FINAL PROJECT

Parallel QR Code Generator with OpenMP and MPI

Juan José Betancourt
Nicolás Delgado
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

Jun 10, 2021

Contents

1 The Code to Parallelize

1.1 Algorithm Code

In the following subsections you can see the main functions of the application to be worked on.

Make Segments

```
vector<QrSegment> QrSegment::makeSegments(const char *text) {
  // Select the most efficient segment encoding automatically
  vector<QrSegment> result;
  if (*text == '\0');  // Leave result empty
  else if (isNumeric(text))
    result.push_back(makeNumeric(text));
  else if (isAlphanumeric(text))
    result.push_back(makeAlphanumeric(text));
  else {
    vector<uint8_t> bytes;
    for (; *text != '\0'; text++)
      bytes.push_back(static_cast<uint8_t>(*text));
    result.push_back(makeBytes(bytes));
  }
  return result;
}
```

Encode Text

```
QrCode QrCode::encodeText(const char *text, Ecc ecl) {
  vector<QrSegment> segs = QrSegment::makeSegments(text);
  return encodeSegments(segs, ecl);
}


QrCode QrCode::encodeBinary(const vector<uint8_t> &data, Ecc ecl) {
  vector<QrSegment> segs{QrSegment::makeBytes(data)};
  return encodeSegments(segs, ecl);
}
```

The encode method that the before functions are calling is the following:

```
1
2  QrCode QrCode::encodeSegments(const vector<QrSegment> &segs, Ecc ecl,
3      int minVersion, int maxVersion, int mask, bool boostEcl) {
4    if (!(MIN_VERSION <= minVersion && minVersion <= maxVersion && maxVersion
       <= MAX_VERSION) || mask < -1 || mask > 7)
5      throw std::invalid_argument("Invalid value");
6
7    //Find the minimal version number to use
8    int version, dataUsedBits;
9    for (version = minVersion; ; version++) {
10     int dataCapacityBits = getNumDataCodewords(version, ecl) * 8;  // Number of data
         bits available
11     dataUsedBits = QrSegment::getTotalBits(segs, version);
12     if (dataUsedBits != -1 && dataUsedBits <= dataCapacityBits)
13       break;   // This version number is found to be suitable
14     if (version >= maxVersion) {  // All versions in the range could not fit the given
         data
15       std::ostringstream sb;
16       if (dataUsedBits == -1)
17         sb << "Segment too long";
18       else {
19         sb << "Data length = " << dataUsedBits << " bits, ";
20         sb << "Max capacity = " << dataCapacityBits << " bits";
21       }
22       throw data_too_long(sb.str());
23     }
24   }
25   if (dataUsedBits == -1)
26     throw std::logic_error("Assertion error");
27
28   //Increase the error correction level while the data still fits in the current version
       number
29   for (Ecc newEcl : vector<Ecc>{Ecc::MEDIUM, Ecc::QUARTILE, Ecc::HIGH}) {  //
       From low to high
30     if (boostEcl && dataUsedBits <= getNumDataCodewords(version, newEcl) * 8)
31       ecl = newEcl;
32   }
33
34   // Concatenate all segments to create the data bit string
35   BitBuffer bb;
36   for (const QrSegment &seg : segs) {
37     bb.appendBits(static_cast<uint32_t>(seg.getMode().getModeBits()), 4);
38     bb.appendBits(static_cast<uint32_t>(seg.getNumChars()), seg.getMode().
         numCharCountBits(version));
39     bb.insert(bb.end(), seg.getData().begin(), seg.getData().end());
40   }
41   if (bb.size() != static_cast<unsigned int>(dataUsedBits))
42     throw std::logic_error("Assertion error");
43
44   // Add terminator and pad up to a byte if applicable
45   size_t dataCapacityBits = static_cast<size_t>(getNumDataCodewords(version, ecl)) * 8;
46   if (bb.size() > dataCapacityBits)
47     throw std::logic_error("Assertion error");
48   bb.appendBits(0, std::min(4, static_cast<int>(dataCapacityBits - bb.size())));
49   bb.appendBits(0, (8 - static_cast<int>(bb.size() % 8)) % 8);
50   if (bb.size() % 8 != 0)
51     throw std::logic_error("Assertion error");
```

```
52
53    // Pad with alternating bytes until data capacity is reached
54    for (uint8_t padByte = 0xEC; bb.size() < dataCapacityBits; padByte ^= 0xEC ^ 0x11)
55      bb.appendBits(padByte, 8);
56
57    // Pack bits into bytes in big endian
58    vector<uint8_t> dataCodewords(bb.size() / 8);
59    for (size_t i = 0; i < bb.size(); i++)
60      dataCodewords[i >> 3] |= (bb.at(i) ? 1 : 0) << (7 - (i & 7));
61
62    // Create the QR Code object
63    return QrCode(version, ecl, dataCodewords, mask);
64  }
```

Create QR

```
QrCode::QrCode(int ver, Ecc ecl, const vector<uint8_t> &dataCodewords, int msk) :
    // Initialize fields and check arguments
    version(ver),
    errorCorrectionLevel(ecl) {
  if (ver < MIN_VERSION || ver > MAX_VERSION)
    throw std::domain_error("Version value out of range");
  if (msk < -1 || msk > 7)
    throw std::domain_error("Mask value out of range");
  size = ver * 4 + 17;
  size_t sz = static_cast<size_t>(size);
  modules = vector<vector<bool> >(sz, vector<bool>(sz));  // Initially all white
  isFunction = vector<vector<bool> >(sz, vector<bool>(sz));

  // Compute ECC, draw modules
  drawFunctionPatterns();
  const vector<uint8_t> allCodewords = addEccAndInterleave(dataCodewords);
  drawCodewords(allCodewords);

  // Do masking
  if (msk == -1) {  // Automatically choose best mask
    long minPenalty = LONG_MAX;
    for (int i = 0; i < 8; i++) {
      applyMask(i);
      drawFormatBits(i);
      long penalty = getPenaltyScore();
      if (penalty < minPenalty) {
        msk = i;
        minPenalty = penalty;
      }
      applyMask(i);  // Undoes the mask due to XOR
    }
  }
  if (msk < 0 || msk > 7)
    throw std::logic_error("Assertion error");
  this->mask = msk;
  applyMask(msk);  // Apply the final choice of mask
  drawFormatBits(msk);  // Overwrite old format bits

  isFunction.clear();
  isFunction.shrink_to_fit();
}
```

1.2 Algorithm

The project is a QR code generator, in different languages, such as: C, C++, Python, Rust, and JavaScript. It has different ways to generate the QR code, either by using numeric characters, or by creating in characters the code to create a SVG/XML (except for C code). In the Parallel Programming project, the focus will be on the code made in C++, where a `namespace` is created, to work with the QR creator, where we could find different parts that can potentially be parallelizable.

