π project

Algorithms & Computability course at the MiNI faculty

Kamil Monicz Marcin Wojnarowski

November, 2022

Contents

1 Description

The π project for the Algorithms & Computability course consists of three components:

1. Generation of π - program generating consecutive decimal digits of π (ratio of a circle's circumference to its diameter) into a text file
2. Finding substrings - program finding a specified string of digits in a given file
3. Comparing files - program comparing two solutions and finding the difference

2 Program

Our solution consists of a single program written in C which exposes required functionalities as subcommands. Namely:

```
./pi: pi generator, substring finder, and solution comparator.
Available commands:
        ./pi generate [file] [n-digits]: generates #n-digits digits of pi into the specified file
        ./pi find [file] [substring]: finds the given substring in the given file and prints the starting index
        ./pi compare [file1] [file2]: compares contents of two files and points to (if there is one) a difference
        ./pi table [pi_file] [out_file] [n]: writes a table of substrings of natural numbers up to n to out_file from the pi_file
```

2.1 Generate

The task of finding consecutive digits of PI has been bugging humanity for thousands of years. Recently with the rise of automated computing the amount of digits we know has skyrocketed. The last 9 world records have all used the Chudnovsky algorithm which is a hypergeometric series which quickly converge to the digits of π. The current world record stands at 10^{14} decimal places, and the algorithm is the following (Alexander J. Yee, n.d.):

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(545140134n + 13591409)}{(3n)!(n!)^3 \left(640320\right)^{3n+\frac{3}{2}}}$$

We will use this algorithm as well.

2.1.1 Algorithm description

To be usable, it has to be simplified and adjusted for iterative computation. The factorial that appears in the computation should preferably be eliminated since it is expensive to compute. The following simplified form will be used (which is the decomposition of the original formula):

$$\pi = C \left(\sum_{n=0}^{\infty} \frac{M_n \cdot L_n}{X_n} \right)^{-1}$$

where

$$\begin{aligned} C &= 426880\sqrt{10005} \\ M_n &= \frac{(6n)!}{(3n)!(n!)^3} \\ L_n &= 545140134n + 13591409 \\ X_n &= (-262537412640768000)^n \end{aligned}$$

The iterative difference can be calculated. First we find the zero terms:

$$\begin{aligned} M_0 &= \frac{(0)!}{(0)!(0!)^3} = 1 \\ L_0 &= 545140134 \cdot 0 + 13591409 = 13591409 \\ X_0 &= (-262537412640768000)^0 = 1 \end{aligned}$$

Then finally find the value of the next term relative to the previous one.

$$\begin{aligned} M_{n+1} &= M_n \cdot \left(\frac{(12n+2)(12n+6)(12n+10)}{(n+1)^3} \right) \\ L_{n+1} &= L_n + 545140134 \\ X_{n+1} &= X_n \cdot (-262537412640768000) \end{aligned}$$

To speed up calculations we applied the technique of binary splitting described both in (Bruno Haible, n.d.) and (Alexander J. Yee, n.d.). In short, it reformulates a formula of a sum of divisions

(which what the Chudnovsky formula is) into a single division of two large operands (sum of fractions becomes a single fraction). Division is expensive, thus this technique is practically much more efficient. Binary splitting also allows for simple parallelization and checkpointing. A more extensive analysis of the Chudnovsky formula can be found in (Milla 2018).

Computation will operate on large decimal numbers, which means there will be a need to store arbitrary precision numbers in a custom data structure. The amount of memory available is limited (only 8 to 16 GB) which means the limiting factor for this task will be the memory, not time itself. We believe we can fill the whole memory with computed digits in a matter of minutes. An optimization to memory can be performed by the means of swapping with a persistent storage (for instance a hard disk). But such an implementation would require considerably more time to implement, and thus we will not do it.

The time complexity of the algorithm is $\mathcal{O}(n \cdot \log(n)^3)$ while the space complexity is simply $\mathcal{O}(n)$ where n is the amount of digits we want to calculate.

2.2 Find

The task of finding a substring is often solved by a family of algorithms called string-searching algorithm. The simplest algorithm is the naive one, where each character in the pattern is checked one by one against the string and on a mismatch we move the string pointer by one and start again. This however is of complexity $\mathcal{O}(mn)$ where m is the length of the pattern and n is the length of the string. An improvement over the naive algorithm is the Knuth-Morris-Pratt and it is the following (Knuth, Morris, and Pratt 1977):

2.2.1 Algorithm description

First we introduce a preprocessing step. We construct a longest prefix suffix (LPS) array of size m. This array will later allow to skip pattern characters when searching. The i-th element of the LPS array is length of the longest prefix of the substring of the pattern from the start up to i-th character while also being its suffix. For instance, consider the following π pattern: 434543524. We can construct its LPS:

4	3	4	5	4	3	5	2	4
0	0	1	0	1	2	0	0	1

- When $i = 2$ (zero-indexed) the pattern substring is "434" so the value of LPS is 1, because "4" is both a prefix and a suffix
- When $i = 5$ (zero-indexed) the pattern substring is "434543" so the value of LPS is 2, because "43" is both a prefix and a suffix

Construction of this LPS array can be done in $\mathcal{O}(m)$. Now that we have this array we can move onto searching. The array allows us to skip characters upon a mismatch without a need of restarting the string pointer.

