Algorithms and Computability - Project Report

Calculating decimal expansion of π to 10^9 decimal places and pattern matching against it.

Akmalkhon Mukhiddinov
Bohdan Soproniuk
Kirill Salohka
Noman Noor

December 2022

Contents

1 Introduction **3**

2 Description of the Project **3**
- 2.1 Pi Decimal Expansion . . . 3
- 2.2 Pattern Matching . . . 3
- 2.3 File Comparison . . . 3

3 Our Solution **3**
- 3.1 Pi Decimal Expansion . . . 3
  - 3.1.1 Description . . . 3
  - 3.1.2 Parallelization . . . 8
  - 3.1.3 Tools used . . . 9
  - 3.1.4 Benchmarks - Timing . . . 9
  - 3.1.5 Testing . . . 9
- 3.2 Pattern Matching . . . 9
  - 3.2.1 Description . . . 9
- 3.3 File Comparison . . . 11
  - 3.3.1 Description . . . 11

4 Compilation Instructions **11**

5 Using the programs **11**
- 5.1 Pi Decimal Expansion . . . 11
- 5.2 Pattern Matching . . . 11
- 5.3 File Comparison . . . 12

6 References **12**

7 Licensing **12**

8 Task allocation **13**

1 Introduction

This is the documentation concerning our Algorithms and Compatibility project. The topic of the project is calculating decimal expansion of π to 10^9 decimal places and pattern matching to find the index of the first occurrence of a given input string in the calculated Pi. Our solution also includes a program for the comparison of two files to compare two different Pi text files.

2 Description of the Project

2.1 Pi Decimal Expansion

The goal of this part is to calculate the decimal expansion of Pi to the 10^9 decimal place and write the said expansion to a text file which shall be used in later stages as will be explained in the following subsections. The text written to the file should not include the decimal point "." itself.

2.2 Pattern Matching

After having computed the said decimal expansion of Pi from the prior section, we try to find the first occurrence of a given input string inside the decimal expansion and print the index of the starting point of the said occurrence.

2.3 File Comparison

Finally, the file comparison part of the project is used to ensure that two computed Pi text files have the exact same content. Not only is it useful for simply comparing files, this will also be useful in validating our files against existing Pi decimal expansions against existing reliable computations such as one computed by Massachusetts Institute of Technology.

3 Our Solution

3.1 Pi Decimal Expansion

3.1.1 Description

Choosing the algorithm: Our solution uses the Chudnovsky's algorithm with binary splitting because it's practically the fastest algorithm known. In theory, Arithmetic Geometric Mean algorithm should be faster because it doubles the number of decimal places each iteration, however It involves square roots and full precision divisions which makes it tricky to implement efficiently on computers. As such, in practice Chudnovsky algorithm is faster.

Chudnovsky Algorithm without Binary Splitting The formula itself is derived from one by Ramanujan, and is:

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

Simplifying it a bit:

$$\begin{aligned}
a &= \sum_{k=0}^{\infty}\frac{(-1)^k(6k)!}{(3k)!(k!)^3 640320^{3k}} \\
&= 1 - \frac{6\cdot 5\cdot 4}{(1)^3 640320^3} + \frac{12\cdot 11\cdot 10\cdot 9\cdot 8\cdot 7}{(2\cdot 1)^3 640320^6} - \frac{18\cdot 17\cdots 13}{(3\cdot 2\cdot 1)^3 640320^9} + \cdots \\
b &= \sum_{k=0}^{\infty}\frac{(-1)^k(6k)!k}{(3k)!(k!)^3 640320^{3k}} \\
\frac{1}{\pi} &= \frac{13591409a + 545140134b}{426880\sqrt{10005}} \\
\pi &= \frac{426880\sqrt{10005}}{13591409a + 545140134b}
\end{aligned}$$

