FINAL PROJECT

Design

Juan José Betancourt
Nicolás Delgado
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

Jun 2, 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.

1.1.1 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;
}
```

1.1.2 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:

```

QrCode QrCode::encodeSegments(const vector<QrSegment> &segs, Ecc ecl,
    int minVersion, int maxVersion, int mask, bool boostEcl) {
  if (!(MIN_VERSION <= minVersion && minVersion <= maxVersion && maxVersion
     <= MAX_VERSION) || mask < −1 || mask > 7)
    throw std::invalid_argument("Invalid value");

  //Find the minimal version number to use
  int version, dataUsedBits;
  for (version = minVersion; ; version++) {
    int dataCapacityBits = getNumDataCodewords(version, ecl) ∗ 8; // Number of data
     bits available
    dataUsedBits = QrSegment::getTotalBits(segs, version);
    if (dataUsedBits != −1 && dataUsedBits <= dataCapacityBits)
      break; // This version number is found to be suitable
    if (version >= maxVersion) { // All versions in the range could not fit the given
     data
      std::ostringstream sb;
      if (dataUsedBits == −1)
        sb << "Segment too long";
      else {
        sb << "Data length = " << dataUsedBits << " bits, ";
        sb << "Max capacity = " << dataCapacityBits << " bits";
      }
      throw data_too_long(sb.str());
    }
  }
  if (dataUsedBits == −1)
    throw std::logic_error("Assertion error");

  //Increase the error correction level while the data still fits in the current version
     number
  for (Ecc newEcl : vector<Ecc>{Ecc::MEDIUM, Ecc::QUARTILE, Ecc::HIGH}) { //
     From low to high
    if (boostEcl && dataUsedBits <= getNumDataCodewords(version, newEcl) ∗ 8)
      ecl = newEcl;
  }

  // Concatenate all segments to create the data bit string
  BitBuffer bb;
  for (const QrSegment &seg : segs) {
    bb.appendBits(static_cast<uint32_t>(seg.getMode().getModeBits()), 4);
    bb.appendBits(static_cast<uint32_t>(seg.getNumChars()), seg.getMode().
     numCharCountBits(version));
    bb.insert(bb.end(), seg.getData().begin(), seg.getData().end());
  }
  if (bb.size() != static_cast<unsigned int>(dataUsedBits))
    throw std::logic_error("Assertion error");

  // Add terminator and pad up to a byte if applicable
  size_t dataCapacityBits = static_cast<size_t>(getNumDataCodewords(version, ecl)) ∗ 8;
  if (bb.size() > dataCapacityBits)
    throw std::logic_error("Assertion error");
  bb.appendBits(0, std::min(4, static_cast<int>(dataCapacityBits − bb.size())));
  bb.appendBits(0, (8 − static_cast<int>(bb.size() % 8)) % 8);
  if (bb.size() % 8 != 0)
    throw std::logic_error("Assertion error");

  // Pad with alternating bytes until data capacity is reached
  for (uint8_t padByte = 0xEC; bb.size() < dataCapacityBits; padByte ^= 0xEC ^ 0x11)
    bb.appendBits(padByte, 8);

  // Pack bits into bytes in big endian
  vector<uint8_t> dataCodewords(bb.size() / 8);
  for (size_t i = 0; i < bb.size(); i++)
    dataCodewords[i >> 3] |= (bb.at(i) ? 1 : 0) << (7 - (i & 7));