1. We keep track of two pointers, one for the string (i) and one for the pattern (j)
2. We keep matching the string and pattern while incrementing both pointers
3. If we reach the end of the pattern, we have found the substring and we can exit
4. If we find a mismatch we do not increment the pointers. Instead, we reinitialize j to a new value: $LPS[j-1]$. This is due to the following observation: we know that the previous j characters of the pattern did match the string, additionally we know that $LPS[j-1]$ is

the count of characters of the pattern which are both the prefix and the suffix, therefore we do not need to match the first $LPS[j-1]$ characters because we already know they will match.

1. Edge case when we find a mismatch at $j = 0$: we just increment i by one instead

The complexity of this solution is: time $\mathcal{O}(n+m)$ and space $\mathcal{O}(m)$ (LPS array). The time complexity can be obtained by the following reasoning:

- The LPS is constructed in $\mathcal{O}(m)$ operations
- The loop performs constant operations
- The loop always increments the i pointer where $i \in [0; n] \subset \mathbb{N}$, except when there is a mismatch and $j \neq 0$
- When the above case happens, notice that the amount of times we are mismatching is in the worst case directly proportional to the amount of correct matches (where i is incremented), so the total amount of operations is at most $2n$. If we matched k characters, we advanced the pattern index by k and thus we can reduce it only by at most k (since $j > LPS[j-1]$)
- The loop stops when $i = n$

Thus $\mathcal{O}(m + 2n + C) = \mathcal{O}(n + m)$

2.3 Compare

The task of comparing file contents is of a linear nature. The content of files is unstructured and no assumptions can be made about it, meaning no heuristics can be applied to improve the complexity of the comparison. We are thus stuck with $\mathcal{O}(\min(m, n))$ where m and n are lengths of the content of the compared files. However, one can take advantage of the available hardware to speed up the process.

Memory wise, it depends on the chosen buffer for file reading. Since it is known we will be working with large files, any buffer size depending on the input size would be a poor decision. Therefore we chose a constant size of $1024 \cdot 1024\text{B} = 1\text{MiB}$ for each file. This size was picked somewhat arbitrarily based on empirical tests.

2.3.1 Algorithm description

In a loop each file is read into its appropriate buffer and then comparisons are made. For comparisons, we employ a simple trick to take advantage of the platform. Instead of comparing character by character (byte by byte) we compare by the word size of the underlying CPU. We transmute the memory of the byte buffer into a word buffer (few bytes combined, depends on the bit architecture of the CPU). Then, comparisons are done on this new buffer. This way, we cut down on the number of comparisons by a constant factor. For example, on a x86-64 CPU we perform 8 times less comparisons (64 bits $\rightarrow$ 8 bytes $\rightarrow$ one word). To illustrate this effect we can examine the generated assembly for a x86-64 CPU using the byte and word comparison (Fig. 1). In practice this optimizations yielded on average 2.24 times faster runtime.

Once a mismatch is found, the composed word is broken down into individual bytes to find the offending byte. Once found, the comparison is stopped and the found offset where the difference lies is returned. If no mismatch is found till the end of both files, program reports that no differences are found.

If we reach the end of one of the files, the program is stopped and the current offset is returned.

Figure 1: Generated x64 assembly for a byte and word comparison. The amount of instructions is the same, but byte comparison uses the `cmpb` instruction while word comparison uses `cmpq` which compares 8 byte registers. Source: Compiler Explorer.

To sum up: $m + n$ reads are done and $\frac{\min(m,n)}{\text{CPU word size}}$ comparisons are done which means the algorithm time complexity is $\mathcal{O}(m + n + \frac{\min(m,n)}{\text{CPU word size}}) = \mathcal{O}(\min(m, n))$, linear.

3 Results

3.1 Testing

As baseline a file containing 10^9 correct digits of π was used (https://stuff.mit.edu/afs/sipb/contrib/pi/pi-billion.txt but with the decimal point removed). Then our program was ran to generate 10^9 digits (Fig. 2). This took roughly 30 minutes. Afterwards a comparison was performed between our generated digits and `pi-billion.txt` (Fig. 3). A mistake was found at position $10^9 - 1$ (zero-indexed), thus we successfully generated the requested amount of digits.

```
→ ./build/apps/pi generate out.txt 1000000000
```

Figure 2: Command used to generate 1 billion digits of π.

```
→ ./build/apps/pi compare out.txt pi-billion.txt
Given files differ at position 999999999:
out.txt: …115275045520…
pi-billion.txt: …115275045519
```

Figure 3: Command used to compare generated 1 billion digits of π with a baseline.

Afterwards we can perform substring searches on the generated files, for example finding π in π (Fig. 4). Finally, we can generate a table containing all substrings formed from consecutive natural numbers until a specified max value (Fig. 5).