It can be seen that each $(k+1)^{th}$ 'a' term can be gotten from the k^{th} 'a' term, and that the b terms are calculated from the a terms. As such, our calculations are simplified to:

$$\begin{aligned}
a_k &= \frac{(-1)^k(6k)!}{(3k)!(k!)^3 640320^{3k}} \\
b_k &= k\cdot a_k \\
\frac{a_k}{a_{k-1}} &= -\frac{(6k-5)(6k-4)(6k-3)(6k-2)(6k-1)6k}{3k(3k-1)(3k-2)k^3 640320^3} \\
&= -\frac{24(6k-5)(2k-1)(6k-1)}{k^3 640320^3}
\end{aligned}$$

As such, Pi after the k iterations is computed by:

$$\pi = \frac{426880\sqrt{10005}}{13591409a_k + 545140134b_k}$$

The formula here is relatively straightforward to implement as a program aside from the problem of arbitrary precision number storage, but this approach is quite slow for our need of computing decimal expansion to a large number of decimal places. As such, we decided to apply binary splitting to the Chudnovsky algorithm.

Recursive Binary Splitting: Binary splitting is a technique for speeding up numerical evaluation of many types of series with rational terms. While it requires more memory than direct term-by-term summation, but is asymptotically faster since the sizes of all occurring sub-products are reduced. Additionally, whereas the most naive evaluation scheme for a rational series uses a full-precision division for each term in the series, binary splitting requires only one final division at the target precision; this is not only faster, but conveniently eliminates rounding errors. To take full advantage of the scheme, fast multiplication algorithms such as Toom–Cook and Schönhage–Strassen must be used; with ordinary O(n2) multiplication, binary splitting may render no speedup at all or be slower.

Suppose we have a general infinite series of the following form.

$$S(0,\infty) = \frac{a_0 p_0}{b_0 q_0} + \frac{a_1 p_0 p_1}{b_1 q_0 q_1} + \frac{a_2 p_0 p_1 p_2}{b_2 q_0 q_1 q_2} + \frac{a_3 p_0 p_1 p_2 p_3}{b_3 q_0 q_1 q_2 q_3} + \cdots$$

Which means it's partial sum from some a to some b is:

$$S(a,b) = \frac{a_a p_a}{b_a q_a} + \frac{a_{a+1} p_a p_{a+1}}{b_{a+1} q_a q_{a+1}} + \frac{a_{a+2} p_a p_{a+1} p_{a+2}}{b_{a+2} q_a q_{a+1} q_{a+2}} + \cdots + \frac{a_{b-1} p_a p_{a+1} p_{a+2} \cdots p_{b-1}}{b_{b-1} q_a q_{a+1} \cdots p_{b-1}}$$

Now, let's define some functions for our convenience:

$$\begin{aligned} P(a,b) &= p_a p_{a+1} \cdots p_{b-1} \\ Q(a,b) &= q_a q_{a+1} \cdots q_{b-1} \\ B(a,b) &= b_a b_{a+1} \cdots b_{b-1} \\ T(a,b) &= B(a,b) Q(a,b) S(a,b) \end{aligned}$$

Let m be the midpoint or one element after or before the middle (making m as near to the middle of a and b will lead to the quickest calculations, but it can be anything between a and m for the proof/derivation of recursive binary splitting or if the computation time doesn't matter as much).
As such, it can be seen that:

$$\begin{aligned} P(a,b) &= P(a,m) P(b,m) \\ Q(a,b) &= Q(a,m) Q(b,m) \\ B(a,b) &= B(a,m) B(b,m) \\ T(a,b) &= B(m,b) Q(m,b) T(a,m) + B(a,m) P(a,m) T(m,b) \end{aligned}$$

The first three are trivial to notice, as for the last one, it is gotten as: (for