  // Create the QR Code object
  return QrCode(version, ecl, dataCodewords, mask);
}
```

1.1.3 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:

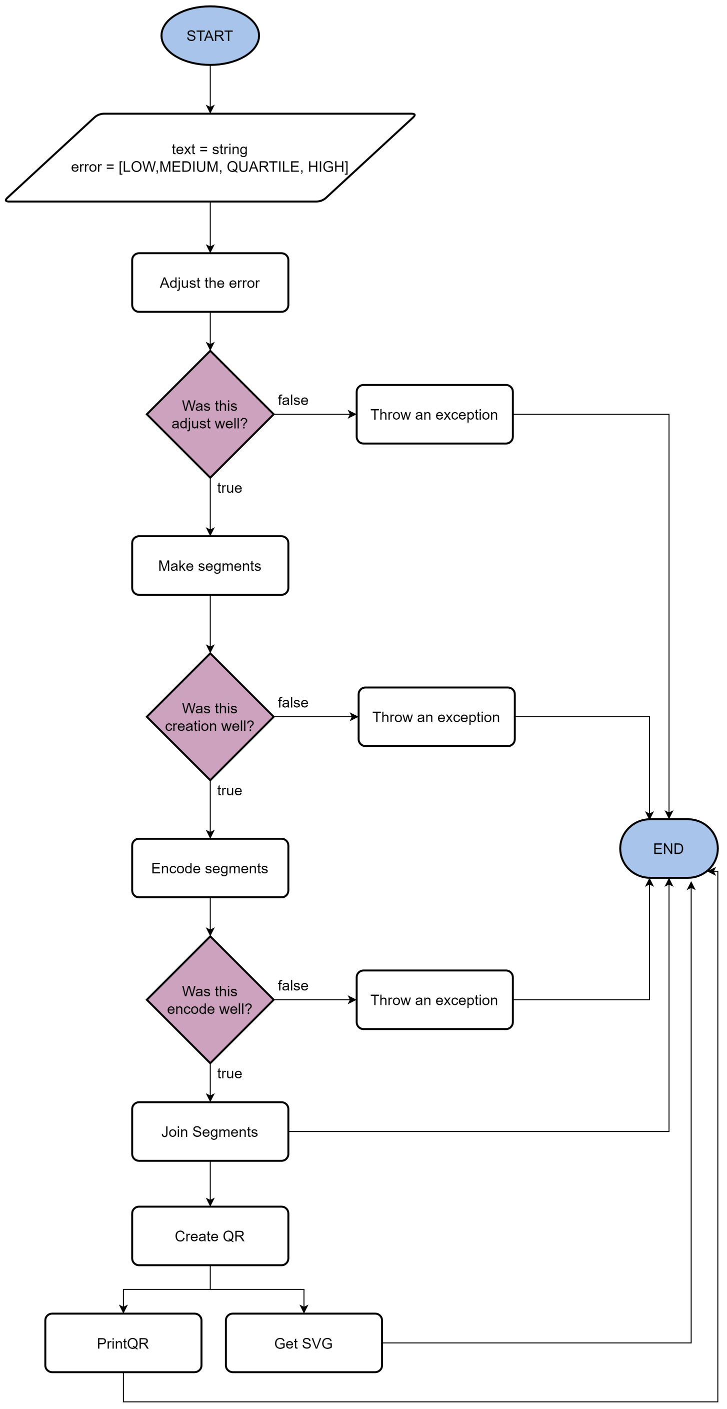
The algorithm basically what it does is, based on a size for the error (which can be corrected as it increases) and a text, to see if the text fits in the respective error, and adjusts it. After this, it goes to the encoding process where the segments are created according to the type of values that 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
END
Encode segments
Was this encode well?
false
Throw an exception
true
Join Segments
Create QR
PrintQR
Get SVG

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`.

1.3.1 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.

1.3.2 Encode Text

