A New Bignum Multiplication Algorithm

Michael Malenkov, Christopher J. Dutra, and Marco T. Morazán
Seton Hall University
Department of Mathematics and Computer Science
400 South Orange Avenue
South Orange, NJ 07079 USA
E-mail: {malenkmi,dutrachr,morazanm}@shu.edu

ABSTRACT

Many modern scientific computations require the manipulation of numbers that are too big to fit in a word of memory. These computations create the need to develop compound representations for integers, called bignums, and algorithms to support arbitrary precision arithmetic operations. One of the fundamental algorithms that must be implemented is bignum multiplication. The most popular and best known implementations of bignum multiplication use as their base case algorithm the classical algorithm described by Knuth. This algorithm is modeled on the multiplication algorithm taught in grade school and is optimized to eliminate redundant memory allocation. Under certain conditions, however, Knuth's classical algorithm suffers from limited degree of locality of reference. In this article, we outline a new base case bignum multiplication algorithm that carries out the computation with a higher degree of locality of reference when the size of the multiplier is less than two thirds the size of the multiplicand.

1. INTRODUCTION

Many modern scientific computations require the manipulation of numbers that are too big to fit in a word of memory. Among the applications that need such large numbers are safe encryption algorithms and the search for the largest known prime number[4]. To gauge the magnitude of the numbers we are referring to it suffices to examine the largest known Mersenne prime number which at this time is $2^{30402457} - 1$. This integer has $91,252,052$ digits and can not be stored in one word of memory using any modern computer architecture. In order to manipulate integers of this magnitude, *bignums* are used to represent integers as compound data structures. A bignum contains a series *bigits* which are digits in the base, $BASE$, used to represent integers. $BASE$ is always much larger than 10 which is the base human beings use on a daily basis. The value of $BASE$ is limited by the hardware limitations of modern computer systems. For efficiency reasons in many systems, $BASE$ does not exceed the largest integer representable using one word of memory.

A bignum is usually represented as a data structure that contains at least three pieces of information:

1. The address of an array of bigits.
2. The length of the array of bigits
3. The sign of the bignum

An array is usually used to store the bigits, instead of a list, in order to have efficient implementations of the basic arithmetic operations. For example, addition and substraction operations always start with the least significant bigits of the integers they receive as arguments while division always starts with the most significant bigits of the integers it receives as arguments. The use of lists, which are not random access, to represent bigums would add a lot of overhead to one or more of the basic arithmetic operations. The size of the maximum integer representable using an array of bigits, of course, is limited by the maximum size of the index into an array. Despite this limitation, however, the use of modern compound representations of integers continues to grow.

Any implementation of bignums must include an implementation for bignum multiplication. Bignum multiplication algorithms can be divided into two categories: base case multiplication algorithms and divide and conquer algorithms. The best known base case multiplication algorithm is the classical algorithm described by Knuth[3]. This algorithm is virtually implemented by all systems and libraries that support bignums (some examples are [1, 5]). The best known divide and conquer algorithms are the Karatsuba algorithm[2] and the Toom-Cook algorithm[1, 3]. The divide and conquer algorithms split very long bignums into smaller pieces until a threshold is reached where a the base case algorithm is applied.

It is important for the base case multiplication algorithm to be efficient in terms of memory space, time, and locality of reference, because it is ultimately always used during bignum multiplication. This article reviews Knuth's classical multiplication algorithm and argues that it does not always exhibit the highest degree of locality of reference. The article then presents a new base case multiplication algorithm inspired in polynomial multiplication that has a

Proceedings of MASPLAS'06
Mid-Atlantic Student Workshop on Programming Languages and Systems
Rutgers University, April 29, 2006

higher degree of locality of reference than Knuth's classical algorithm under conditions that are detectable at runtime. The article ends with some concluding remarks and directions of future work.

2. CLASSICAL MULTIPLICATION ALGORITHM

The classical bignum multiplication base case algorithm simulates the multiplication-by-hand algorithm taught in grade school. In the multiplication-by-hand algorithm, each bigit, b_i, of the multiplier is multiplied by each bigit of the multiplicand to produce a result that is multiplied by $BASE^i$. After all these intermediate results are known, they are added together to produce the desired product.

