Theoretical analysis: Experimental evidence of the Collatz conjecture

Student: Guillaume Leclerc (224338)

Specification of the program

In this project we want to determine the number of steps it takes to go from a number to 1 using the Syracuse conjecture (see project description).

The output of the program will consist in a list of integer, the number of steps to go from the starting point to 1. The program will compute this value for each number starting at 2 and up to T the first and only argument given to the program.

Here is an example of a call to the program:

```
$ ./a.out 10

1
7
2
5
8
16
3
19
6
```

Preliminary

Before we can do a proper analysis we need to do some empirical tests. Indeed for each number we don't know in advance how many steps it will take before we reach 1. So we can not really determine the algorithmic complexity of the algorithm in advance. For this analysis we will use the naive sequencial approach to determine the number of steps. For example to reach 1 starting from 3 it take 7 steps to reach 1 :

```
3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
```

Using the very naive approach we computed the average number of steps from 10^1 to 10^8. Let's note $f(x)$ the average number of steps per integer to compute from 1 to 10^x. For example $f(1) = 7.44$ because as you can see in the example we gave before, the average of the values is 7.44. Here are the values we found for $f(1)$ to $f(8)$ (It takes too much time on a single machine to compute more).

```
7.4444444444444
31.737373737374
59.601601601602
84.975097509751
107.53947539475
131.43455543456
155.27249862725
179.23493762235
```

As you might remark, the values are almost perfectly linear. It was not expected but it will be very convinient for our analysis. As we are in exponential scale we can deduce that the number of steps for each starting point is logarithmic. To be more precise here is the approximate formula :

$$f(x) \simeq 7.44 + 24.6(x - 1)$$

We now have the number of steps of the naive algorithm :

$$T \times f(\log_{10}(T)) = \mathcal{O}(T \log(T))$$

The Euristic

A simple (yet powerfull) euristic for this problem is to apply a time-memory tradeof. We can save for each number we already computed the number of iterations before reaching 1.

For example we compute the number of steps to go from 5 to 1 it take 5 steps, we memorize it. When we want to comput starting at 10. We apply the recurence once and we find 5 (because $10/2 = 5$) we can deduce that the number of steps from 10 to 1 is $1 + 5 = 6$. And it took only 2 iterations instead of 6 for the naive algorithm. The problem to evaluate this new algorithm is that we cannot predict how many iterations it will take before we reach a value we already know.

To overcome this we also did an empirical approach to estimate this value. We note $g(x)$ the number of **iterations** it took with this new algorithm. x goes also from 1 to 8.

```
4
9.1010101010101
6.3833833833834
6.1842184218422
6.2091520915209
6.2262642262642
6.2359355235936
6.2389850023899
```

As we can see it seems that the values of $g(x)$ are converging to a value near 6.24. This is great because we can assume that with this algorithm the amortized complexity for each integer is constant. We now have the complexity for the new algorithm:

$$T \times g(log_{10}(T)) = \mathcal{O}(T)$$

This is great because we got rid of the $log_{10}(T)$ term to reach linear complexity. An important thing to note is that the constant of our algorithm is quite small too (way better than the constant of the naive algorithm).

Distributing the improved algorithm

The naive algorithm is very easy to parallelise because there is no shared state between the iteration. On the other side in the improved algorithm we memorize the previous values. The problem is that nodes cannot share all their memory. If we distribute the algorithm, at some point precomputed values will not be available we needed. We had to evaluate the how the algorithm behaves when only part of memorized values are available.

Here are the number of **iterations** needed to compute the values from 10^1 to 10^8 if only 50% of the precomputed values are availables:

```
7.6666666666667
11.058823529412
7.7884231536926
8.0161967606479
8.1495170096598
8.1739256521487
8.1845663630867
8.1816952563661
```

And here are the result if only 1% of the values are availbles:

```
7.4444444444444
31.737373737374
54.368314833502
40.619634380366
41.315612973606
41.427014720187
42.121434533188
42.143153897544
```

As the experiment shows, it seems that the value is still constant, Only the the constant change. This is great for us because it means we still have a linear algorithm even if all the memory is not shared between nodes.

Timings analysis

Let define the 3 parameters of our models:

N : The number of computation nodes

M : The number of rounds

T : The target (The number of series we want to compute)

We will cut the interval we want to compute into $M \times N$ sub-intervals named τ_1 to $\tau_{M \times N}$. In our model we will assume the the work associated to a sub-intervals do not deviate too much from the average case. We will note $t = \frac{T}{N \times M}$ the number of series to compute in a given sub-interval.