$a < m < b$)

$$
\begin{aligned}
&T(a,b) = B(a,b)Q(a,b)S(a,b)\\
&= B(a,b)Q(a,b)\left[\left(\frac{a_a p_a}{b_a q_a} + \frac{a_{a+1}p_a p_{a+1}}{b_{a+1}q_a q_{a+1}} + \frac{a_{a+2}p_a p_{a+1}p_{a+2}}{b_{a+2}q_a q_{a+1}q_{a+2}} + \cdots + \frac{a_{m-1}p_a p_{a+1}p_{a+2}\cdots p_{m-1}}{b_{m-1}q_a q_{a+1}\cdots p_{m-1}}\right)\right.\\
&\left. + \left(\frac{a_m p_a p_{a+1}\cdots p_m}{b_m q_a q_{a+1}\cdots q_m} + \frac{a_{m+1}p_a p_{a+1}\cdots p_{m+1}}{b_{m+1}q_a q_{a+1}\cdots q_{m+1}} + \frac{a_{m+2}p_a p_{a+1}\cdots p_{m+2}}{b_{m+2}q_a q_{a+1}\cdots q_{m+2}} + \cdots + \frac{a_{b-1}p_a p_{a+1}\cdots p_{b-1}}{b_{b-1}q_a q_{a+1}\cdots p_{b-1}}\right)\right]\\
&= B(a,b)Q(a,b)\left[\left(\frac{a_a p_a}{b_a q_a} + \frac{a_{a+1}p_a p_{a+1}}{b_{a+1}q_a q_{a+1}} + \frac{a_{a+2}p_a p_{a+1}p_{a+2}}{b_{a+2}q_a q_{a+1}q_{a+2}} + \cdots + \frac{a_{m-1}p_a p_{a+1}p_{a+2}\cdots p_{m-1}}{b_{m-1}q_a q_{a+1}\cdots p_{m-1}}\right)\right.\\
&\left. + \frac{p_a p_{a+1}\cdots p_{m-1}}{q_a q_{a+1}\cdots q_{m-1}} * \left(\frac{a_m p_m}{b_m q_m} + \frac{a_{m+1}p_m p_{m+1}}{b_{m+1}q_m q_{m+1}} + \frac{a_{m+2}p_m p_{m+1}p_{m+2}}{b_{m+2}q_m q_{m+1}q_{m+2}} + \cdots + \frac{a_{b-1}p_m p_{m+1}p_{m+2}\cdots p_{b-1}}{b_{b-1}q_m q_{m+1}\cdots p_{b-1}}\right)\right]\\
&= B(a,b)Q(a,b)\left(S(a,m) + \frac{P(a,m)}{Q(a,m)}S(m,b)\right)\\
&= B(a,b)\left[Q(a,b)S(a,m) + \frac{Q(a,b)P(a,m)}{Q(a,m)}S(m,b)\right]\\
&= B(a,b)\left[Q(a,b)S(a,m) + P(a,m)Q(m,b)S(m,b)\right]\\
&= B(a,b)Q(a,b)S(a,m) + B(a,b)P(a,m)Q(m,b)S(m,b)\\
&= B(m,b)Q(m,b)B(a,m)Q(a,m)S(a,m) + B(a,m)P(a,m)B(m,b)Q(m,b)S(m,b)\\
&where\,[Q(a,b) = Q(a,m)Q(b,m)]\,and\,[B(a,b) = B(a,m)B(b,m)]\\
&= B(m,b)Q(m,b)T(a,m) + B(a,m)P(a,m)T(m,b)\\
&where\,[T(a,b) = B(a,b)Q(a,b)S(a,b)]
\end{aligned}
$$

We can use these relations to expand the series recursively, so if we want S(0,8) then we can work out S(0,4) and S(4,8) and combine them. Likewise to calculate S(0,4) and S(4,8) we work out S(0,2), S(2,4), S(4,6), S(6,8) and combine them, and to work out those we work out S(0,1), S(1,2), S(2,3), S(3,4), S(4,5), S(5,6), S(6,7), S(7,8).We don't have to split them down any more as we know what P(a,a+1), Q(a,a+1) etc is from the definitions above.