Knuth points out that directly implementing this algorithm on a computer is inefficient, because it causes a great deal of unnecessary intermediate memory allocation. Instead of allocating a bignum to store each product obtained from multiplying each bigit of the multiplier by the multiplicand, Knuth's classical algorithm adds each product as it is being generated to the result. This guarantees memory is only allocated is for the result of the multiplication.

Figure 1 displays pseudo-code for an implementation of Knuth's base case multiplication algorithm. The procedure *bignumMult* traverses the multiplier. For every bigit, b_i, of the multiplier that is not 0, b_i is multiplied by the multiplicand and added to the result in the procedure *MultAndAdd*. The procedure *MultAndAdd* traverses the multiplicand to multiply it by b_i. For each bigit, a_j, of the multiplicand, the product of a_j and b_i is computed. If this product is greater than the base of the bignum representation[1], a propagation takes place starting from the $i + j + 1^{th}$ bigit of the result, R, and the product is updated to only contain the non-propagated portion. The result of all propagations to R_{i+j} is then added to the non-propagated portion of the current product and a propagation takes place if necessary. After this step, all necessary propagations to all the bigits more significant than R_{i+j} has taken place and R_{i+j} is set to the non-propagated portion left which is the $(i + j)^{th}$ bigit of the result.

```
proc bignumMult(Multiplicand, Multiplier, Result)
    i = -1
    plierlen = MultiplierLen
    while (i < plierlen - 1)
        i = i + 1
        if (Multiplier_i != 0)
            MultAndAdd(i, Multiplicand, Multiplier, Result)

proc MultAndAdd(i, Multiplicand, Multiplier, Result)
    j = -1
    candlen = MultiplicandLen
    plierIdigit = Multiplier_i
    while (j < candlen)
        j = j + 1
        temp = (Multiplicand_j * plierIdigit)
        if (temp > BASE)
            propagate(temp DIV BASE, Result, i+j+1)
            temp = temp % BASE
        temp = temp + Result_{i+j}
        if (temp > BASE)
            propagate(temp DIV BASE, Result, i+j+1)
            temp = temp % BASE
        Result_{i+j} = temp MOD BASE

proc propagate(carryIn, Result, start)
    k = start
    carry = (carryIn + Result_k) DIV BASE
    Result_k = (carryIn + Result_k) MOD BASE
    while (carry > 0)
        k = k + 1
        temp = Result_k + carry
        Result_k = temp MOD BASE
        carry = temp DIV BASE
```

Figure 1: Implementation Pseudo-code for Knuth's Base Case Multiplication Algorithm.

3. CLASSICAL ALGORITHM AND LOCALITY OF REFERENCE

The degree of locality of reference of an algorithm can be measured by the size of the memory window that needs to be accessed at each step of the algorithm. Consider the multiplication of an n-bigit number, A, by an m-bigit number, B, where n $\geq$ m. Without loss of generality, we will let A be the multiplicand and B be the multiplier. That is, the multiplier is always the bigit with the smaller magnitude.

Knuth's classical algorithm requires that for each non-zero bigit of B, A be traversed. That is, the bignum with the largest magnitude is linearly traversed multiple times. It is inefficient to traverse the bignum with the largest magnitude multiple times. In the worst case, there are no 0 bigits in B and a memory window containing the n-bigits of A is accessed for each bigit of B. Furthermore, the algorithm also accesses an n-bigit memory window of the result. Thus, we have that a minimum of $2n + 1$ different bigits must be accessed at least once for every bigit of the multiplier.

For large enough values of n, any virtual memory system will be forced to perform page swapping between main memory and cache and between main memory and backing store. Although paging in a virtual memory system can not be entirely avoided, performance can be improved by making the size of the memory window accessed smaller. It is desirable to develop an algorithm that does not traverse the bignum with the largest magnitude multiple times.