In round r each node (of id n) will compute $\tau_{n+r \times N}$. And between each computation round, rach node share the new values he found so the others can use it to shorten their computation time.

Here are some timing diagrams for different parameters.

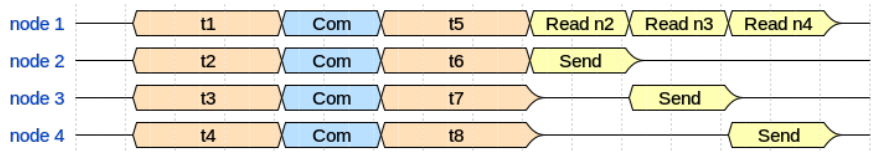

Figure 1: $N = 4, M = 2$

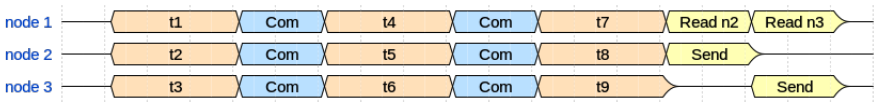

Figure 2: $N = 3, M = 3$

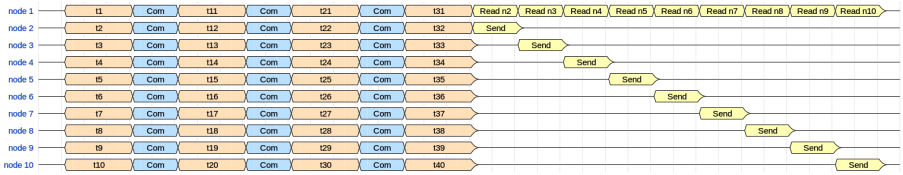

Figure 3: $N = 10, M = 5$

We could come up with a more clever way of scheduling work (by filling the blank under the send "diagonal"). But it seems the merging part of the algorithm will not take a lot of time so it might be an overoptimisation. We might reconsider it after if the expermientations shows that this assumption is wrong.

As we can see on the timing diagram the critical path is clearly a straight line on node 1.

We will now try to estimate the speedup using this model. The time to solve the problem with the optimized sequential algorithm is $W_{seq} = C_1 \times T$ for some C_1

The time to solve a sub-interval(red block in the timing diagram) assuming we have enough precomputed values available is $W_{si} = C_2 \times t$. In average we have $C_2 > C_1$.

In the communication part, each node has to send the new values it precomputed during the previous round to all other nodes. We know we can't have more new values than the number of the iterations of the algorithm we can therefore deduce that each broadcast can be done in $C_3 \times \log(N) \times t$ for some C_3. We need to do this for each node so the communication part can be done in $W_{com} = C_3 \times N \log(N) \times t$.

In the last part (yellow in the timing diagram), each node has to send his results. But it does not need to send it all because most of it was already sent during the previous communication rounds. It only has to send exactly t values. It can be done in time $C_4 \times t$ for some C_4. The whole merging phase can be done in $W_{merge} = C_4 N t$.

We can now compute the length of the critical path (W_{cp}).

$$\begin{aligned}
W_{cp} &= M \times W_{si} + (M-1)W_{com} + W_{merge} \\
&= MC_2 t + (M-1)tN\log(N)C_3 + NtC_4 \\
&= MC_2 \frac{T}{NM} + (M-1)\frac{T}{NM}N\log(N)C_3 + N\frac{T}{NM}C_4 \\
&= \frac{C_2 T}{N} + (M-1)\frac{T}{M}\log(N)C_3 + \frac{T}{M}C_4
\end{aligned}$$

To simplifiy our understanding instead of computing W_{cp} we will compute an upper bound for it. Let's consider $C_5 = \max(C_3, C_4)$

$$\begin{aligned}
W_{cp} &< \frac{C_2 T}{N} + \frac{T}{M}\left((M-1)\log(N)C_5 + C_5\right) \\
&< \frac{C_2 T}{N} + \frac{T}{M}\left((M-1)\log(N)C_5 + C_5\log(N)\right) \text{because } N \geq 1 \\
&= \frac{C_2 T}{N} + \frac{T}{M}M\log(N)C_5 \\
&= \frac{C_2 T}{N} + T\log(N)C_5 \\
&= T\left(\frac{C_2}{N} + \log(N)C_5\right)
\end{aligned}$$

Let's compute the speedup:

$$
\begin{aligned}
S_p &= \frac{W_{seq}}{W_{cp}} \\
&> \frac{TC_1}{T\left(\frac{C_2}{N} + \log(N)C_5\right)} \\
&= \frac{C_1}{\frac{C_2}{N} + \log(N)C_5} \\
&= \frac{NC_1}{C_2 + N\log(N)C_5}
\end{aligned}
$$

As we can see the limit of our speedup when N reach ∞ is 0. But for some values of the constant we can get a positive speedup for some values of N. We can split our analysis into 3 cases.