In addition, in the `test/` directory one can find unit tests of individual functions to ensure their correctness.

```
→ ./build/apps/pi find out.txt 031415926
Found substring in the given file at position 573011190:
…562024928203141592643214756 44…
```

Figure 4: Command used to find 8 first digits of π in π itself. The searched string was prefixed by "0" to avoid the obvious match which is the prefix of π.

```
→ ./build/apps/pi table out.txt table.csv 10000
→ head -n20 table.csv
0,32
1,1
2,6
3,0
4,2
5,4
6,7
7,13
8,11
9,5
10,49
11,94
12,148
13,110
14,1
15,3
16,40
17,95
18,424
19,37
```

Figure 5: Command used to generate a table containing all substrings formed from consecutive natural numbers until 10000 and displaying a part of it.

3.2 Benchmarks

The final implementation was ran on Linux x64 with an AMD Ryzen 5 3600 @ 3.6GHz CPU to generate 10^n digits of π for $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. All results were checked against `pi-billion.txt`. The obtained results are compared against a state-of-the-art implementation of π digits computation: y-cruncher (Alexander J. Yee, n.d.), which uses the same algorithm but is much more hardware-optimized and has some clever tricks applied.

The time includes memory allocation, computation, and writing the result to a file on disk. As one can see the resulting timings are off by a factor of 2 from a state-of-the-art implementation (Fig. 7), showing that the implementation is relatively good for the amount of work put into it. The problem can be observed to scale linearly with respect to input (Fig. 6). Our implementation seems to handle small values of n better, but this is completely irrelevant: both implementations finish under 10ms. Around $n = 10^4$ the overhead of computation overshadows side tasks such as memory allocation and file writing.

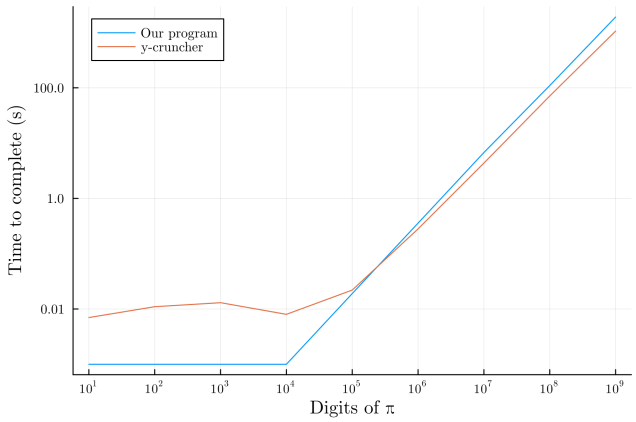

Figure 6: Runtime of our and y-cruncher program for various targets of digits of π. Log scale for the y-axis.

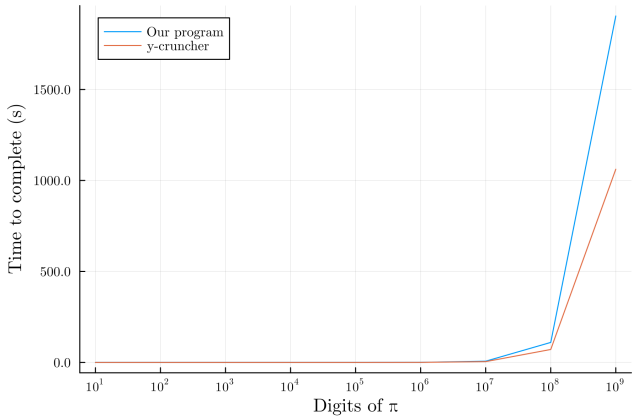

Figure 7: Runtime of our and y-cruncher program for various targets of digits of π. Log scale for the y-axis.

The program was written in C and had a requirement of working on Windows. Due to that the program is not multithreaded because of a lack of standard library support from Microsoft. All benchmarks were performed in a single-threaded environment. Another limitation of the implementation is the previously mentioned lack of support for computations exceeding RAM space and lack of distributing capabilities, all of which are available in y-cruncher.

References

Alexander J. Yee, Shigeru Kondo. n.d. "10 Trillion Digits of Pi: A Case Study of Summing Hypergeometric Series to High Precision on Multicore Systems." https://hdl.handle.net/2142/28348.

Bruno Haible, Thomas Papanikolaou. n.d. "Fast Multiprecision Evaluation of Series of Rational Numbers." https://www.ginac.de/CLN/binsplit.pdf.

Knuth, Donald E., James H. Morris Jr., and Vaughan R. Pratt. 1977. "Fast Pattern Matching in Strings." *SIAM Journal on Computing* 6 (2): 323–50. https://doi.org/10.1137/0206024.

Milla, Lorenz. 2018. "A Detailed Proof of the Chudnovsky Formula with Means of Basic Complex Analysis – Ein Ausführlicher Beweis Der Chudnovsky-Formel Mit Elementarer Funktionentheorie." arXiv. https://doi.org/10.48550/ARXIV.1809.00533.