4. POLYNOMIAL MULTIPLICATION

Knuth's classical multiplication algorithm focuses on explicitly multiplying each bigit of the multiplier by the multiplicand at each step in the algorithm. An alternate bignum multiplication algorithm is suggested by focusing, instead, on what is needed for each bigit of the result. To determine the dependencies of each bigit of the result, each bignum can be thought of as the representation of a polynomial. Each bigit of a bignum is the coefficient of a power of the base used in the representation. Thus, bignum multiplication can resemble polynomial multiplication.

[1]This implementation assumes that the product of two bigits fits in one memory location.

An n-degree polynomial, $A(x)$, and an m-degree polynomial, $B(x)$, can be characterized as follows:

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-2}x^{n-2} + a_nx^{n-1}$$
$$B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-2}x^{m-2} + b_nx^{m-1}$$

The product of $A(x) * B(x)$ can be characterized as follows:

$$\begin{aligned} A(x)B(x) = {} & a_0b_0 \\ & + (a_0b_1 + a_1b_0)x \\ & + (a_0b_2 + a_1b_1 + a_2b_0)x^2 \\ & \cdots \\ & + (a_0b_{n+m} + a_1b_{n+m-1} + \cdots + a_{n+m}b_0)x^{n+m}. \end{aligned}$$

Therefore, we have that each coefficient, r_i, of $A(x) * B(x)$ can be characterized as follows:

$$r_i = \textstyle\sum_{j=0}^{i} a_j b_{i-j},$$

where a_j is 0 if $j > n$ and, similarly, b_{i-j} is 0 if $i - j < 0$ or $i - j > m$.

5. NEW MULTIPLICATION ALGORITHM

5.1 The Algorithm

The mathematical formulation of the product of two polynomials immediately suggests an algorithm for bignum multiplication. At each step during the algorithm one of the bigits of the result is computed. Unlike polynomial multiplication where there is an unknown value represented by a variable, in bignum multiplication the are no unknown values. This means that i^{th} bigit of the result depends on r_i and on the carry generated from the computations to obtain the values of $r_0 \ldots r_{i-1}$. This carry must be added to r_i and a carry must be generated for the computation of r_{i+1}.

Pseudo-code for this algorithm is displayed in Figure 2. Each bigit of the result is computed starting from the least significant bigit. For each bigit i of the result, R_i, that is computed, a carry is propagated to the bigits $R_{i+1} \ldots R_{len-1}$. This work is done by the procedure *bignumMult*.

The computation of the i^{th} bigit of the result and the propagation of any carry is done the the procedure *compCoeffAndPropagate*. This procedure traverses through the values in $[0 \ldots i]$ adding a bigit product term corresponding to $A_i * B_{i-j}$ to R_i and propagating a carry if necessary. A product term is 0 if either of the subscripts to the multiplier or the multiplicand are to *imaginary* bigits (i.e. $i \notin [0..Alen-1]$ and $i - j \notin [0..Blen - 1]$) and is $A_i * B_{i-j}$ otherwise.

5.2 Properties

The first observation that can be made about algorithm I is that at each step at most $Blen$ bigits of the multiplicand, $Blen$ bigits of the multiplier, and $Blen-1$ bigits of the result are accessed. This algorithm requires a memory window of $3Blen - 1$ bigits at each step of the computation.

It is necessary to determine when the size of the memory window needed by each step of this algorithm is less than the size of the memory window needed by each step of Knuth's classical algorithm. Solving the following inequality for $Blen$ yields the answer:

$$\begin{aligned} 3Blen - 1 &< 2Alen + 1 \\ Blen &< \tfrac{1}{3}(2Alen + 2) \end{aligned}$$

This result states that each step of this new multiplication algorithm requires a smaller memory window than each step of Knuth's classical algorithm when the length of the multiplier is, roughly, less than $\frac{2}{3}$ the length of the multiplicand. When this condition does not hold, the size of the memory window required by each step of Knuth's classical algorithm is the same or smaller. This suggests that by comparing the lengths of the multiplicand the multiplier at runtime a bignum multiplication implementation can determine which algorithm ought to be used.