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

1.3.3 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.

1.3.4 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.

1.3.5 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	GCC 7.5.0
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

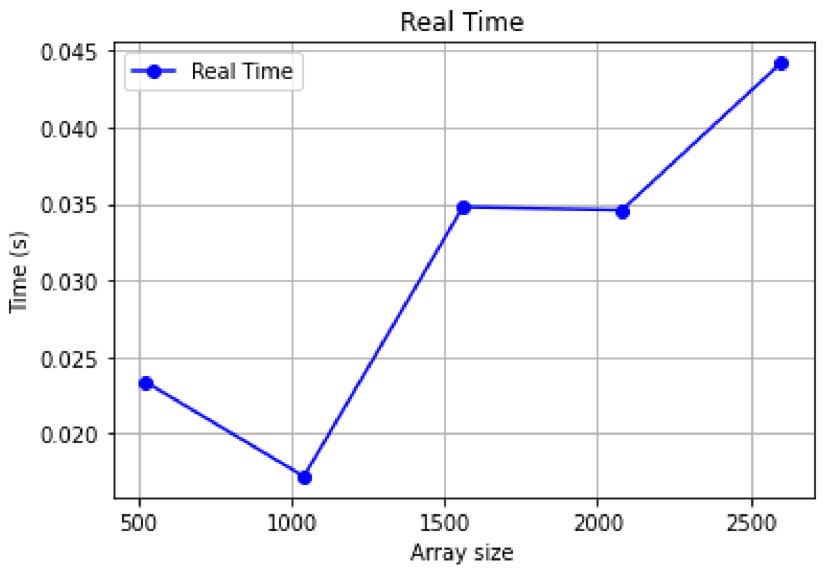
Configuration (RAM)	**Value (RAM 1)**	**Value (RAM 2)**
Total Width	64 bits	64 bits
Data Width	64 bits	64 bits
Size	4GB	8GB
Type	DDR4	DDR4
Type Detail	Sync. Unbuffered	Sync. Unbuffered
Speed	2133 MT/s	2133 MT/s
Manufacturer	Samsung	Kingston
Configured Clock Speed	2133 MT/s	2133 MT/s

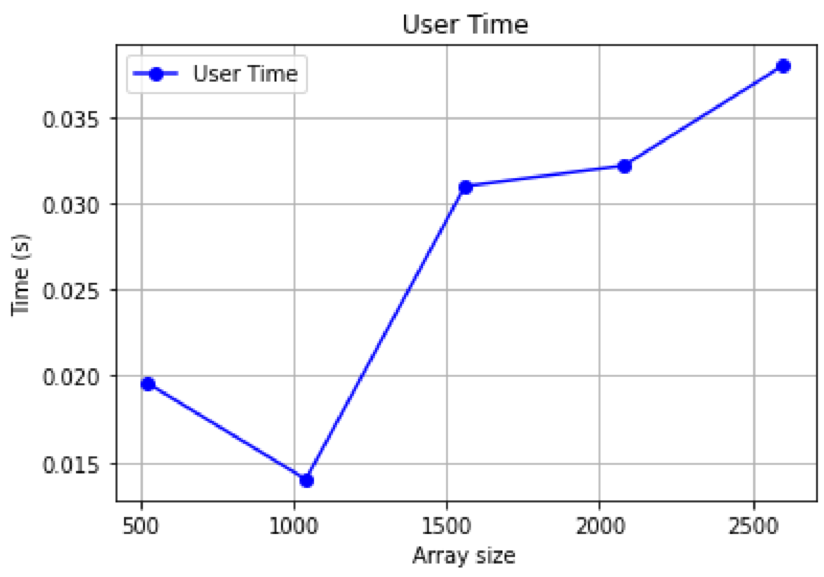
Configuration (Disk)	**Value**
Model Name	ST2000LM007
Capacity	2 TB
Width	32 bits
Speed	66MHz
Buffer size	128 MB
Average Seek Time	13 ms
Data Transfer Rate	600 MBps
