Computing with sparse integers

MIDN 1/C Matthew J.P. Yates

Advisor: Asst. Prof. Daniel S. Roche
Computer Science Department
United States Naval Academy

December 17, 2012

Abstract

We explore some algorithmic aspects of arithmetic with *sparse integers*. numbers in which only the nonzero bits are explicitly represented. The *sparse signed digit* representation is proposed as a good balance between expressiveness and speed, and under this representation we present algorithms for basic arithmetic including multiplication, division with remainder, and modular exponentiation. Finally, a high-performance C library implementation of these routines is examined and compared to existing approaches.

1 Introduction

Traditionaly Intergers are held with an array of bits. At each position n in the array the bit, ethier a 0 or 1, acts a coeffient for 2^n. This is repeated for every position to convert to the base 10 version of the number. This approch has a limitation in how large of an number can be stored. The 32 bit unsigned integer can represent up to 8589934591 before every combination of 0's and 1's for each of the 32 bits has been used. This maximum may be large enougth for many applications, but in certain feilds, such as Cryptography increbly large integers that exceed normal 32, 64, or even 128 bit integers are required. In these instances not even a largest primitive types can be trusted to not overflow in the process of computing an answer. The solution to this is to use non primitive numbers data types for integers, such as a BigInt. A BigInt stores arbitrarily large numbers using an ever expanding buffer to hold the bits of any number for which the computer has enough disk space to hold. One shortcoming with the BigInt approach is that they hold every single bit, even if the number may have 99% of its bits as 0's. This sort of number is considered to be sparse. The sparse number may be held more efficiently if only the locations of where the set bits exist. Sparse Integers are a way of holding certain incredibly large numbers in relatively little memory space when compared to traditional implementations. With

this conversion memory space need for huge numbers can greatly reduced, and in for some mathematical operations, the Sparse Representation is faster than the traditional storage methods.

(Elaborate more here.) Various algorithms for long integer arithmetic are covered extensively in Brent and Zimmermann [2010]. The sparse signed digit representation is based on the non-adjacent form (NAF) first presented by Reitwiesner [1960]. Vuillemin [2009] recently looked at a completely different way of representing sparse integers. Sparse integers have been studied in the context of cryptography by Banks and Shparlinski [2002], and are specified in standards such as NIST [2009].

2 Signed digit representation

The signed representation works with a series of signs, differences between positions, and size of the highest set bit.

For exapmle concider the following base 10 number 34359738240

1. base 10 34359738240
2. base 2 11111111111111111111111110000000
3. base 2 signed ++++++++++++++++++++++++++++++0000000
4. base 2 after canceling +0000000000000000000000000000-000000
5. sparse signed positions $+35 - 7$
6. sparse signed difference $+28 - 7$
7. with the top position $+28 - 7, 35$

34359738240 would need to be represented with at least 5 bytes if held in traditional form, for it could not be allocated a partial byte. Where as in in singed sparse from it needs at only the two signed differences, and the highest position. These can be held with three C chars and and two C bools. Which with a vector to hold the bools both could be represented in one char. In total the sparse representation would take take 4 bytes to represent the number 34359738240. The space savings are amplified as the size of the number increases.

3 Arithmetic

In addition to the obvious savings in terms of space complexity by only storing a few bytes to reprsent numbers that could require kilobytes or more for a traditional bigint represetanation, some arthmetic operations will be faster than the bigInt equilvelent.

3.1 Rules for Operands

Whenever numbers are passed to signed sparse arithmetic operations, our model assumes that lists of bits are ordered by size and that they are the sparsest possible representations of their numbers with no repeated bits. The modle also assumes that every bit is given a sign.

3.2 Carry Logic

The advantage of signed bits is that negative bits allow for many more numbers to represented in a sparse form and that many small operations will not case a sudden increase in density of the number. The heuristic for sparsity is that the most sparse representation of any number will have no two consecutive set bits.

Two same signed bits at the same position will lead to a new bit being placed one position higher with the same sign as the two matching bits. The two matching bits will be removed. Two bits with matching positions but not matching signs will be both removed. If one bit is one position lower than the other and they share signs then the lower bit will have its sign flipped and the higher bit will have its position incremented by one. If two bits are one apart in position and are different signs then the higher placed bit is removed and the lower bit has its sign flipped.

3.3 Addition and subtraction

Addition and subtraction are fundamentally achieved by combining two lists of signed bits by a merge sort. As the lists are combined the carry logic is applied for each bit leaving the result in its most sparse and in sorted order. The difference between subtraction and addition is that the subtraction has the sign of each bit flipped before combining the two lists.