Input:

- ***text:*** is the text string that will be encoded for the QR code. Its maximum length must be 2600 characters, since for larger values a Segmentation Fault error occurs.
- ***n:*** is the length of the *text* or the number of characters.

Output:

- ***svgString:*** is the text string describing the SVG image of the QR code.

Description:

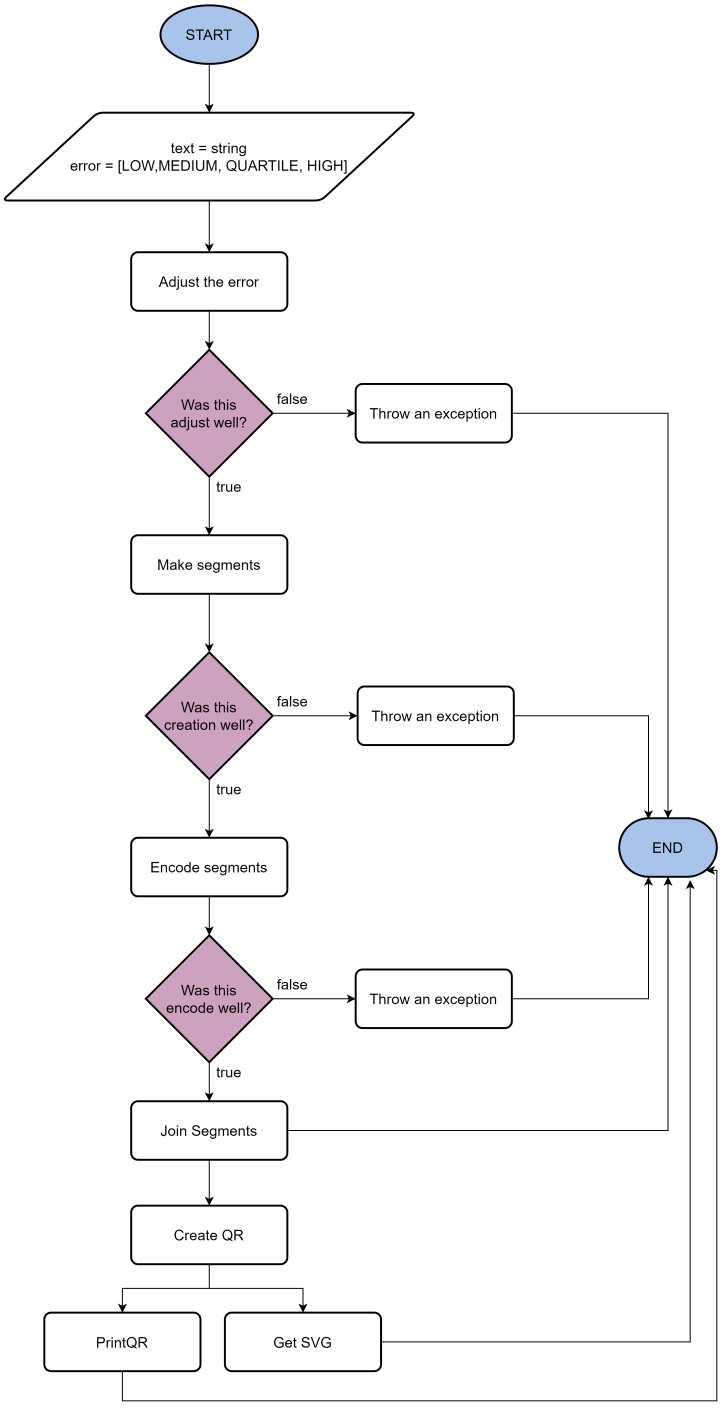
The algorithm basically what it does is, based on a size for the error (which can be corrected as it increases) and a text, to see if the text fits in the respective error, and adjusts it. After this, it goes to the encoding process where the segments are created according to the type of values that were entered according to whether it has alphanumeric or numeric values and the size of the text that was sent. Once it has been transformed, each segment is encoded, where it is checked if the size is adequate and each segment is joined together.

If there is no error when joining the segments it creates the QR code and after that you can use the method to print the QR code found in the demo or we can also create a file in SVG format with functional QR code.

This creation of the QR code in an SVG file is done as follows: it is sent to have 4 edges, the format of an SVG/XML file is copied and in the path we proceed to go transforming module by module to what goes in the path of the SVG files and so on until there are no more modules, this could be sent as an output to a single file with .svg format, to obtain the desired file or simply printed in console what goes inside the SVG file. There is also the option to print in console a representation of the QR code with # where black cells would go in the SVG.

Flowchart:

The operation of the algorithm can be better understood in the following flowchart:


START
text = string
error = [LOW,MEDIUM, QUARTILE, HIGH]
Adjust the error
Was this adjust well?
false
Throw an exception
true
Make segments
Was this creation well?
false
Throw an exception
true
Encode segments
Was this encode well?
false
Throw an exception
true
Join Segments
Create QR
PrintQR
Get SVG
END

1.3 Complexity

In some functions we work with a matrix going through the whole array, in other functions we work just going through an array. But when we just want to work with a QR Code and to generate a SVG image proving with different sizes it have a lineal behavior. As it is shown in the time vs size graphic using `wtime`.

Make Segment

In this case it work with a conditional so this function can be $O(1)$ or $O(n)$ if we work with last option, with n as the bit size of the string.

Encode Text

It call `makeSegment` so at the worst case it is $O(n)$ or in better case it is $O(1)$.

Encode Segments

The first for cycle work with at much 40 steps, $O(n)$ with n being the version size, but it has a very small amount of steps. It has more $O(1)$ operations and then a for of $O(4)$ that is $O(1)$ After that it go through all the segments in a for. And another for that go through all the bytes, that are 325 bytes in the worst case. And then a last for that go through the buffer. We then work with a $O(n)$ for the worst case.

Create QR

The function have a lot of one step operations and a for for all the string. It means $O(n)$ and it call two function that works with operations of $O(n)$ and $O(n-7)$ but it also work with a function that go through a matrix and it is $O(n^2)$ at the end is an $O(n^2)$ function but it works with a very small portion of data.