Internal Data Rate	100 MBps

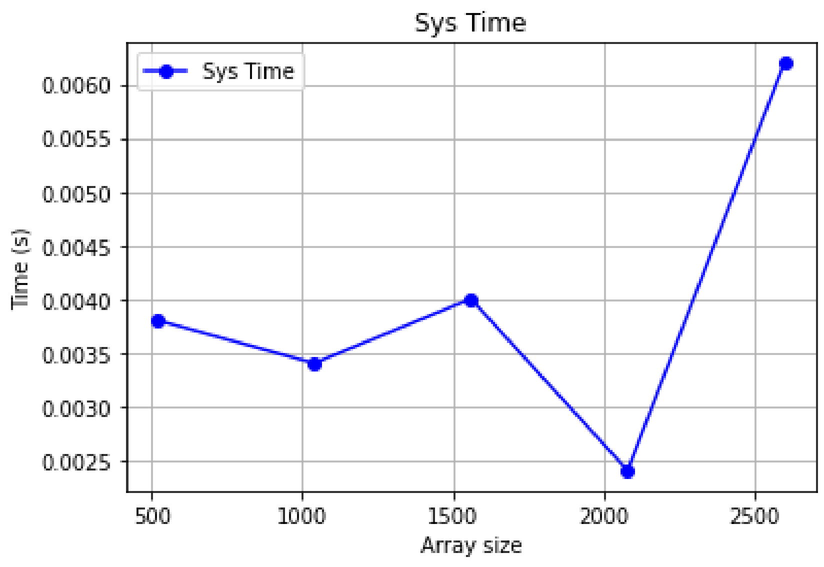
Measuring the execution time with the `time` tool for different values of `n`, the following result was obtained:

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:



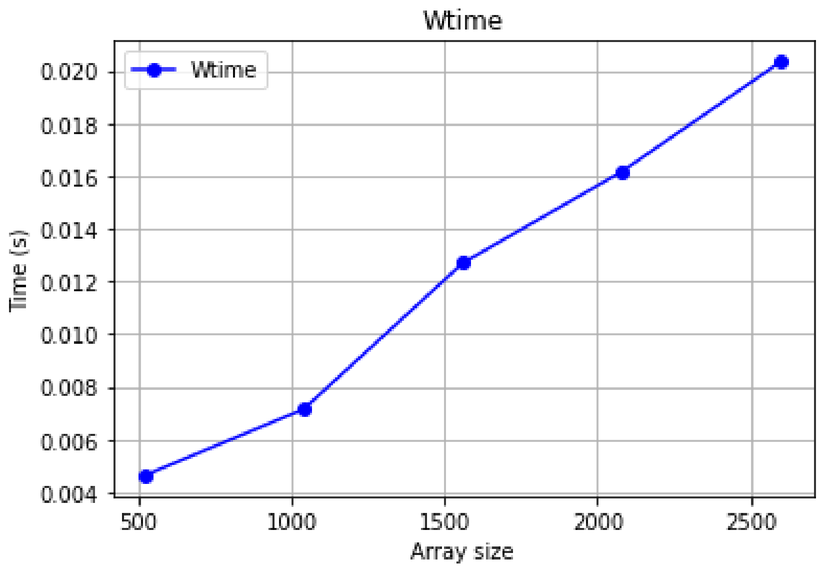

User Time
User Time
Time (s)
0.035
0.030
0.025
0.020
0.015
500
1000
1500
2000
2500
Array size


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

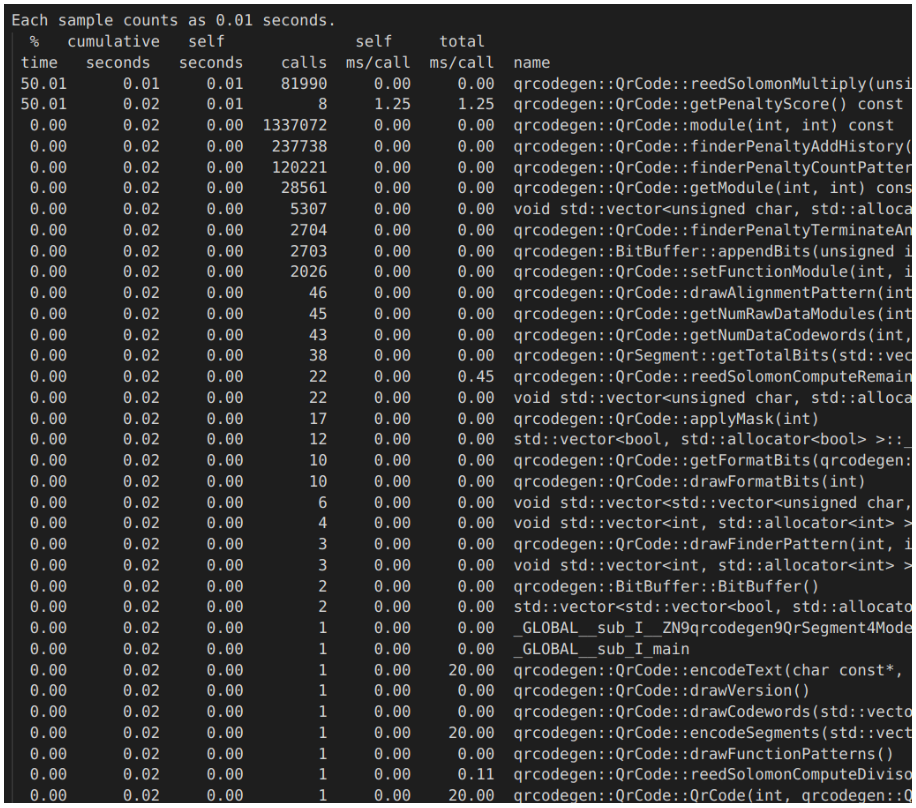
Using the `wtime` tool to measure the execution time of the `doBasicDemo()` function at the same input sizes, the following results are obtained:

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 References

- Course material, available on Blackboard
- `Nayuki`. 2020. Taken from this Github Repo: `cpp-QR-Code-generator`