;
72     }
73 
74 
75     MPI_Finalize();
76     return 0;
77 }
78 
79 
80 
81 
82 
83 bool isDiagonalMatrix(const la::dense_matrix& matrix) {
84     int rank;
85     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
86     // Check for diagonal properties considering an index shift
87     for (int i = 0; i < matrix.rows(); ++i) {
88         for (int j = 0; j < matrix.columns(); ++j) {
89             if (i + rank * matrix.rows() != j && matrix(i,j) != 0) {
90                 return false; // Found a non-zero element outside the diagonal
91             }
92         }
93     }
94 
95     return true; // All non-diagonal elements are zero
96 }
```

Through the MPI_Init function, the MPI environment is initialized. Subsequently, after performing argument verification, rank 0 reads the input file and broadcasts the matrix size (n) to all processes using the MPI_Bcast function. Rows are distributed across all processes using the MPI_Scatter function. In this manner, each process checks, through the isDiagonalMatrix function, whether its matrix portion satisfies the diagonal condition. This function determines if a given matrix is diagonal by iterating over each element and checking if non-diagonal elements are zero. It employs a rank-based index shift $(i + rank * local\_n)$ to access elements.

The final result is computed by an MPI_Reduce logical AND (MPI_LAND) operation. The MPI_IN_PLACE option enables the root process (rank 0) to utilize the same buffer for both sending and receiving during reduction. Alternatively, despite requiring additional communication, all ranks can execute MPI_Allreduce(MPI_IN_PLACE, &local_result, 1, MPI_UNSIGNED, MPI_LAND, MPI_COMM_WORLD) to obtain the result, which is equivalent to lines 59—-62.

Exercise 3 (4 points)

Consider the following piece of code:

```
#include <iostream>

class BinaryIntOp {
public:
    virtual int calc(int v1, int v2) = 0;
};

class Add : public BinaryIntOp {
public:
    int calc(int v1, int v2) override {
        return v1 + v2;
    }
};

class Sub : public BinaryIntOp {
public:
    int calc(int v1, int v2) override {
        return v1 - v2;
    }
};

class Mult : public BinaryIntOp {
public:
    int calc(int v1, int v2) override {
        return v1 * v2;
    }
};

class Div : public BinaryIntOp {
public:
    int calc(int v1, int v2) override {
        return v1 / v2;
    }
};

class OppAdd : public Add {
public:
    int calc(int v1, int v2) override {
        return -(Add::calc(v1, v2));
    }
};

class AbsMult : public Mult {
public:
    int calc(int v1, int v2) override {
        return abs(Mult::calc(v1, v2));
    }
};

```

```
50 int main() {
51
52     BinaryIntOp *op1;
53     Add op2;
54     Mult op3;
55     Div op4;
56     OppAdd op5;
57     AbsMult op6;
58
59     op1 = &op2;
60     int res1 = op1->calc(op2.calc(op3.calc(-2, 6), op4.calc(3, 7)), 1);
61
62     std::cout << res1 << "\n";
63
64     Mult &op7 = op6;
65     op1 = new Sub();
66     int res2 = op1->calc(op5.calc(-4, op7.calc(5,op4.calc(-4, 2))), op6.calc(-10, 2));
67
68     std::cout << res2 << "\n";
69
70     op2 = op5;
71     op1 = &op5;
72     int res3 = op2.calc(op1->calc(3, 7), op7.calc(-1, op2.calc(0,(new Sub())->calc(0, 8))));
73
74     std::cout << res3 << "\n";
75
76     op1 = &op7;
77     Mult &op8 = op3;
78     int res4 = op1->calc(2, (new Sub)->calc(op3.calc(10, op8.calc(5, -2)), 10));
79
80     std::cout << res4 << "\n";
81
82     return 0;
83 }
```

What numbers are printed at the following lines?

(a) Line 62;

(b) Line 68;

(c) Line 74;

(d) Line 80.

Solution 3

TBC

(a) **-11**; indeed, at line 60, where the result is computed, op1 is of type Add, op2 is of type Add, op3 is of type Mult, and op4 is of type Div; hence, the computation amounts to: $((-2*6)+(3/7))+1=-11$ (recall that Div is the division among integers, so $3/7=0$).

(b) **-26**; at line 66 op1 is of type Sub, op5 is of type OppAdd, op7 is of type AbsMult, op4 is of type Div op6 is of type AbsMult; hence, the computation amounts to: $-(-4+|5*(-4/2)|)-|-10*2|=-26$.

(c) **-2**; at line 72 op2 is of type Add, op1 is of type OppAdd, and op7 is of type AbsMult; hence, the computation amounts to: $-(3+7)+|-1*(0+(0-8))|=-2$.

(d) **220**; at line 78 op1 is of type AbsMult, op3 is of type Mult, and op8 is of type Mult; hence, the computation amounts to: $|2*((10*(5*-2))-10)|=220$.