- $C_5 \sim C_2$: The speedup is always decreasing, and less than 1.
- $C_5 < C_2$: The speedup grows until it reach a global maximum and the decrease to 0.
- $C_5 \ll C_2$: The speedup is almost linear and the slope of the line is C_1/C_2

You can see the 3 cases on Figure 4.

Estimating the values of the constants

In order to predict if there will be a speedup in real life we need to have an rough idea of C_1, C_2 and C_5.

From experiments on the sequential program we found that $C_1 \simeq C_2 \simeq 3 \times 10^{-8}$

If we suppose we have an Intel 40Gb/s QDR IB network, according to the mellanox website we would have a throughput of 3.2GB/s. C_5 roughly represent the time required to send an integer. Our integers are encoded with 32bits. We can estimate $C_5 \sim \frac{32}{3.2 \times 2^{30}} = 9.3 \times 10^{-9}$

With this constants the best speedup we could acheive is 1.5 with 3 nodes

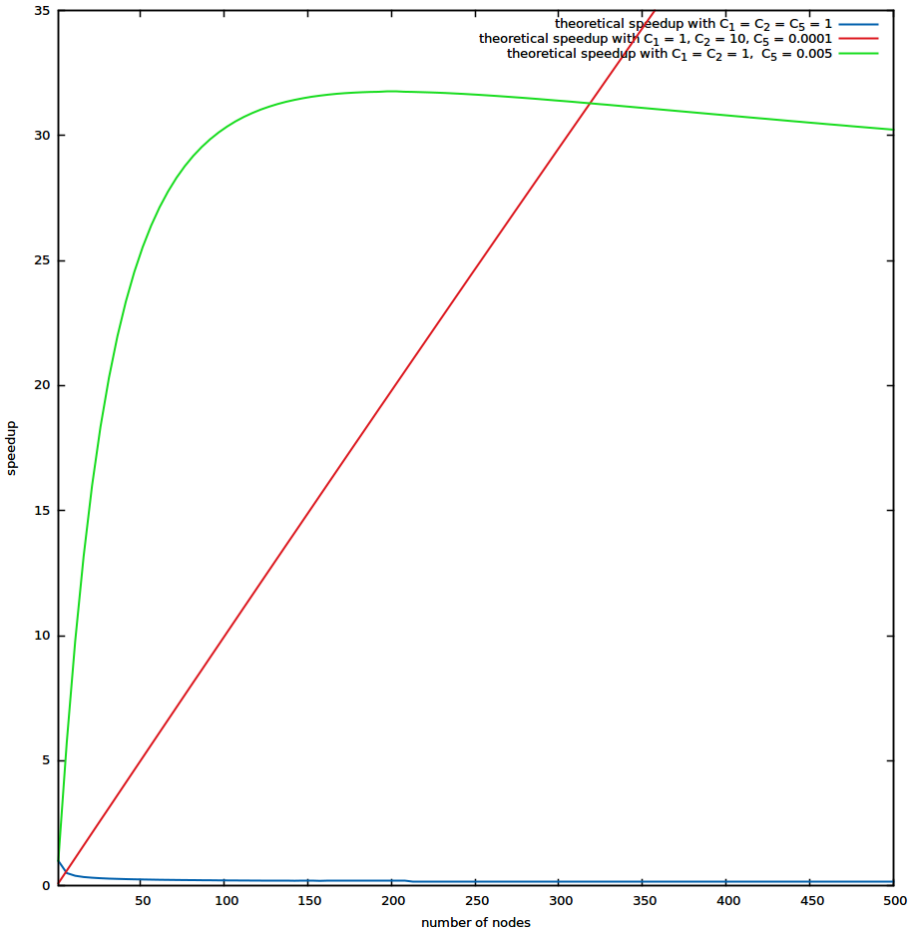

Figure 4: Theoretical simulations of the speedup