3.4 Multiplication

Multiplication is solved by a summation of the smaller of the two numbers with each iteration having been shifted to the left by a signed bit of the larger number.

3.5 Division with remainder

Division is essentially solved through repeated subtraction of two sparse numbers. At each iteration of subtraction the divisor is multiplied by the difference between the difference between the position highest bit of the numerator and the the position of the highest bit in the divisor. Then the modulo operation is esentially division without any concern with how many times the denominator was subtracted from the numerator.

Algorithm 1: The carry Algorithm

Input: Sparse Int a and new set position n such that $n >= a[-1]$
Output: Sparse integer b that is a with the new position appended
$b \leftarrow a$;
if $b[-1] = n$ **then**
    if *Signs of* $a[-1]$ *and* n *are the same* **then**
        $a[-1] \leftarrow a[-1] + 1$;
    else
        popback(a);
    return;
if $b[-1] + 1 = n$ **then**
    if *Signs of* $a[-1]$ *and* n *are the same* **then**
        $a[-1] \leftarrow a[-1] * -1$;
        flip the sign of the last element pushback(a,$n + 1$);
        add the new position + 1
    else
        $a[-1] \leftarrow a[-1] * -1$;
        flip the sign of the last element and do not add a new bit
    return;
pushback(a,n);
if the other steps did not return than there is no carry and the new position is just added

Algorithm 2: Addition Algorithm

Input: Two sparse integers a and b
Output: Sparse integer c that is equal to $a + b$
$i, j \leftarrow 0$;
while $i < \#a$ *and* $j < \#b$ **do**
    if $a[i] < b[j]$ **then**
        $c \leftarrow c + a[i]$ with Carry Logic ;
        $i \leftarrow i + 1$
    if $a[i] > b[j]$ **then**
        $c \leftarrow c + b[j]$ with Carry Logic ;
        $j \leftarrow j + 1$
    if $a[i] = b[j]$ **then**
        if *Signs of* $a[i]$ *and* $b[i]$ *are the same* **then**
            $c \leftarrow c + 2a[i]$ with Carry Logic;
        $i \leftarrow i + 1$;
        $j \leftarrow j + 1$

Algorithm 3: Multiplication Algorithm

Input: Two sparse integers a and b
Output: Sparse integer c that is equal to $a * b$
if $\#a > \#b$ **then**
  $long \leftarrow a$;
  $short \leftarrow b$;
else
  $long \leftarrow a$;
  $short \leftarrow b$;
$i, j, c \leftarrow 0$;
while $i < \#long$ **do**
  $c \leftarrow c + short << long[i]$;
  $i \leftarrow i + 1$

Algorithm 4: Division Algorithm

Input: Two positive sparse integers a and b
Output: Sparse integer c that is equal to a/b
$i, j, c \leftarrow 0$;
$numerator \leftarrow a$ **while** $numerator > b$ **do**
  $c \leftarrow c + numerator[-1] - b[-1]$
  $numerator \leftarrow numerator - b << (numerator[-1] - b[-1])$;
if $numerator! = 0$ **then**
  $c \leftarrow c - 1$

Algorithm 5: Modulo Algorithm

Input: Two positive sparse integers a and b
Output: Sparse integer c that is equal to $a\%b$
$i, \leftarrow 0$;
$c \leftarrow a$ **while** $numerator > b$ **do**
  $c \leftarrow c - b << (c[-1] - b[-1])$;

Algorithm 6: Modular Exponation Algorithm

Input: Three sparse integers a, b, and c
Output: Sparse integer d that is equal to $a^b\%c$
$i, j \leftarrow 0$;
$d \leftarrow 1$;
$base \leftarrow b$ $seenNegative \leftarrow false$ **while** $i < \#b$ **do**
  $k \leftarrow 0$ **while** $k < b[i] - j$ **do**
    $base \leftarrow base * base\%c$ $k \leftarrow k + 1$ **if** $seenNegative$ **then**
      $d \leftarrow d * base\%c$
  if $seenNegative$ **then**
    $d \leftarrow d * base\%c$
  $seenNegative \leftarrow !signOf(b[i])$ $i \leftarrow i + 1$

3.6 Modular exponentiation

Modular exponentiation follow the classic algorithm with the variation of having to perform logical checks during the iteration to convert the number to positive signed only.

4 Implementation

The Spares Int data structure is made using C++ class with two template arguments. The template argument S is used for setting the data structure used to hold the difference between two positions. The other template Argument is T for the data type used to hold the position of the highest set bit. The differences between positions and the signs of each position are held in a vector from STL. The signs are held as bools for a vector of bools provides a signifigant space savings.