$$
\begin{aligned}
P(a,a+1) &= p_a\\
Q(a,a+1) &= q_a\\
B(a,a+1) &= b_a\\
S(a,a+1) &= \frac{a_a p_a}{b_a q_a}\\
T(a,a+1) &= B(a,a+1)Q(a,a+1)S(a,a+1)\\
&= b_a q_a \frac{a_a p_a}{b_a q_a}\\
&= a_a p_a
\end{aligned}
$$

And when we finish computing P(0,n),Q(0,n),B(0,n), and T(0,n), we can get S(0,n) by the following formula:

$$S(0,n) = \frac{T(0,n)}{B(0,n)Q(0,n)}$$

As can be seen, this final division is the only one that is used in the whole process. This is the *"only one final division at the target precision"* referenced in the introductory paragraph of this subsection.

Chudnovsky Algorithm with Binary Splitting Due to the performance benefits of binary splitting explained at the beginning of the previous section, using binary splitting with the Chudnovsky algorithm results in significantly faster computation times compared to the plain version of Chudnovsky. As such, our application was built to calculate the decimal expansion of Pi using Binary Splitting with Chudnovsky algorithm.

The binary splitting parameters for Chudnovsky series are:

$$\begin{aligned}
p_0 &= 1 \\
p_a &= (6a-5)(2a-1)(6a-1) \\
q_0 &= 1 \\
q_a &= a^3 \cdot 640320^3/24 \\
b_a &= 1 \\
a_a &= (13591409 + 545140134a)
\end{aligned}$$

These can now be plugged back into the general equation derived in the previous section i.e.

$$S(0,n) = \frac{T(0,n)}{B(0,n)Q(0,n)}$$

where

$$\begin{aligned}
P(a,a+1) &= p_a \\
Q(a,a+1) &= q_a \\
B(a,a+1) &= b_a \\
S(a,a+1) &= \frac{a_a p_a}{b_a q_a} \\
T(a,a+1) &= B(a,a+1)Q(a,a+1)S(a,a+1) \\
&= b_a q_a \frac{a_a p_a}{b_a q_a} \\
&= a_a p_a
\end{aligned}$$

This is exactly the algorithm used in our solution.

3.1.2 Parallelization

In order to try and optimize the program/solution, we added multi-threading for the recursive binary splitting part. If you think about it, single thread recursive binary splitting is quite alike a depth-first search in binary trees. The following figure should help demonstrate the point. Clearly, in such a case, the order is:

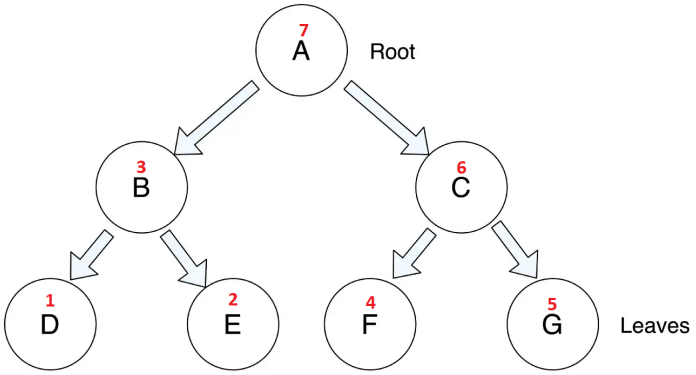

Figure 1: Representation of recursive binary splitting as a binary tree. Red numbers are the order of computation.

$A- > B- > CalculateD- > B- > CalculateE- > CalculateE- > A- > C- > CalculateF- > C- > CalculateG- > CalculateC- > CalculateA$. Also, it should be obvious a nodes at the same depth of the tree and its children are not dependant on the other nodes at that depth and their children. E.g. B,D, and E are independent of C,F, and D (and vice-versa).

