Wilsom Prime Space Complexity Reduction

Mathematical Proof for solving wprime with lower space complexity

Version 1.0 VALDEMAR LINDBERG

Abstract

Wilsom's theorem on binary systems can be solved using the ALU on the processing units with a lower space complexity in respect to the space complexity required for computing factorial products with space complexity of $\infty(n!)$. However, the space complexity can be reduced by using common supported instruction on processing units such as integer remainder, integer multiplication, and integer addition that will yield a space complexity of $O(n)$ and $O(log_{2^32}(n))$.

Contents

1 Introduction

A prime number is an interesting topic in mathematics because of many of its properties. However, one of the challenges is to determine if an integer is prime. This paper will provide solutions for solving small primes by using both serial and parallel computation. This solution will be done with the fixed size of the registers on the ALU hardware. However, this technique can very easily be expanded by using large numbers, emulated in software.

This paper is about proving that *modular arithmetic* can be used for reducing the space complexity for computing big factorial numbers on fixed-sized computer registers on the ALU (Arithmetic logic unit). This paper will also prove that the *"Wilson theorem"* can by subdividing it into multiple factor product sequences. This can be used for solving Wilson's theorem with parallel computation, such as with *GPU general computing*.

1.1 Limitation

This paper will only cover the mathematical proof, theorems and lemma for which is required by the proofs. Where the code soluations will be presented as*pseudo code* for the parallel (2) and serial (1) as computation solution. The source code in *C* can be found at *github*, serial[**?**] and for parallel[**?**].

1.2 Purpose

This research was done in order to find a faster way for determining if a number is prime with *Wilsom's theorem*. Because the biggest problem with *Wilsom's therorem* is that it requires to compute big factorial numbers. Where computers have fixed sized registers for storing and computing the arithmetic with the integrated ALU (Arithmetic logic unit). This would otherwise require big numbers and, CPU hardware does not support this natively, which requires software emulated arithmetic operations, which would increase computation time.

2 Mathematical Background

The following subsections will cover the mathematical theory required in order to perform the proof at page 3 that will present each of the definitions, theorems, and lemmas.

2.1 Prime Number

Because The *Wilsom's Theorem* is about solving if a number is a prime. That yield that the definition of a prime number is the first definition to cover, see Definition 2.1.

Definition 2.1. *Prime Number*
A prime number *is a number that is only divisible by one and itself. That includes all-natural numbers excluding 1 and 0. See equation 2.1 for the subset for which a prime may exist.*

$$\mathbb{N}_p = \{2, 3, 4, \cdots, n-1, n\} \tag{1}$$

The reason why *one* is not a prime number is because of a composite number, see following definition 2.2.

Definition 2.2. *Composite Number*
A composite number is a non-prime number that can be expressed as a product of a unique sequence of prime factors.

$$c = p_0 p_1 p_2 \cdots p_{n-1} p_n, c \in \mathbb{N}^+ \tag{2}$$

Where c is the composite number whereas $p_i \in \mathbb{N}_p$.

For example, $120 = 2 \cdot 2 \cdot 3 \cdot 2 \cdot 2 \cdot 5$, $15 = 3 \cdot 5$ are both composite numbers by the definition 2.2. Because each factor in the composite number is a prime number. However, if number one were be included, then there would exist infinitely many expressions for a composite number could be expressed. Additionally, factoring one will never alter the value.

2.2 Modular Arithmetic

Modular arithmetic is a mathematical model for which many people have been tough in school and are using it daily. However, are commonly never explicitly informed about because is associated with the how a clock works, which many people know how to compute and use. The clock goes from $1, 2, 3, \cdots, 11, 12$ and goes back to 1 and continues like that for all eternity. *Modular arithmetic* works similar. See following definition of modular arithmetic 2.3.