Conclusion

The main program works with a $O(n) + O(n) + O(n) + O(n^2)$ that it is equal to $O(n^2)$, in theoretical terms.

1.4 Time Analysis

The experimental environment where the tests were taken are as follows:

Configuration	Value
System Version	Ubuntu 18.04.5
Compiler	G++ 7.5.0 & mpiCC
CPU	Intel Core i5-7200U 2.50GHz x4
GPU	Intel HD Graphics 620
Memory Capacity	11,5 GiB
OpenMP Version	OpenMP 4.5
MPI Version	MPICH 3.4.1

Configuration (CPU)	Value
Model Name	Intel(R) Core(TM) i5-7200U @ 2.50GHz
Architecture	x86_64
CPU(s)	4
Thread(s) per Core	2
Core(s) per Socket	2
Socket(s)	1
CPU Max MHz	3100,0000
CPU Min MHz	400,0000
BogoMIPS	5399.81
Virtualization	VT-x
L1d Cache	32K
L1i Cache	32K
L2 Cache	256K
L3 Cache	3072K

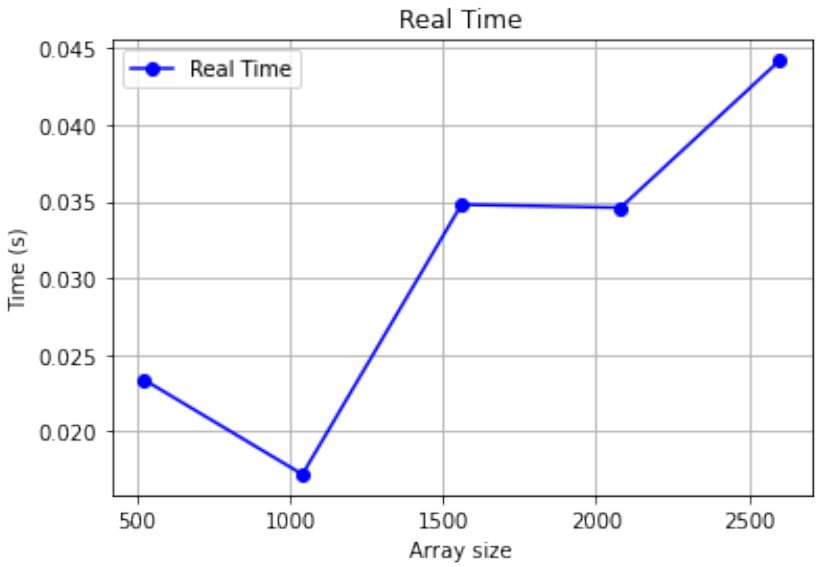
Configuration (RAM)	Value (RAM 1)	Value (RAM 2)
Total Width	64 bits	64 bits
Data Width	64 bits	64 bits
Size	4GB	8GB
Type	DDR4	DDR4
Type Detail	Sync. Unbuffered	Sync. Unbuffered
Speed	2133 MT/s	2133 MT/s
Manufacturer	Samsung	Kingston
Configured Clock Speed	2133 MT/s	2133 MT/s

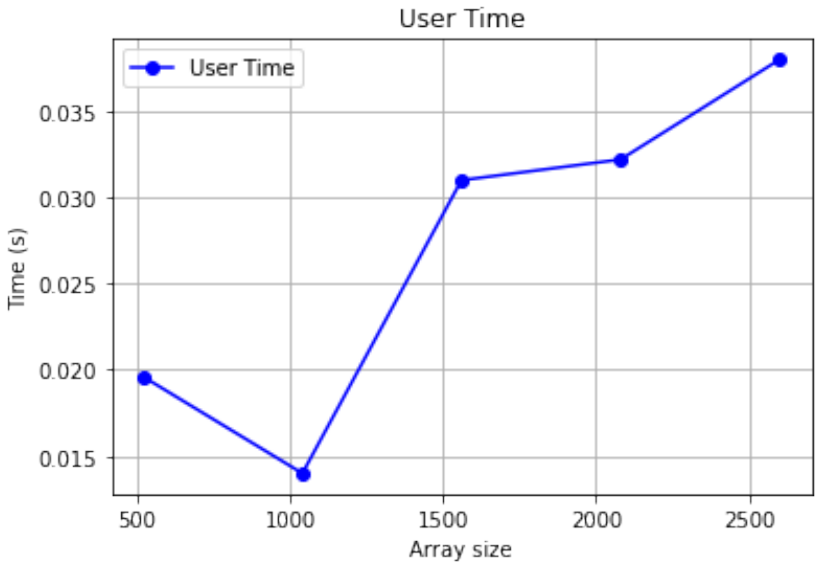
Configuration (Disk)	Value
Model Name	ST2000LM007
Capacity	2 TB
Width	32 bits
Speed	66MHz
Buffer size	128 MB
Average Seek Time	13 ms
Data Transfer Rate	600 MBps
Internal Data Rate	100 MBps

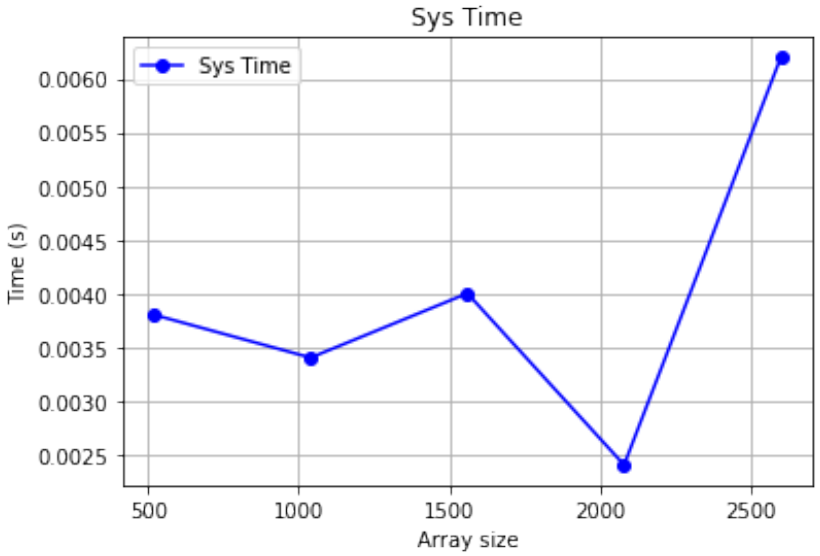
Measuring the execution time with the `time` tool for different values of `n`, the following result was obtained:

n	Real $\bar{e}$	Real σ_e	User $\bar{e}$	User σ_e	Sys $\bar{e}$	Sys σ_e
520	0.023	0.007	0.020	0.005	0.004	0.002
1040	0.017	0.009	0.014	0.007	0.003	0.003
1560	0.035	0.011	0.031	0.011	0.004	0.003
2080	0.035	0.010	0.032	0.009	0.002	0.002
2600	0.044	0.008	0.038	0.006	0.006	0.006