As such, using our knowledge about binary trees to devise an efficient manner of parallelization/multi-threading. First of all, we notice that all the nodes at a depth k are independant of each other and hence suitable for parrellization. Therefore, the number of such threads will be the number of nodes at depth k (k=0,1,2,...), i.e. (2^k). Also, since each of these threads have to be generated concurrently, all the nodes in the tree upto that depth must also have their own threads which will just wait for all the threads beneath them to finish. Therefore, the depth k determines number of concurrent threads which will be doing the bulk of the processsing. So, the total number of threads in our application is the maximum number of nodes of a binary tree of depth k, i.e. $2^{k+1} - 1$. The number of leaf/non-dependant concurrent threads created in our program is equal to the number of threads in the computer running it.

This is how our multi-threading solution is designed, but we also create a thread for calculating the $\sqrt{10005 * 10^{n^2}} \equiv \sqrt{10005 * 10^{2n}}$ (where n is the

number of digits to calculate for Pi after the decimal point) which our implementation uses it as a scalar/shifter to make all calculations in integer form and as such we need it to not lose the required precision. While upto $k = 10^8$ digits is realtively unaffected, this should have a greater effect when computing digits $k >= 10^9$ digits.

3.1.3 Tools used

Language: C++ (C++20 standard)
Operating System tools: Windows process and thread priority setters.
External library used: GMP (self-compiled 64-bit) for efficient arbitrary precision arithmetic.
Final program compiled for x64 (64-bit) architectures to get better performance.

3.1.4 Benchmarks - Timing

Digits	Single-threaded (32bit)	Parallel (32-bit)	Parallel (64-bit)	Linux Parallel (64-bit)
10^6	1.70143	1.03556	0.528629	-
10^7	25.6792	13.5332	7.73433	-
10^8	412.302	205.908	89.8545	-
10^9	-	-	-	972.55

Time in seconds, run on Intel Core i5 8300H (4 cores, 8 threads)

The program for everything except 10^9 were tested on Windows PCs.Though the program supports Windows, there wasn't a Windows computer with enough free/available RAM (15 GBs) to test our Windows compliant version for 10^9 digits of $\pi after decimal point, and as such, we benchmarked it for that on Linux.$

3.1.5 Testing

The output of the program for 10^8 and 10^9 digits was compared to one known to be correct made available by MIT (Massachusetts Institute of Technology), and as such we verified that our results were correct.

Apart from that, on each run, the program checks if the file generated is of the correct size and prints to the terminal accordingly.

3.2 Pattern Matching

3.2.1 Description

Pattern matching is a broad class of problems in computer science, which consists of checking a given sequence of tokens for the presence of the constituents of some pattern. The match must be exact: "either it will or will not be a match." This goal differentiates pattern matching from pattern recognition.

i	0	1	2	3	4	5	6	7	8	9
W[i]	7	7	7	5	7	7	7	7	7	5
T[i]	0	1	2	0	1	2	3	3	3	4

Table 1: A visualization of a partial match table.

Given the task of creating a mapping from a natural number to its index among the digits of pi, we turn our attention to string-searching algorithms, where we try to find a place where a pattern is located within a larger string.

A naive approach is an easy but ineffective way to see where one sequence occurs inside another by checking each index one by one. Let us call the long string the "haystack" and the short one - the "needle." First, we test if there's a copy of the needle starting at the first character of the haystack. Otherwise, we check for a copy of the needle beginning at the second character of the haystack. The process continues until the match is found or the end of the haystack is reached. Usually, we only have to look at one or two characters for each wrong position to see that it is an incorrect position, so in the average case, this takes $\Omega(n + m)$ steps, where n is the length of the haystack and m is the length of the needle. In the worst case, searching for a string like "0001" in a string like "0000000001" takes $O(n * m)$ time.