Definition 2.3. *Modular Arithmetic*
Modular arithmetic is the computation of the remainder of a fraction rather than the quotation of a fraction.

$$qp + r \equiv r \pmod{q}, where q, p, r \in \mathbb{N} \tag{3}$$

If we were to express the clock with modular arithmetic based on the Definition 2.3. We would get the following expression:

$$h \pmod{12} + 1, \text{where } h \text{ is the hour.} \tag{4}$$

The adding of *one* on the right side was done in order to solve the problem that $hmod\,12$ alone will circle through the set $\{0, 1, 2, 3, \cdots, 10, 11\}$. However, By adding one will offset to each number and yield the set $\{1, 2, 3, 4, \cdots, 11, 12\}$, which the clock uses.

Modular arithmetic is also associated with the division. Because, for instance, $23 \pmod 5$ is the equivalence with $\frac{23}{5} + \frac{r}{5} = rqp + r$. This means that the modular arithmetic is the *carry* in computer terms and the *remainder* in mathematics terms, that has a significant amount of application in both computer softwares and mathematics.

An important lemma is required for the proof to work, which is the basis for how the reduced space complexity is deduced from, see the following Lemma:

Lemma 2.1. *Expand Modular Factorlabellem:modprodgen*
if $c_0 \in \mathbb{Z}$ is composite number and $p \in \mathbb{N}^+$. The composite number with modular arithmetic 2.3 can be expression as followed:

$$\begin{aligned} &c_0 \pmod p = (f_0 \cdot f_1 \cdot f_2 \cdots f_{n-2} \cdot f_{n-1}) \pmod p \\ &\qquad \Longleftrightarrow \\ &c_0 \pmod p = f_0 \pmod p \cdot f_1 \pmod p \cdot f_2 \pmod p \cdots f_{n-2} \pmod p \cdot f_{n-1} \pmod p \end{aligned} \tag{5}$$

2.3 Wilsom's Theorem

Theorem 2.2. *Wilsom's Theorem*
A number $p \in \mathbb{N}$ can determine if its a prime number by using the following equation:

$$(p-1)! + 1 \equiv 0 \pmod p \tag{6}$$

Where if no remainders defined by the modular arithmetic Definition 2.3 implies that p is a prime number.

By the theorem in 2.2, this would mean that $(p-1)! + 1$ *must* be divisible by p in order for the p to be prime accordingly the modular arithmetic definitions 2.3. It can also be expressed as followed:

$$p|(p-1)! + 1 \;\Rightarrow\; \text{p is prime.} \tag{7}$$

3 Proof

The proof section contains two proofs. Where the first one is about the *space complexity*, and the second for proving that the *Wilsom's theorem* can be solved in parallel.

3.1 Space Complexity

The following proof is about proving that *modular arithmetic* 2.2 can be used in order to subdivide the factorial expression in order to reduce the size of the number, this is because a factorial number increases in size incredibly quickly with $\theta(n!)$. The straight forward approach for solving n being prime is as followed.

Corollary 3.0.1. *Wilsom prime can be simplified as a product of all integers from $p-1$ and additions with one, based on Lemma* **??**.

$$\left(\prod_{i=1}^{p-1} i\right) + 1 \equiv 0 \pmod{p} \tag{8}$$

This equation has the problem that the product will increase in size rapidly. This becomes a problem for most CPU (central computation unit), because the registers responsible for storing storing the values as a binary representation of the whole numbers. Because as of the time of this paper CPU's has support for *ALU*(Arithmetic logic unit) for up to 128 bit numbers. That is say 10^{38}. However this register would been been overflowed, using equation 8 before p reaches 36. That implies it would only be able to solve for $p < 35$.