The time averages can best be seen in the following graphs:



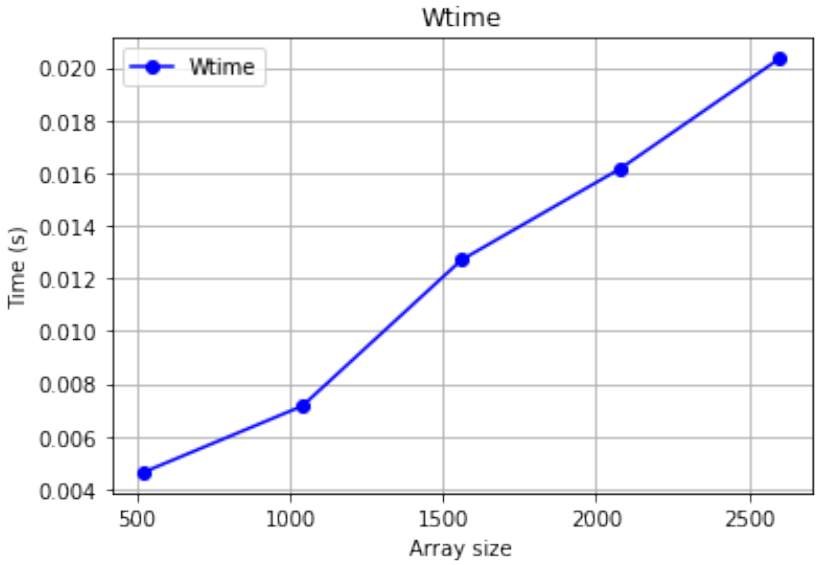

User Time
User Time
0.035
0.030
0.025
0.020
0.015
Time (s)
500
1000
1500
2000
2500
Array size


Sys Time
Sys Time
0.0060
0.0055
0.0050
0.0045
0.0040
0.0035
0.0030
0.0025
Time (s)
500
1000
1500
2000
2500
Array size

Using the `wtime` tool to measure the execution time of the `doBasicDemo()` function at the same input sizes, the following results are obtained:

n	**Wtime** $\bar{e}$	**Wtime** σ_e
520	0.00463	0.00040
1040	0.00715	0.00157
1560	0.01270	0.00126
2080	0.01617	0.00236
2600	0.02035	0.00156

The average times can best be seen in the following graph:



The application was analyzed using the tool `gprof`. Because the result was too long to fit in the document, only a part of it is shown, its full version can be seen in the file `gprof_analysis.txt` in the folder where this report is located.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.01      0.01     0.01    81990     0.00     0.00  qrcodegen::QrCode::reedSolomonMultiply(unsi
 50.01      0.02     0.01        8     1.25     1.25  qrcodegen::QrCode::getPenaltyScore() const
  0.00      0.02     0.00  1337072     0.00     0.00  qrcodegen::QrCode::module(int, int) const
  0.00      0.02     0.00   237738     0.00     0.00  qrcodegen::QrCode::finderPenaltyAddHistory(
  0.00      0.02     0.00   120221     0.00     0.00  qrcodegen::QrCode::finderPenaltyCountPatter
  0.00      0.02     0.00    28561     0.00     0.00  qrcodegen::QrCode::getModule(int, int) cons
  0.00      0.02     0.00     5307     0.00     0.00  void std::vector<unsigned char, std::alloca
  0.00      0.02     0.00     2704     0.00     0.00  qrcodegen::QrCode::finderPenaltyTerminateAn
  0.00      0.02     0.00     2703     0.00     0.00  qrcodegen::BitBuffer::appendBits(unsigned i
  0.00      0.02     0.00     2026     0.00     0.00  qrcodegen::QrCode::setFunctionModule(int, i
  0.00      0.02     0.00       46     0.00     0.00  qrcodegen::QrCode::drawAlignmentPattern(int
  0.00      0.02     0.00       45     0.00     0.00  qrcodegen::QrCode::getNumRawDataModules(int
  0.00      0.02     0.00       43     0.00     0.00  qrcodegen::QrCode::getNumDataCodewords(int,
  0.00      0.02     0.00       38     0.00     0.00  qrcodegen::QrSegment::getTotalBits(std::vec
  0.00      0.02     0.00       22     0.00     0.45  qrcodegen::QrCode::reedSolomonComputeRemain
  0.00      0.02     0.00       22     0.00     0.00  void std::vector<unsigned char, std::alloca
  0.00      0.02     0.00       17     0.00     0.00  qrcodegen::QrCode::applyMask(int)
  0.00      0.02     0.00       12     0.00     0.00  std::vector<bool, std::allocator<bool> >::_
  0.00      0.02     0.00       10     0.00     0.00  qrcodegen::QrCode::getFormatBits(qrcodegen:
  0.00      0.02     0.00       10     0.00     0.00  qrcodegen::QrCode::drawFormatBits(int)
  0.00      0.02     0.00        6     0.00     0.00  void std::vector<std::vector<unsigned char,
  0.00      0.02     0.00        4     0.00     0.00  void std::vector<int, std::allocator<int> >
  0.00      0.02     0.00        3     0.00     0.00  qrcodegen::QrCode::drawFinderPattern(int, i
  0.00      0.02     0.00        3     0.00     0.00  void std::vector<int, std::allocator<int> >
  0.00      0.02     0.00        2     0.00     0.00  qrcodegen::BitBuffer::BitBuffer()
  0.00      0.02     0.00        2     0.00     0.00  std::vector<std::vector<bool, std::allocato
  0.00      0.02     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN9qrcodegen9QrSegment4Mode
  0.00      0.02     0.00        1     0.00     0.00  _GLOBAL__sub_I_main
  0.00      0.02     0.00        1     0.00    20.00  qrcodegen::QrCode::encodeText(char const*,
  0.00      0.02     0.00        1     0.00     0.00  qrcodegen::QrCode::drawVersion()
  0.00      0.02     0.00        1     0.00     0.00  qrcodegen::QrCode::drawCodewords(std::vecto
  0.00      0.02     0.00        1     0.00    20.00  qrcodegen::QrCode::encodeSegments(std::vect
  0.00      0.02     0.00        1     0.00     0.00  qrcodegen::QrCode::drawFunctionPatterns()
  0.00      0.02     0.00        1     0.00     0.11  qrcodegen::QrCode::reedSolomonComputeDiviso
  0.00      0.02     0.00        1     0.00    20.00  qrcodegen::QrCode::QrCode(int, qrcodegen::Q
```

2 How to Parallelize the Algorithm

2.1 Proposal for Parallelization

After reviewing the results of `gprof` and analyzing the originally implemented code in depth, we decided to parallelize as much as possible except for portions of code that have something related to reading up to the terminating character (`'\0'`) of a string, portions of code that had `return` in the middle of the code and not at the end of each function or portions of code where insertions are made to arrays that must preserve an order, since a QR code must have a certain order as described in the ISO/IEC 18004 standard.

With this in mind, we will proceed to list the portions of code to be parallelized, in order to speed up the process of generating a QR code. The portions of code that will be listed will be accompanied with the corresponding line of code from the *original repository* (more specifically in the file "QrCode.cpp"). This is expected to give the reader a better understanding of what is being proposed.

- **Lines 282 to 285:** it is proposed to parallelize this cycle with `#pragma omp parallel for`. This decision was made after having analyzed the flow line and the logic of the algorithm. It is believed that the `#pragma omp critical` instruction will be needed to avoid a race condition where two threads attempt to assign a value to the `ecl` variable, at line 284.