5 Time Trails

Tests where run on how many times basic mathmatical operations where run per second. The horizontal axis represents how large the highest set bit is in the traitional binary represenation of the number; a sample was drawn from every 1000. The vertical axis represents how many bits where in the number; a sample was drawn every 5. Each point in the each plot had three randomly generated pairs of operands that meet the highest bit and number of bits requirements. Addition, Subtraction, Multiplaction, and explonential modulus all use operands with equal input pramaters. Divison and Modulus give the denominator half the number of set bits as the numerator. The green represents areas where Sparse Ints were able to perform more operations per second. The red show areas where

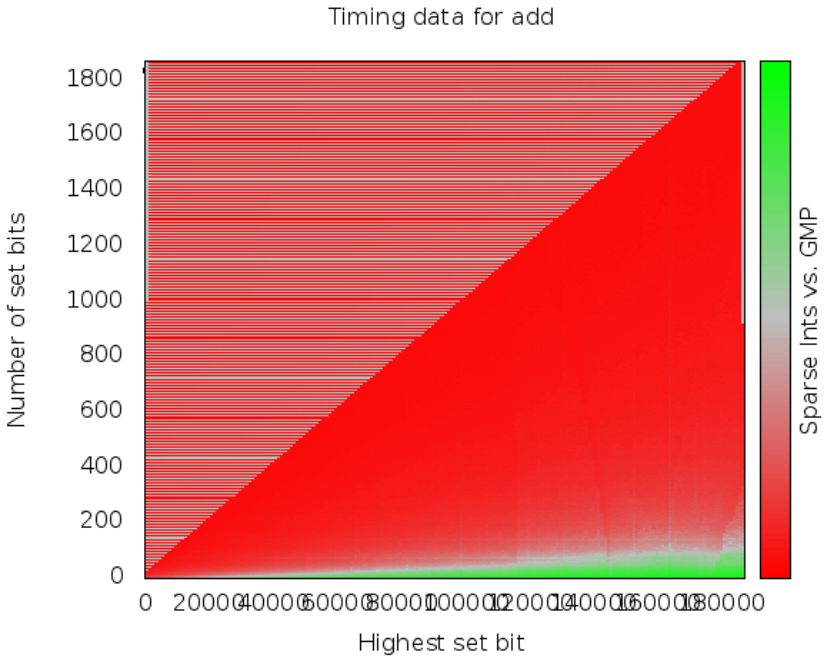

Figure 1: *Sparse Int addition compared to GMP Addition*

GMP was able to do more. The exponential modulus graph made with same paramters as the graphs above, save that both axis represent sampling at every 50.

textheight=9in

6 Conclusion

It is apperant that the GMP implementation for now is faster for a large varitey of cases. However it is important to note that in any case that is a tie, the gray colored areas, that Sparse Ints are using far less space to represent the same number. Using less space would mean more space in main memory may be saved for other aspects of computation.

7 Future Work

- Work on decreasing the time complexity of the algorithms
- Expreriment with using thread is the number as large enough gaps in positions.
- Experiment with using a heap in multiplication to hold the result as it builds.