This first algorithm, although provably correct and more *local* than Knuth's classical algorithm under the condition stated above, is rather unsatisfying because a lot of work is done to add product terms that are 0. These 0 terms correspond to the products of bigits that do not exist in the bignums being multiplied and will occur in every computation of R_i for $i > Blen$.

```
proc bignumMult(Multiplicand, Multiplier, Result)
   i = -1
   while (i < ResultLen - 2)
      i = i + 1
      compCoeffAndPropagate(i,Multiplicand, Multiplier,Result)
proc compCoeffAndPropagate(i,A,B, Res)
   j = -1
   while (j < i)
      j = j + 1
      Res_i = Res_i + bigitProductTerm(i,i-j,A,B)
      if (Res_i > BASE)
         propagate(Res_i/BASE,Res,i+1)
         Res_i = Res_i MOD BASE
proc bigitProductTerm(asub, bsub, A, B)
   term = 0
   if ((asub < Alen) ∧ (bsub > -1) ∧ (bsub < Blen))
      term = A_asub * B_bsub
   return(term)
proc propagate(carryIn, Result, start)
    k = start
    carry = (carryIn + Result_k) DIV BASE
    Result_k = (carryIn + Result_k) MOD BASE
    while (carry > 0)
       k = k + 1
       temp = Result_k + carry
       Result_k = temp MOD BASE
       carry = temp DIV BASE
```

Figure 2: Pseudo-Code for Multiplication Algorithm I.

6. ALGORITHM REFINEMENT

6.1 Location of 0 Terms

To eliminate the work done for 0 terms, the computation required for each r_i can be examined to determine where

the 0 terms occur. Recall that r_i is given by:

$$r_i = a_0b_i + a_1b_{i-1} + a_2b_{i-2} + \ldots + a_{i-2}b_2 + a_{i-1}b_1 + a_ib_0.$$

Notice that for all product terms needed for r_i the sum of the subscript of a and the subscript of b is always equal to i. This invariant property can be exploited to determine which product terms are 0 terms and which are not 0 terms. If $i < Blen$, then there are no 0 terms in the computation of r_i. This follows from observing that the subscript of any a is always less than or equal to $Alen - 1$ and observing that the subscript of any b is never negative and is never greater than $Blen - 1$. In other words, the range of subscripts for both a and b are in $[0..Blen-1]$ which means that all a_ib_{i-j} are the product of two existing bigits. Thus, we have that:

$$i < m \Rightarrow r_i = \sum_{j=0}^{i} a_jb_{i-j}$$

For the computation of each r_i when $i > Alen - 1$, the most significant bigit of A, A_{Alen-1}, and the most significant bigit of B, B_{Blen-1}, are needed. Thus, the maximum valid subscript for a in the computation of r_i is $Alen-1$. Note that the values of a_j for $j \in [Alen \ldots i]$ are 0 terms and there is no need to add these terms to r_i. To find the minimum valid subscript for a for the computation of r_i, set the subscript of b to $Blen - 1$ and solve the following equation for j:

$$\begin{aligned} i - j &= Blen - 1 \\ -j &= -i + Blen - 1 \\ j &= i - Blen + 1 \end{aligned}$$

Thus, we have that:

$$i > Alen - 1 \Rightarrow r_i = \sum_{j=i-Blen+1}^{Alen-1} a_jb_{i-j}$$

For the computation of each r_i when $m \leq i \leq Alen-1$, there are $Blen$ non-zero product terms. This means that each r_i depends on each bigit of B. Thus, $i - j \in [0 \ldots B_{len} - 1]$. To find the smallest valid subscript for a solve the following equation as before:

$$\begin{aligned} i - j &= Blen - 1 \\ -j &= -i + Blen - 1 \\ j &= i - Blen + 1 \end{aligned}$$