Proof. Given that 2.2 is true. We can use modular definition 2.3 to rewrite *Wilsom's theorem* to the following, see equation 9.

$$p = ap + r \,(\text{mod } p) \tag{9}$$

$$\begin{aligned} \theta(p) = \Pi_{i=1}^{p} i \mod (p) \equiv \\ p_1 mod(p) \cdot p_2 mod(p) \cdot p_3 mod(p) \cdot \cdots \cdot \end{aligned} \tag{10}$$

$$x \equiv y mod\, z, x, c, z \in \mathbb{N}_0$$
$$x \equiv (y + kz) mod\, z, (p-1)! + 1 \equiv 0 mod\, p (p-1)! + 1 \equiv (0 + kp) mod\, p \equiv (p-1)! + 1 \equiv 0 mod\, p \tag{11}$$

□

3.2 Parallel

The following proof is regarding enabling parallel computation when solving if $n \in \mathbb{N}$ is prime with *Wilsom's Theorem* 2.2. This is done by subdividing products into multiple sub-products.

$$e_1^1 \tag{12}$$

3.3 Time Complexity

3.3.1 Serial

The serial time computation is the most straight forward. Because for any $p \in N^+$ It will have to perform at least $p-1$ iteration, as stated in Theorem 2.2. This will require at least a multiple, modular operation and addition for each iteration. Thus the time complexity can be derived as $\theta(n)$.

3.3.2 Parallel

The parallel time computation is not as straight forward as the serial complexity. Because of the two phases in its process and an unknown variable of a number of the subregion. Additional subregions will increase the time complexity defined by the Serial 3.3.1. Because each subregion has to merge in the second phase. Furthermore, the number of subregions that be computed at the same time may vary. However, with the assumption that all subregions are computed simultaneously.

Phase 1 time complexity yields $\frac{n}{s}$ because each region are computed simultaneously. Additionally, $\frac{n}{s}$ is the number of elements that have to be computed with serial on each parallel computation. Thus time complexity is $\theta(\frac{n}{s})$

Phase 2 time complexity yields to s since it is the number of elements that have to be processed with the serial computation.

The final time complexity yields: $\theta(\frac{n}{s} + s)$

4 Result

The following section contains the result for the time complexity and space complexity achieved based on the proof Section 3 at page 4.

Type	Time complexity	Space Complexity
Parallel - Reduced	$\theta(\frac{n}{s} + s)$	$\theta(n)$
Serial - Reduced	$\theta(n)$	$\theta(log_{2^{32}}(n))$
Serial - Straight	$\theta(n)$	$\theta(n!)$

Table 1: Time complexity

The *Straight* refers to the straight forward solution for solving *Wilsom's Theorem*. Whereas *Reduces* refers to the solution presented in this paper.

5 Future work

Future work is to solve for bigger numbers. Where the number has to be emulated in software rather than using hardware instruction. This can perhaps be resolved by using a *FPGA* (Field Programmable Gate Array) for creating bigger hardware register for performing addition, multiplication, and division with low latency and potential with parallel processing.

6 Pseudo Code

The following section covers the pseudo for implementing the *Wilsom's thereom* solution for both in serial and parallel, see following sections.

6.1 Serial Computation

The serial computation is defined in a single function on a single thread. Where if the return value is equal to zero implies p was a prime, otherwise non-prime number. See following Algorithm 1.

Algorithm 1: Serial Computation.

```
1  input: unsigned p
2  begin
3          x ← 1
4          n ← 1
5          for n < p − 1
6                  x ← x ∗ n
7                  x ← x % p
8          end
9          return (x + 1) % p
10 end
```

6.2 Parallel Computation

The parallel computation solution requires additional code. However, the time complexity is much greater than the serial version for bigger numbers, see the result in table 1 at page 6. The algorithm consists of two phases. First, the factor sequence is divided into multiple sub-product sequences. Next, each sub-factor sequence is computed individually into a single product. Finally, once all sub-sequences products have been computed. The final product is computed by multiply all the sub-products by using the pseudo Algorithm 1.

Algorithm 2: Wilsom's Prime - Parallel Computation.

```
1  input: unsigned p, unsigned t
2  begin
3          n ← p / t
4  end
5  input: unsigned p, unsigned e
6  begin
7          x ← 1
8          n ← p
9          for n < p + e
10                 x ← x ∗ n
11                 x ← x % p
12         end
13         return (x + 1) % p
14 end
```