- **Lines 312 and 313:** is intended to use MPI, partitioning the size of the *for* into a certain number of processes, so that each one gets a portion of the *for* loop. After this, the *Ireduce* instruction will be used, which allows to collapse the information of all the processes in a master process. It can be noticed that in line 313 a logical OR is made, which is supported by the *Ireduce* instruction. In addition, the non-blocking version was chosen so that the processes have greater versatility and deadlocks are avoided.

- **Lines 418 to 421:** is a simple loop that is intended to be easily parallelized with OpenMP.

- **Lines 431 to 437:** this part is a double nested loop, which makes a call to an external function. In case problems arise that cannot be solved with the OpenMP instructions, then this portion of code will be attempted with MPI. If the problem persists, then this idea will be discarded.

- **Lines 445 to 470:** in this portion of code is the function `drawFormatBits`, which has several non-nested for cycles, so it is intended to parallelize each and every one of them, placing barriers after each for cycle to synchronize the threads. In the implementation of the proposals it will be analyzed if the sentences included in lines 449 and 450 will be carried out with OpenMP or with MPI.

- **Cycles `for` of lines 479 and 486** we intend to parallelize both, taking care not to produce race conditions. Once we have a first version of the parallel algorithm, we will check if there are race conditions between the threads or problems with the processes.

- **Lines 498 to 503:** is intended to exploit MPI, so that the *for* cycle is distributed in the processes.

- **Lines 575 to 590:** it is intended to fully parallelize those three nested for cycles. Perhaps one with OpenMP and another with MPI or visceversa, in the implementation it will be decided. Care must be taken since inside the body of these functions there are variable updates that are very important for the correct operation of the algorithm, so it will be seen in the future if the `atomic` or `critical` instructions will be used.

- **Lines 600 to 616:** similar to the previous proposals, it is intended to parallelize the double nested cycle, taking special care with the case in which an exception is thrown.

- **Lines 628 to 642:** in this portion of the code you must be quite careful with variable updates, since we assume (we are quite convinced) that race

conditions will be present, so we assume that it will be necessary to use statements that avoid these problems.

- **Lines 670 to 676:** it is expected that MPI can be used to carry the accumulated sum within the body of that *for* loop, probably a statement that communicates all the processes to me and allows sending a message with the accumulated sum of each process to a master process.
- **Lines 681 to 686:** is a bit more of the same, i.e., that nested double-loop *for* can be fully parallelized with OpenMP.
- In addition, we consider it relevant to try to parallelize the sections between lines 746 to 754 (or at least a part of this section); or lines 765 and 766 or the code section between lines 775 and 778. These considerations will be evaluated in the implementation of the project, it will be seen if they can be parallelized either with OpenMP or with MPI.

It will be done a emphasis in the parallelization and optimization of the last section that was mentioned (the one between 775 and 778 lines), since these lines are part of one of the most time-consuming functions as shown in the `gprof` section.

2.2 Data Independence

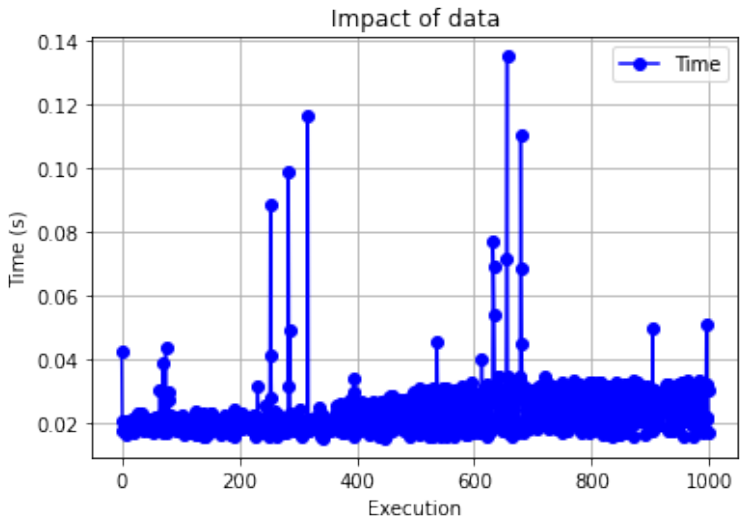
We analyzed the full code from the `cpp` repository and then we separate the functions with data independence from the ones without, or at least the portion of code inside a function that have data independence to try to parallalize it. This verifies that the portions of code mentioned in the previous subsection do not have any data dependency, thus facilitating their parallelization.

We decided to parallelize as much as possible except for portions of code that have something related to reading up to the terminating character (`'\0'`) of a string, portions of code that had `return` in the middle of the code and not at the end of each function or portions of code where insertions are made to arrays that must preserve an order.

3 Verification and Analysis

3.1 Impact of Data

To verify the impact of the data content on the performance of the algorithm, the creation of the `text` (with 2.600 elements) was executed 1.000 times, and the execution time of the creation of the SVG string with each of these texts was taken. As a result, the mean time was $\bar{t} = 0.02401$ seconds and the standard deviation was $\sigma_t = 8.7993 \times 10^{-3}$ seconds. In the following graph you can see the time it took for the algorithm to run in each iteration.



As can be seen from the graph above and based on the standard deviation, the execution time does not vary much depending on the array data.