The largest valid subscript for a is given by:

$$\begin{aligned} i - j &= 0 \\ -j &= -i \\ j &= i \end{aligned}$$

Thus, we have that:

$$Blen \leq i \leq Alen - 1 \Rightarrow r_i = \sum_{j=i-m+1}^{i} a_jb_{i-j}$$

```
proc bignumMult(cand, plier, Result)
   i = -1
   while (i < ResultLen - 2)
      i = i + 1
      sumAndPropValidBigitProducts(i, cand, plier, Result)
sumAndPropValidBigitProducts(i, A, B, Res)
   if (i < Blen)
      sumAndProp(0, i, i, A, B, Res)
   else if (i > Alen - 1)
         sumAndProp(i-Blen+1, Alen-1, i, A, B, Res)
     else
         sumAndProp(i-Blen+1, i, i, A, B, Res)
sumAndProp(start, finish, i, A, B, Res)
   j = start - 1
   coeff = Res_i
   while (j < finish)
      j = j + 1
      coeff = coeff + bigitproductTerm(j,i-j, A, B)
      if (coeff ≥ BASE)
         propagate(coeff/BASE,Res,i)
         coeff = coeff MOD BASE
   Res_i = coeff
bigitproductTerm(asub, bsub, A, B)
   return(A_asub * B_bsub)
proc propagate(carryIn, Result, start)
   k = start
   carry = (carryIn + Result_k) DIV BASE
   Result_k = (carryIn + Result_k) MOD BASE
   while (carry > 0)
      k = k + 1
      temp = Result_k + carry
      Result_k = temp MOD BASE
      carry = temp DIV BASE
```

Figure 3: Pseudo-code for Multiplication Algorithm II.

6.2 Algorithm

The above reformulation of the equation for r_i suggests an improved bignum multiplication algorithm that does not perform any work to determine if a term is 0 during the computation of r_i. The pseudo-code for this refinement if displayed in Figure 3. The procedure *bignumMult* traverses the bignum allocated for the result to compute each r_i and generate the carry for r_{i+1}. No computation takes place for the most significant bigit of the result, r_{Rlen-1}, since it only depends on the carry generated during the computation of r_{Rlen-2}.

The procedure *sumAndPropValidBigitProducts* computes each r_i by identifying in which range of interest (i < Blen, i > Alen - 1, or Blen ≤ i ≤ Alen - 1) i lies in and calling *sumAndProp* with the indexes of lowest and highest valid bigits of A. This action guarantees that during the computation of each r_i there is no work done for 0 terms.

The procedure *sumAndProp* takes as part of its input the indexes into A where the computation of r_i should start and finish. It sets the variable $coeff$, which represents the value

of r_i computed so far, to Res_i which stores the carry it receives from r_{i-1}. It then loops through indexes, j, of A from $start$ to $finish$ adding product terms to $coeff$ and propagating a carry if necessary. Each of these product terms is known to only index valid bigits of A and B and, therefore, the procedure *bigitproductTerm* always returns $a_i * b_{j-1}$.

Eliminating the addition of 0 terms during the computation of r_i does not change the number of different bigits accessed. Therefore, we have that the size of the memory window required for this refined algorithm is the same as the algorithm in Figure 2.

7. COST OF REDUCING THE MEMORY WINDOW SIZE

The reduction of the memory window size when $Blen < \frac{1}{3}(2Alen + 2)$ comes at a runtime cost when the multiplier contains many 0 bigits. Knuth's classical algorithm performs no bigit by bigit multiplications when a bigit of the multiplier is 0. There is no need to, because the product of the multiplicand and 0 is always 0 and adds no value to the result. The new algorithm described in this article performs all bigit by bigit multiplications for a multiplier bigit that is 0. Thus, the number of memory accesses and bigit by bigit multiplications performed is larger for the new algorithm in the presence of bigits that are 0 in the multiplier.

