CS 4414: Systems Programming **(Due: 12/09/24)**

Homework 4 Part 2 - RSA Optimization

Instructor: Ken Birman *Name:* Nicholas Andersen, *Netid:* na534

With the most basic RSA encryption and decryption program using Bignum from part 1 we notice that it is very slow and wouldn't be very practical in real world use. Therefore, we need to optimize the program and leverage more of the computing power by analyzing the bottlenecks of the program to speed up the program. We can use GProf to see where the computer is spending most of the time and then optimize from there.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 87.58      1.34     1.34  1410323     0.00     0.00  Bignum::operator-(Bignum const&)
  6.54      1.44     0.10     3062     0.00     0.00  Bignum::operator*(Bignum const&)
  3.92      1.50     0.06  8671140     0.00     0.00  void std::vector<int, std::allocator<int> >::_M_realloc_insert<
  1.31      1.52     0.02                             _init
  0.65      1.53     0.01     2554     0.00     0.00  Bignum::operator/(Bignum const&)
  0.00      1.53     0.00     3062     0.00     0.00  std::vector<int, std::allocator<int> >::_M_fill_insert(__gnu_cx
```

Question 3 answered here

Since the program is so slow, it would be nice to get some guaranteed speedup for the larger bodies of text. Hence, it was obvious that after getting a working program, we needed to make use of multi-threading. First, we needed to look at code that is independent of other iterations. Considering that we encrypt and decrypt line by line without the encryption or decryption of a preceding line affecting the next; this was an obvious opportunity to parallelize (embarrassingly parallel). This also significantly speeds up the optimization/development process as there is a lot less wait time when testing. After finding the critical sections and establishing a maximum thread count we see the increase in performance:

Table 1: Five line encryption and decryption using multi-threading

Process time	**Time (s)**
real	4.300
user	19.758

Question 1 answered here

Onto the **results from GProf**. On the both the encryption and decryption the majority of the time is spent on the *Bignum::operator-* and *Bignum::operator** function. This is largely expected, but what is interesting is that **std::vector** operations such as memory reallocation (_M_realloc_insert) is up there too, and it was the most called function in the program. We also see that *Bignum::operator/* takes is fourth (from research '_init' is used for profiling and doesn't count). This means that there is a significant performance hit with the way our vectors are being handled. Specifically, the front insertions are causing memory to be reallocated constantly. **std::vector** insertion operations are very costly because we need to shift every value of the vector in memory to make space for the value we are pushing into the front. This takes O(n) time. The first method used to try and fix this was to pre-allocate memory for the vectors as we know that the maximum number of digits a number can have with this implementation of RSA is 154 digits. We also know that with *Bignum::operator-* the result can never be negative and therefore the maximum digits is the number of digits of the result is the number of digits of the object we are subtracting from. We can pre-allocate this memory. With larger texts, this saw an immediate increase in performance:

Table 2: Average times for encryption and decryption

Number of lines	**W/o preallocation (s)**	**W/ preallocation (s)**
1	2.823	2.561
5	4.271	3.136
25	16.538	12.276
50	32.644	24.628

Note that in the data above, when running with fewer number of lines, the improvement isn't as large. This is clearly because when we only have 1 line we run on 1 thread. This means the performance increase we see with one line is the flat performance gain without parallelization.

This was good, but seeing in GProf that the majority of time (84.33%) was spent in *Bignum::operator-* and again vector memory operations were called a lot, it made sense to try extract some more performance there. After redesigning the **Bignum** minus function (by building the vector in reverse) we are able to fix the large performance drop caused by the vector insertion operations, and this is the result:

Table 3: Average times for encryption and decryption

Number of lines	**Front vector insertion (s)**	**Back vector insertion (s)**
1	2.561	1.134
5	3.136	1.467
25	12.276	5.631
50	24.628	11.547

Question 2 answered here

This is a really good performance increase. We can rinse and repeat with GProf to keep finding further ways to optimize. After redesigning the minus operator function, GProf is still telling us that 80.53% of the time is spent on the minus operator function. We can also see that there are 4227321 calls of this function. For future optimization it might be a good idea to look at the algorithm being used and if there is a way to reduce the amount of calls to the function to increase the performance. For example one of the reasons *operator** and *operator/* were called so much was due to the way *operator%* was written. After changing the algorithm, we saw a decrease from 8435 to 3199 and from 9367 to 4176 calls for *operator/* and *operator** respectively (ran on 5 lines of poetry). This also reduced the percentage of time used on *operator/* by 7.92%.

Before (left) and after (right) modulo operator change :

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 79.08      3.10     3.10  4621095     0.00     0.00  Bignum::operator-(Bignum const&)
 17.09      3.77     0.67     9367     0.07     0.07  Bignum::operator*(Bignum const&)
  2.30      3.86     0.09  4746002     0.00     0.00  std::vector<int, std::allocator<
  0.77      3.89     0.03     8435     0.00     0.38  Bignum::operator/(Bignum const&)
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 86.69      2.93     2.93  4114810     0.00     0.00  Bignum::operator-(Bignum const&)
  9.17      3.24     0.31     4176     0.00     0.00  Bignum::operator*(Bignum const&)
  2.07      3.31     0.07  4356158     0.00     0.00  std::vector<int, std::allocator<i
  1.18      3.35     0.04     4235     0.00     0.00  Bignum::operator%(Bignum const&)
  0.59      3.37     0.02     3199     0.00     0.00  Bignum::operator/(Bignum const&)
```

In conclusion, we've seen an increase in performance of almost 300% at the worst case (single thread performance). For future optimizations as mentioned we could look at the algorithm to reduce the number of times the minus operator function is called for example. Additionally, we could also look at threading the chunks of each line for encryption and then a thread per line of encrypted data for decryption (because 2 encrypted lines = 1 decrypted line).