Theoretically, the algorithm takes the characters, identifies their type (alphanumeric or numeric) and proceeds to perform the respective operations, which are done bit by bit, which does not affect the execution time by varying the character. For this reason the execution time of the application is not affected by the text data, but by its length.

3.2 Data Races

There was an error with the coderrect, it is believed that it was due to the use of an own namespace used in the original code. As an alternative proposed by the professor, the functions are evaluated with the worst cases, which in this case was 2600 where it was evidenced that apparently there was no data race. After testing approximately 100 times it was evident that the final result of the QR code did not change.

3.3 Issoeficency, Strong and Weak Scaling

3.3.1 Isoefficiency

To verify isoefficiency, we measured the efficiency of three text sizes: 650, 1300, and 2600, each with the four possible threads. The averages and deviations can be seen in the following tables:

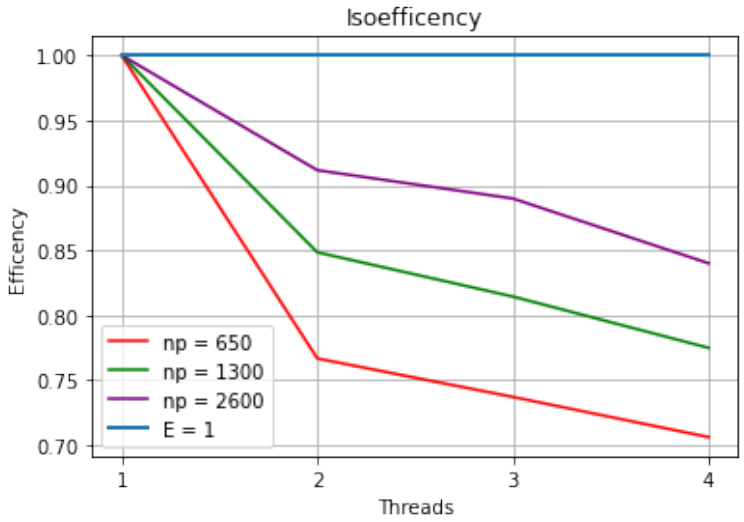
Data Size	1 thread	2 threads	3 threads	4 threads
650	1.00000	0.76667	0.73704	0.70613
1300	1.00000	0.84835	0.81429	0.77480
2600	1.00000	0.91154	0.88974	0.83990

Table 1: Efficiency (Average)

Data Size	1 thread	2 threads	3 threads	4 threads
650	0.00000	0.03727	0.03638	0.02468
1300	0.00000	0.02702	0.01065	0.00615
2600	0.00000	0.01053	0.00702	0.01049

Table 2: Efficiency (Standard Deviation)

Which can be seen better in the following graphic:

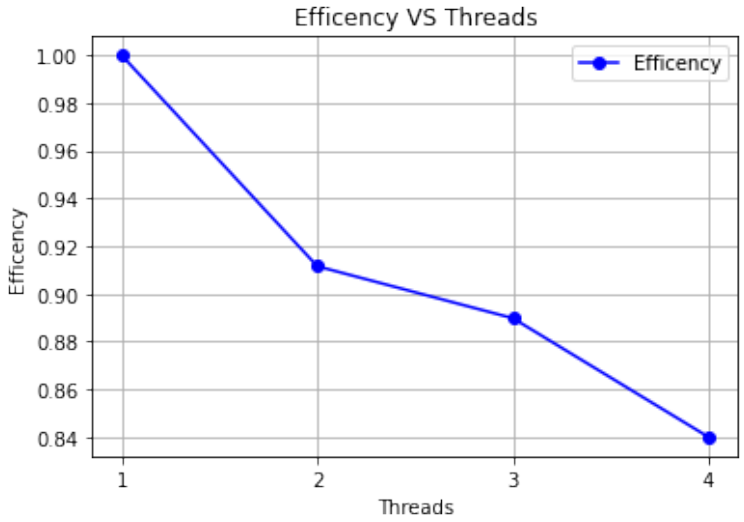


3.3.2 Strong Scaling

For the strong scaling analysis we took the size of the worst-case problem, and calculated the efficiency for each number of threads with the given n and obtained the following results:

Threads	Efficiency (Average)	Efficiency (Standard Deviation)
1	1.00000	0.05477
2	0.91154	0.08944
3	0.88974	0.17889
4	0.83990	0.08367

Which can be seen better in the following graph:

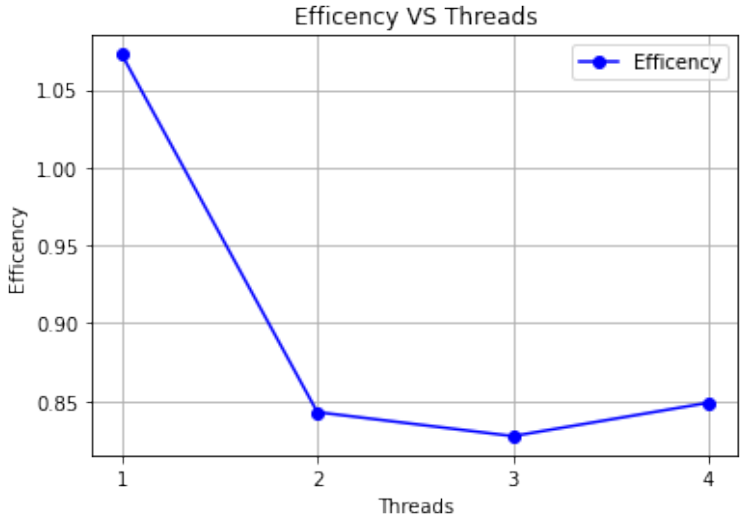


3.3.3 Weak Scaling

For the weak scaling analysis we took the problem size for each core $n = 650$ and calculated the efficiency for each number of threads with the given n and obtained the following results:

Threads	Efficiency (Average)	Efficiency (Standard Deviation)
1	1.07273	0.04472
2	0.84286	0.05477
3	0.82745	0.05477
4	0.84892	0.04472

Which can be seen better in the following graph:



4 How It Was Paralyzed

Initially, the intention was to parallelize as much as possible. An analysis of the code was done at the time of submitting the design. However, at the moment of doing what was proposed in the design, we realized that some things that we thought could be parallelized, turned out not to be so. Listed below are the lines of code (the new, parallel one) where either some OpenMP or some MPI was applied.

- **Line 332 to 341:** Here it was decided to use the `sections` by dividing the `cycle` in half and two sections were used, one for each half of the `section`. This is because we wanted to try a different way than the parallel for.
- **Line 468:** A parallel for cycle was used because there were no operations with which it had problems.
- **Line 471 to 500:** There was an internal loop inside the `for` on line 468 that could be parallelized by using a parallel `for` loop or by splitting it into two parts and using `omp` sections, which was the decision that was made.
- **Line 556:** It is a simple `for` loop that has no major problem when it comes to being parallelized since there are no operations with dependencies.
- **Line 564:** It is a simple `for` loop that has no major problem when it comes to being parallelized since there are no operations with dependencies.
- **Line 576:** It is a simple `for` loop that has no major problem when it comes to being parallelized since there are no operations with dependencies. However, despite having a conditional in it, it was not necessary to use the critical.
- **Line 593 and 595:** Two nested `for` cycles that can be parallelized since only one operation is called within the deepest cycle, regardless of which parameters are used.

- **Line 690 and 692:** Two nested `for` cycles that can be parallelized since inside the deepest cycle there is only one conditional switch that has no operations with dependencies and outside the next for cycle there is also another one without dependencies.
- **Line 719 and 724:** Two nested `for` cycles that has no major problem with them it comes to being parallelized since there are no operations with dependencies. Given that the conditional inside takes one of two paths.
- **Line 743 and 748:** Two nested `for` cycles that has no major problem with them it comes to being parallelized since there are no operations with dependencies. Given that the conditional inside takes one of two paths.
- **Line 778 to 786:** Here the `MPI` part is performed, what is basically done is that based on a summation operation given a conditional reduces them to a precise position with the `MPI` function: `MPI_Ireduce`.