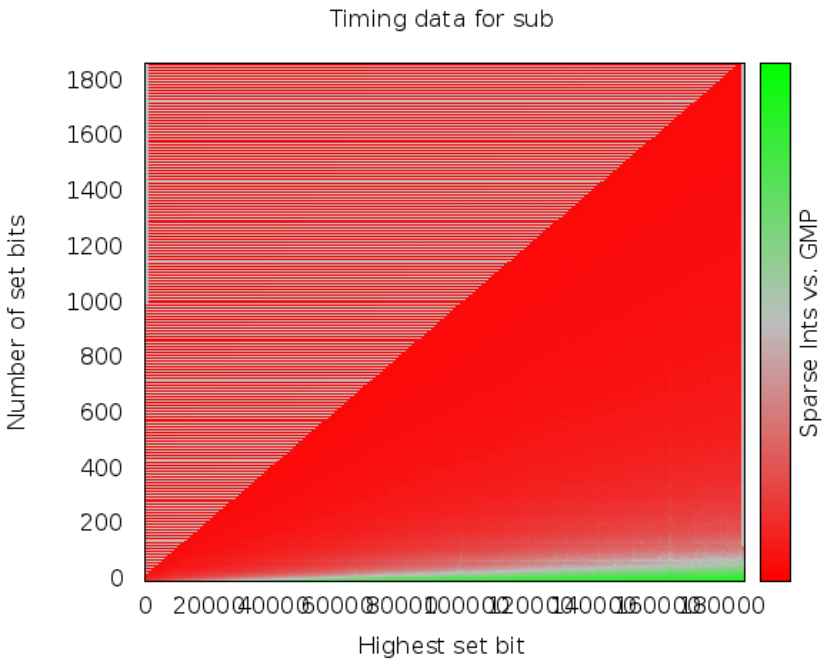

Figure 2: *Sparse Int subtraction compared to GMP subtraction*

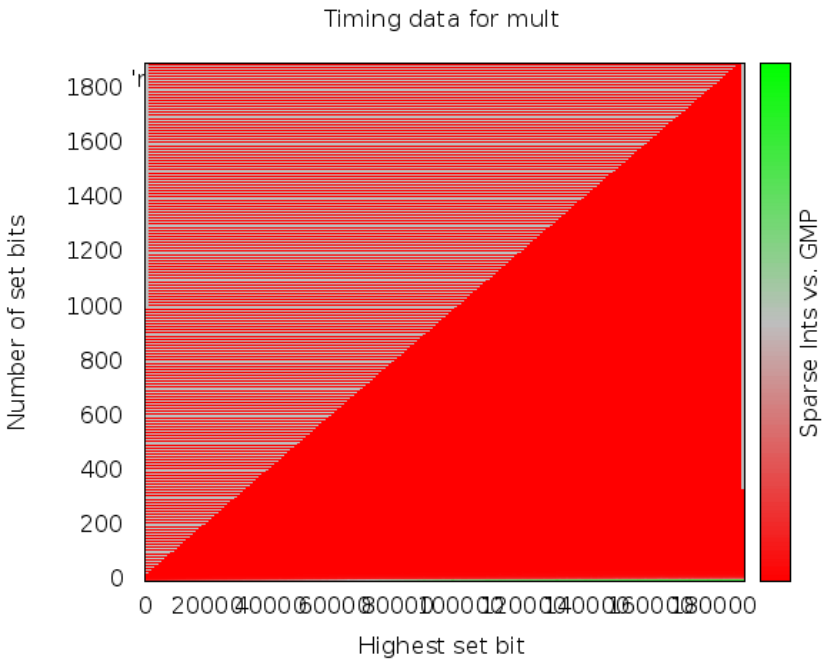

Figure 3: *Sparse Int multiplation compared to GMP multiplaction*

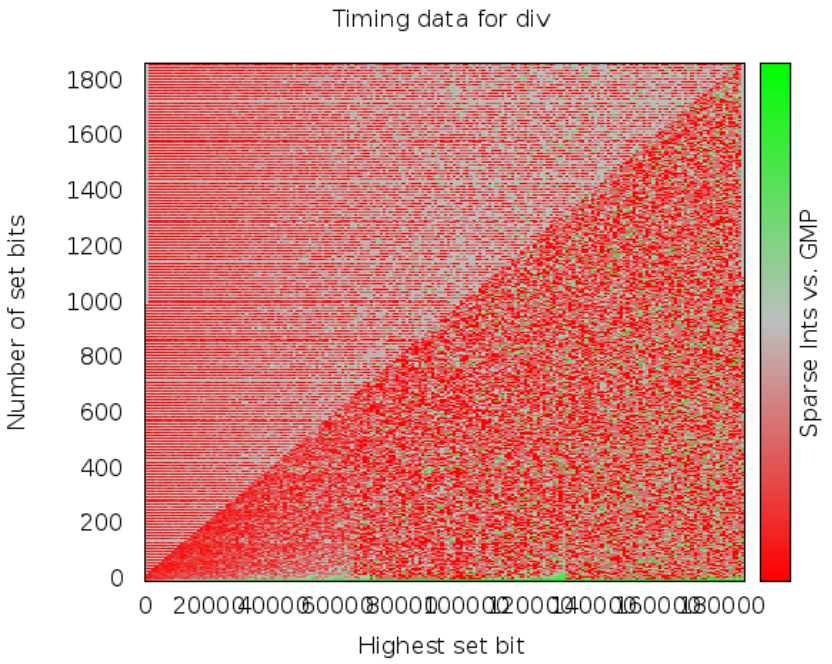

Figure 4: *Sparse Int division compared to GMP division*

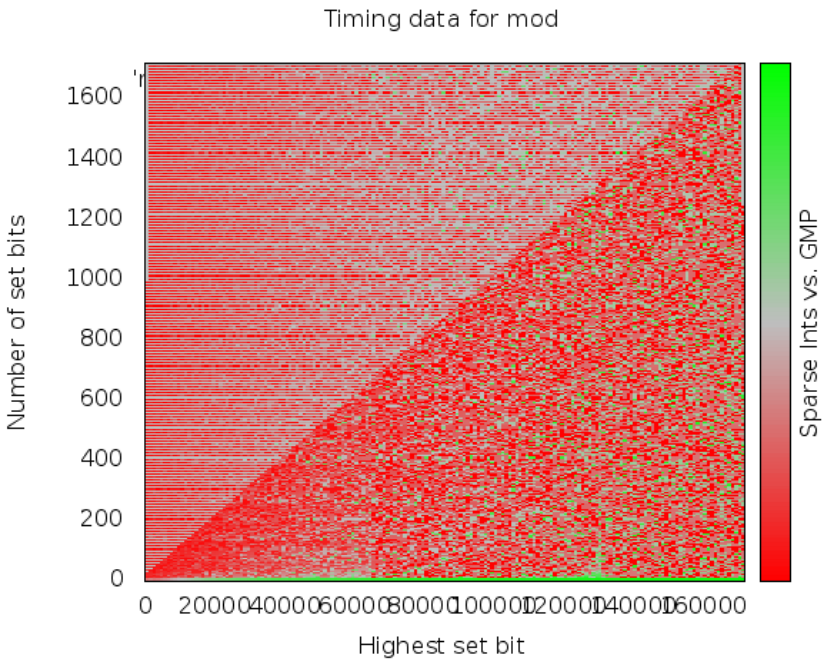

Figure 5: *Sparse Int modulus compared to GMP modulus*

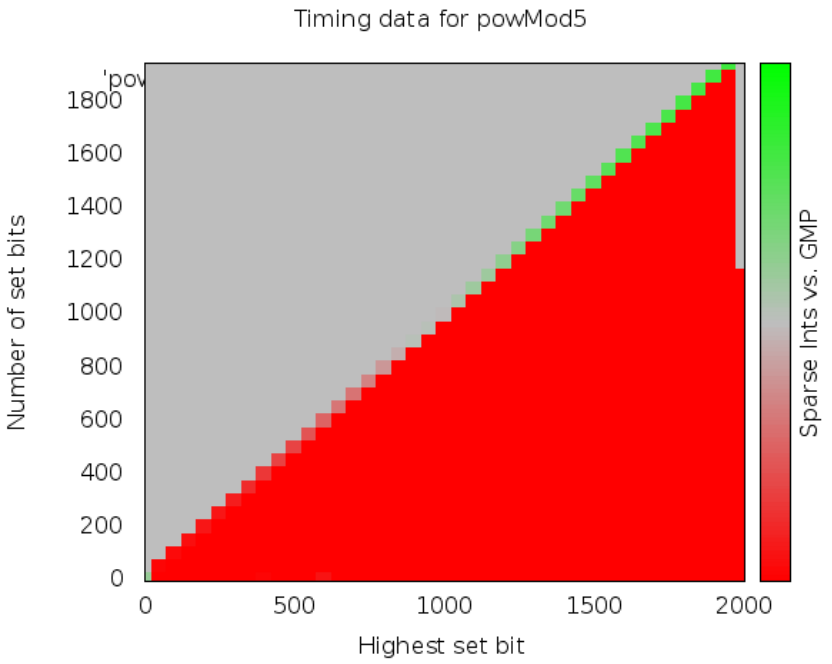

Figure 6: *Sparse Ints compared to GMP with regard to exponental moduls*

- Impliment an optimized squaring routine.
- Experiment with moving to a frequency domain as is used in polynomial multiplication.

References

William D. Banks and Igor E. Shparlinski. On the number of sparse RSA exponents. *Journal of Number Theory*, 95(2):340–350, 2002. doi: 10.1006/jnth.2001.2775.

Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Number 18 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge Univ. Press, November 2010.

National Institute of Standards and Technology NIST. FIPS 186-3, Digital Signature Standard (DSS). Technical report, Department of Commerce, June 2009. URL `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.

George W. Reitwiesner. Binary arithmetic. *Advances in Computers*, pages 231–308, 1960.

J.E. Vuillemin. Efficient data structure and algorithms for sparse integers, sets and predicates. In *Computer Arithmetic, ARITH 2009, 19th IEEE Symposium on*, pages 7–14, June 2009. doi: 10.1109/ARITH.2009.25.