The number of extra bigit by bigit multiplications performed by the our algorithm is given by:

$$ExtraMults = Number\ of\ multiplier\ 0\text{-}bigits * Alen$$

When there are no bigits that are 0 in the multiplier, Knuth's classical algorithm and the new algorithm described in this article perform the same number of bigit by bigit multiplications. If the number of bigits that are 0 in the multiplier is small, then the impact on performance of the extra bigit by bigit multiplications done by the new algorithm is likely to be negligible. On the other hand, if the multiplier is expected to have many 0 bigits then the effort to reduce the memory window size may be counter productive. In the worst case, all the bigits of the multiplier but 1 may be 0 and the new algorithm performs $(Blen-1)*Alen$ more bigit by bigit multiplications. For small values of $Alen$, however, the impact on performance is likely to be negligible.

The likelihood of having small values of $Alen$ is high, because base case multiplication algorithms like those described in this article are not directly used in practice to multiply bignums that are very long. For very long bignums, divide and conquer algorithms such as Karatsuba multiplication are employed. These algorithms divide long bignums repeatedly into smaller bignums until the lengths of the pieces are small enough to apply a base case multiplication algorithm. Therefore, we conjecture that the impact on performance of the extra multiplications performed by the new algorithm described in this article is negligible.

8. CONCLUDING REMARKS

This article describes a new base case multiplication algorithm for integers that are bignums. When compared to Knuth's classical bignum multiplication algorithm, the new algorithm reduces the size of the memory window required at each step of the computation if the length of the multiplier is less than $\frac{2}{3}$ the length of the multiplicand. Knuth's classical multiplication algorithm revolves around multiplying each bigit of the multiplier by the multiplicand. Our new algorithm, in contrast, revolves around computing each bigit of the product. By focusing of computing each bigit of the product instead of multiplying each bigit of the multiplier by the multiplicand, the new algorithm avoids having to repeatedly traverse the multiplicand which is always the bignum with the largest magnitude. Thus, yielding a bignum multiplication algorithm that does not perform any unnecessary allocations as Knuth's classical algorithm and that makes the process of multiplication more local.

The new algorithm, however, can perform more memory accesses and more bigit-by-bigit multiplications than Knuth's classical algorithm when the multiplier contains bigits that are 0. The classical algorithm reduces the amount of memory accesses and of multiplications by performing no work when it processes a multiplier bigit that is 0. The new algorithm presented in this article can avoid performing actual multiplications by 0, but can not avoid the extra memory accesses. This is due to not individually processing each bigit of the multiplier and, instead, individually processing each bigit of the result. Our future work includes refining the new multiplication algorithm to avoid unnecessary memory accesses and multiplications when a bigit is 0. Our focus, however, is not on bigits of the multiplier. Instead, our refinement focuses on what to do when bigits of the multiplicand are 0. Given that the multiplicand has a larger magnitude than the multiplier, it is likely to contain more zeroes than the multiplier. Reducing memory accesses and bigit-by-bigit multiplications for bigits of the multiplicand that are zero is likely to yield an algorithm that performs less work than Knuth's classical bignum multiplication algorithm.

Our future work also includes performing benchmark experiments to gather empirical measurements to quantify the impact on performance of the final version of the new algorithm. In addition, our future work also includes studying how the new algorithm can be efficiently used to implement Karatsuba multiplication and other divide and conquer multiplication algorithms.

9. ACKNOWLEDGEMENTS

The authors would like to thank the Department of Mathematics and Computer Science of Seton Hall University for their support. We are also very grateful to Dr. Barbara Ryder and the MASPLAS'06 Organizing Committee for their efforts to provide a forum for the presentation of our work.

10. REFERENCES

[1] Torbjörn Granlund. GNU MP: The GNU Multiple Precisision Arithmetic Library. http://swox.com/gmp/, September 2004.

[2] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Phys. Dokl.*, 7:595–596, 1963.

[3] Donald E. Knuth. *Seminumerical Algorithms*. Addison-Wesley, 1981.

[4] Mersenne.org. 43rd Known Mersenne Prime Found!! http://mersenne.org/prime.htm, December 2005.

[5] The Larceny Project. Larceny User's Manual. http://www.ccs.neu.edu/home/will/Larceny/manual/, 2006.