- **Line 858 to 883:** Here again we had a cycle `for` that we decided to do it with a `OMP` section and it was needed to divide the `for` cycle.

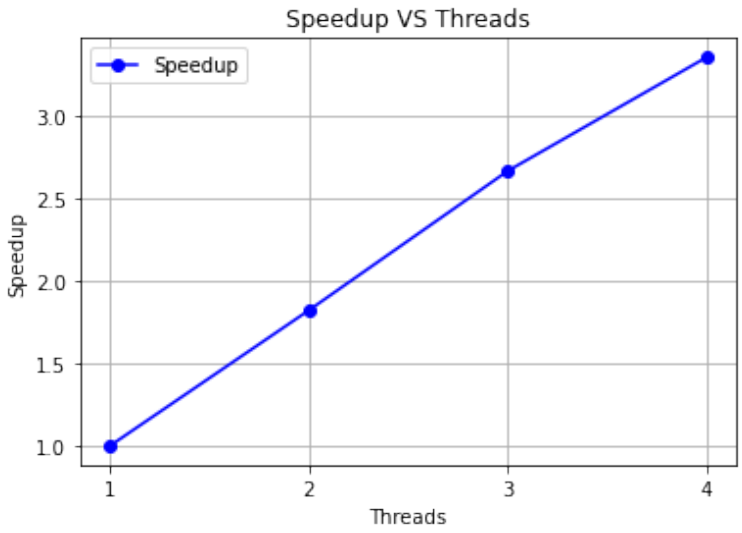
5 Results

5.1 Speedup

To calculate the speedup, it was taken 5 times (with the worst-case) and the average and standard deviation were obtained as shown in the following table:

Threads	Speedup (Average)	Speedup (Standard Deviation)
1	1.00000	0.00000
2	1.82308	0.02107
3	2.66923	0.02107
4	3.35960	0.04194

Which can be seen better in the following graph:

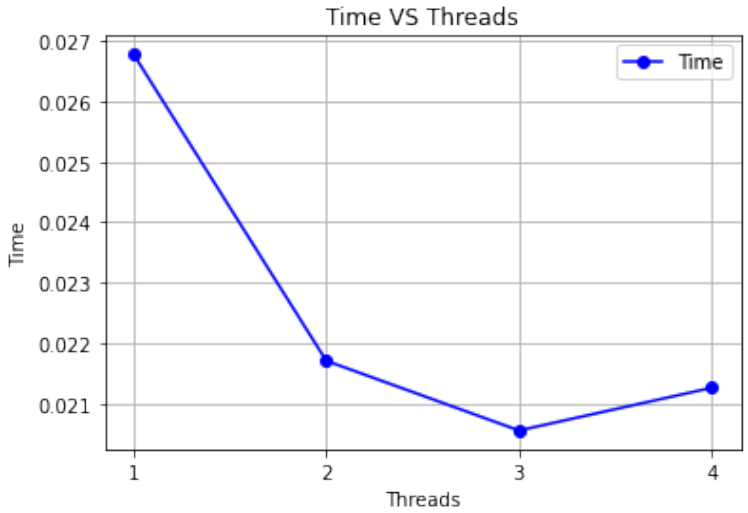


5.2 Execution Time

To calculate the execution time, it was taken 5 times (with the worst-case) and the average and standard deviation were obtained as shown in the following table:

Threads	Time (Average)	Time (Standard Deviation)
1	0.02677	0.00056
2	0.02171	0.00016
3	0.02056	0.00259
4	0.02127	0.00083

Which can be seen better in the following graphic:



The execution time was also measured depending on the data size and the number of threads used. The following two tables show the average and deviation, respectively, taking into account these factors.

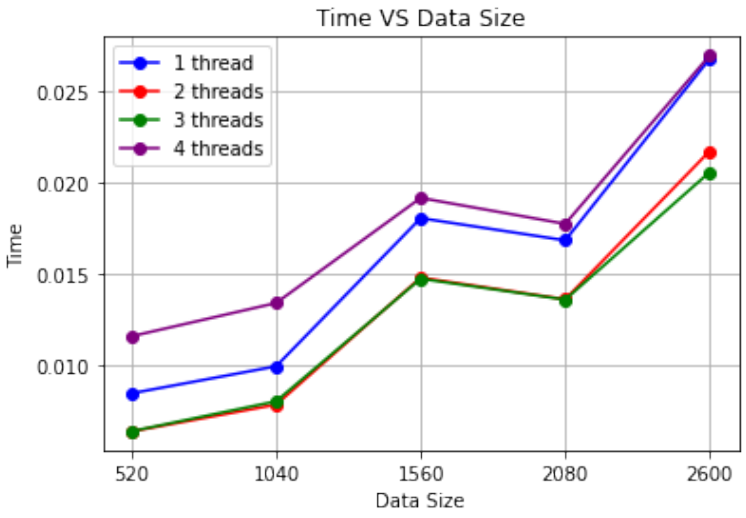
Data Size	1 thread	2 threads	3 threads	4 threads
520	0.00845	0.00636	0.00637	0.01157
1040	0.00993	0.00783	0.00801	0.01340
1560	0.01804	0.01478	0.01472	0.01915
2080	0.01682	0.01360	0.01358	0.01773
2600	0.02677	0.02171	0.02056	0.02695

Table 3: Time (Average)

Data Size	1 thread	2 threads	3 threads	4 threads
520	0.00021	0.00009	0.00003	0.00331
1040	0.00021	0.00019	0.00004	0.00257
1560	0.00034	0.00021	0.00014	0.00218
2080	0.00023	0.00017	0.00003	0.00272
2600	0.00056	0.00016	0.00259	0.00276

Table 4: Time (Standard Deviation)

Which can be seen better in the following graphic:

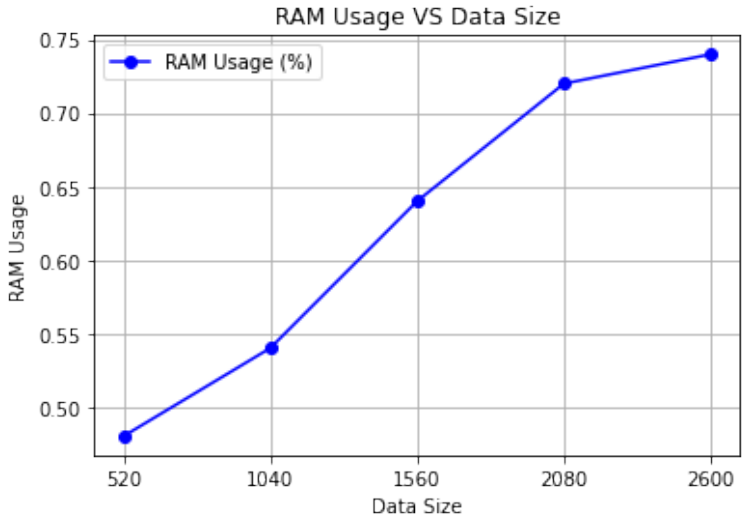


5.3 RAM Usage

The percentage of RAM used by the program during execution with four threads was measured, for different text lengths. The results can be seen in the following table:

Data Size	RAM Usage (Average)	RAM Usage (St. Deviation)
520	0.48	0.04
1040	0.54	0.05
1560	0.64	0.05
2080	0.72	0.04
2600	0.74	0.05

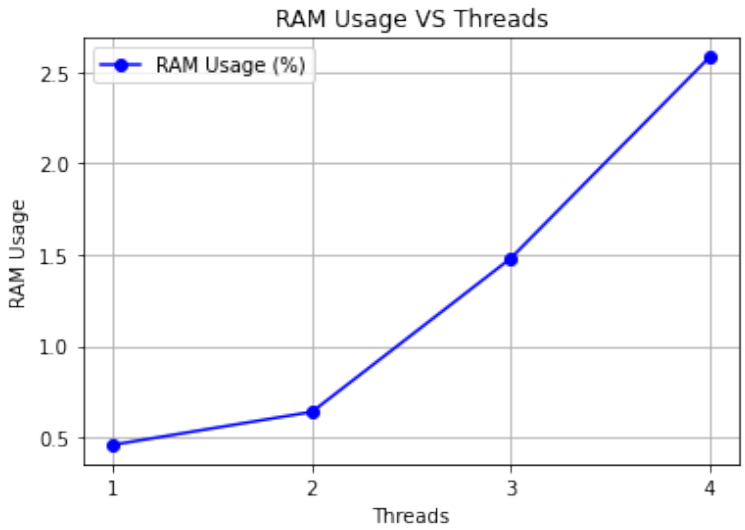
Which can be seen better in the following graphic:



The percentage of RAM used by the program during worst-case execution was measured, for each number of threads. The results can be seen in the following table:

Threads	RAM Usage (Average)	RAM Usage (St. Deviation)
1	0.46	0.05
2	0.64	0.09
3	1.48	0.18
4	2.58	0.08

Which can be seen better in the following graphic:



6 Problems With Parallelization

The problems encountered in executing what was initially presented in the design document were little evident when they were proposed, these were:

- When implementing what had been planned in the design, it was found that some of the statements such as `#pragma omp critical` did not allow for good parallelization, since in `for` nested loops that have too many iterations, it slowed down too much, and the execution time increased considerably ($\approx +20$s). When testing the code without this option, we were able to see an improvement in the time and the correctness of the algorithm was maintained.
- Some of the parts of the code could not be parallelized, especially the `for` cycles that were of type *for each*, or those that had a data dependency that we had not previously detected. Fortunately, these cycles did not represent a large part of the execution time, and some others that had not been considered in the design were parallelized, which compensates a little for the fact that they were not parallelized.
- Some other proposed parallelizing `for` cycles took into account the order in which a QR code should be generated, but this dependency was highly implicit, so we did not notice it in the previous design analysis.

It is likely to encounter problems with race conditions between data, which can be manipulated by several threads at the same time. However, after doing about 100 QR code generation tests, we found that the mentioned changes or problems did not alter the correct creation of QR codes.

7 Application Usage

QR codes are widely used in different areas, such as marketing, payments, and so on. Since they store different types of information, they can be used for:

- Display simple text, with which welcome messages can be given at conferences, for example.
- Provide addresses that allow localization.
- URLs that allow direct access to a certain web page. It is also possible to send download links for mobile applications that are redirected to the Google Play Store or App Store.
- Allows to make payments, storing bank or credit card data.
- Online account authentication, where web pages allow a QR code to be displayed for a user to log in. Also used in two-step verification.
- Due to the pandemic, some sectors, such as restaurants or banks, have used QR codes to manage menus or some services, in addition to payments.

8 References

- Course material, available on Blackboard
- Nayuki. 2020. Taken from this Github Repo: cpp-QR-Code-generator
- Real-world applications of QR codes