Instead, we implemented the Knuth–Morris–Pratt string-searching algorithm (or KMP algorithm for short). This procedure searches for occurrences of a "word" W within a main "text string" S by observing that when a mismatch occurs, the word embodies enough information to determine where the next match could begin, thus bypassing the re-examination of previously matched characters.

Let n denote the length of S and k represent the length of W. The algorithm consists of the table-building stage and the search stage. In the beginning, we build a table to allow the algorithm not to match any character of S more than once (also known as the "failure function"). The critical observation about the linear search is that in having checked some part of the string S against an initial segment of the pattern W, we know precisely at which places a new potential match that could continue to the current position could begin before the current index. We "pre-search" the word and create a list of all possible fallback positions that bypass a maximum of hopeless characters while not sacrificing any potential matches.

A "partial match" table T indicates where we need to look for the start of a new match when a mismatch is found. Firstly, if the first character in the word W is a mismatch, we cannot backtrack and must inspect the next character. Secondly, not checking characters we already know will match results in a more efficient search.

The total complexity of the algorithm comes from the preprocessing portion and the matching portion and is equal to $O(n + k)$. These complexities are the same, no matter how many repetitive patterns are in W or S; thus, the worst-case performance is guaranteed to be $O(n + k)$.

3.3 File Comparison

3.3.1 Description

Since the decimal digits of pi are stored in a file, comparing two pi expansions is no different from comparing regular files. The shortcut we can take here is to check the sizes first. Only when they are the same does the application read data and compare characters one by one.

4 Compilation Instructions

All of the three programs we talked about in this report are present alongside their Visual Studio solution (".sln") files. As such, the process of building any of them is:

1. Using Visual Studio, open the ".sln" file in the folder containing the source code of the program.
2. (Optional, but improves performance quite a lot.) Set the configuration to "Release" and "64-bit".
3. Click on the "build" menu item on the top menu, and then click on "Build Solution".

We recommend using Visual Studio 2022 for the procedure mentioned above.

5 Using the programs

For our purpose, the programs should only be run in the order in which they appear.
It is recommended to keep all the executables in the same folder.

5.1 Pi Decimal Expansion

1. Run the PiCalc executable.
2. At the beginning, the program will ask for the number of digits to calculate.
3. At the moment, the program only supports the number of digits that can be expressed as powers of 10.
   e.g. 1 Billion = 1000000000, 100 million = 100000000, etc.

5.2 Pattern Matching

1. Run the executable.
2. Program will ask you to input min range and max range of where to look for substrings

3. After entering ranges, program will validate the input and ask again if the input was incorrect
4. If input was correct, program will start to search substrings from 0 to 9999 in the given range
5. Program will display on which substring it is currently working
6. After finding all substrings, program will output a message about successful completion
7. While it's running, the program will print to the console for reporting completion of various stages of the program.

5.3 File Comparison

1. Run the executable.
2. Program will ask you to input Pi file directory.
3. After entering filenames you will see the results

6 References

- "Chudnovsky Algorithm" - Wikipedia Page
- "Pi - Chudnovsky" by Nick Craig-Wood
- "Binary splitting method" web-page on numbers.computation.free.fr
- "Binary splitting" - Wikipedia Page
- "Knuth–Morris–Pratt algorithm" - Wikipedia Page
- "KMP PATTERN MATCH ALGORITHM" on AlgoTree.org

7 Licensing

The licensing of the third-party tools and libraries that were used while making the program:

- GMP: The GNU Multiple Precision Arithmetic Library is distributed under the dual licenses, GNU LGPL v3 and GNU GPL v2.
- MPIR: Multiple Precision Integers and Rationals is licensed under LGPL v3+

8 Task allocation

Person	**Task**
Noman Noor	Pi Calculation - Program and Documentation
Akmalkhon Mukhiddinov	Pattern Matching
Kirill Salohka	File Comparison
Bohdan Soproniuk	Documentation for Pattern Matching and File Comparison