MIT/LCS/TM-45

FAST ON-LINE INTEGER MULTIPLICATION

Michael J. Fischer
Larry J. Stockmeyer

MAY 1974

This blank page was inserted to preserve pagination.

MAC TECHNICAL MEMORANDUM 45

FAST ON-LINE INTEGER MULTIPLICATION

Michael J. Fischer

Larry J. Stockmeyer

May 1974

This research was supported in part by the National Science Foundation under research grant GJ-34671, and in part by the Advanced Research Projects Agency, Department of Defensewhich was monitored by ONR Contract No. N00014-70-A-0362-0003 and N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

This empty page was substituted for a blank page in the original document.

TM-45

FAST ON-LINE INTEGER MULTIPLICATION[†]

Michael J. Fischer and Larry J. Stockmeyer

Massachusetts Institute of Technology
Cambridge, Massachusetts

ABSTRACT

A Turing machine multiplies binary integers *on-line* if it receives its inputs low-order digits first and produces the jth digit of the product before reading in the $(j+1)$st digits of the two inputs. We present a general method for converting any off-line multiplication algorithm which forms the product of two n-digit binary numbers in time $F(n)$ into an on-line method which uses time only $O(F(n) \log n)$, assuming that F is monotone and satisfies $n \leq F(n) \leq F(2n)/2 \leq kF(n)$ for some constant k. Applying this technique to the fast multiplication algorithm of Schönhage and Strassen gives an upper bound of $O(n (\log n)^2 \operatorname{loglog} n)$ for on-line multiplication of integers. A refinement of the technique yields an optimal method for on-line multiplication by certain sparse integers. Other applications are to the on-line computation of products of polynomials, recognition of palindromes, and multiplication by a constant.

[†]This research was supported in part by the National Science Foundatioon under research grant GJ-34671 to M.I.T. Project MAC, and in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research contract number N00014-70-A-0362-0003 to the M.I.T. Artificial Intelligence Laboratory, and contract number N00014-70-A-0362-0006 to M.I.T. Project MAC. The final preparation of the manuscript was supported by the University of Toronto.

1. Introduction

The problem of finding the product of two integers expressed, say, in binary notation is basic to the study of computation, yet the number of computational steps required is still not well understood. The classical algorithm taught in school uses a number of steps that grows as n^2 to multiply n-digit numbers. More sophisticated algorithms have considerably reduced this rate of growth; the best algorithm known, due to Schönhage and Strassen [8], requires only $O(n \log n \text{ loglog } n)$ computational steps. This bound can be achieved even on a multitape Turing machine, the model we investigate in this paper. However, no interesting lower bounds are known. It is not even known if the number of steps must grow faster than linearly in the length of the input.

The on-line restriction constrains the manner in which a computation is carried out. Informally, a function is computed *on-line* if the input symbols are read sequentially, the output is produced sequentially, and the machine produces the j^{th} output symbol before reading the $j+1^{\text{st}}$ input symbol(s). This is in contrast to *off-line* algorithms in which the machine has access to all input symbols before any output need be produced. The on-line model is a natural paradigm for interactive computing and process control where the future inputs depend in an unpredictable way on the current outputs. Our interest in the on-line restriction stems partially from such practical considerations and partially from the mathematical tractability of on-line computation which enables some nontrivial lower bounds to be obtained.

The particular on-line problem of integer multiplication is motivated by a deep result due to Cook and Aanderaa [1] and strengthened by Paterson, M. Fischer and Meyer [7] which says that for any of a broad class of machine models, a machine $\mathcal{M}$ to perform on-line integer multiplication of n-digit numbers requires more than $cn \log n/(\text{loglog } n)$ steps for all sufficiently large n, where $c > 0$ is a constant depending only on $\mathcal{M}$. In the case of a one-dimensional multitape Turing machine, the bound becomes $cn \log n$. Unfortunately, the methods in [1] and [7] do not apply if the computation is done off-line.

It is not immediately obvious that on-line multiplication is even possible, for the usual multiplication algorithms do not obey the on-line restriction. The reader can easily convince himself that on-line multiplication can in fact be done, but the best on-line algorithm previously known requires time $O(n^2)$, leaving a considerable gap between the upper and lower bounds. The results given here significantly reduce this gap.

Our main result is given as an off-line to on-line conversion method. That is, given a Turing machine *OFF* which computes integer multiplication off-line, we describe a Turing machine *ONLINE* which computes integer multiplication on-line using *OFF* as a subroutine. We will henceforth assume that all Turing machines mentioned have one-dimensional tapes.

Theorem 1. Let *OFF* be a multitape Turing machine which performs off-line integer multiplication. Let $F(n)$ bound the time required by *OFF* to multiply n-digit integers. Assume F is monotone and $n \leq F(n) \leq F(2n)/2 \leq kF(n)$ for some constant $k \geq 1$. Then there is a multitape Turing machine *ONLINE* which performs integer multiplication, obeys the on-line restriction, and produces the n^{th} output digit in $O(F(n) \log n)$ computational steps.

We remark that the constant coefficient implicit in the "O"-notation can be made arbitrarily small using linear speedup for Turing machines [4] with a consequent increase in the sizes of the tape alphabets and finite control.

The following corollary is immediate using the method of Schönhage and Strassen referred to earlier.

Corollary 2. There is a multitape Turing machine which performs on-line integer multiplication and produces the n^{th} output digit in $O(n\ (\log n)^2\ \text{loglog}\ n)$ steps.

There are two senses in which Theorem 1 can be said to be optimal. First of all, the $cn \log n$ lower bound of [7] for on-line multiplication by multitape Turing machines holds even if the machine is augmented with a special multiply instruction which performs off-line integer multiplication in real time. A real time computation produces one output digit at each step and hence multiplies n-digit integers within time $2n$. The proof of Theorem 1 extends to these machines, resulting in a time $O(n \log n)$ method. Thus, Theorem 1 is the best possible result for a general off-line to on-line multiplication conversion method on these augmented Turing machines and strongly suggests the same for ordinary multitape Turing machines as well.

Another indication of the optimality of our basic conversion method is shown in Section 3 where we obtain a time $O(n \log n)$ on-line algorithm for multiplying an arbitrary integer x by a particular number K_n, where n is the length of x. Time $cn \log n$ is required for this problem as well [7].

The definition of on-line computation is given in [1] and [5], and we repeat it here, defining also the concept of half-line computation which simplifies the description of the conversion method. Informally, a machine

computes a two argument function half-line if one entire argument is available off-line and the other is read subject to the on-line restriction.

Definition. Let $\mathcal{M}$ be a machine which computes a function f on sequences, where f: $\Sigma^* \times \Sigma^* \to \Delta^*$, Σ and Δ are sets. $\mathcal{M}$ is said to compute f *on-line* if for all input sequences $a = a_0a_1\ldots a_n$, $b = b_0b_1\ldots b_n$ $(a_i, b_j \in \Sigma)$ and corresponding outputs $f(a,b) = c_0c_1\ldots c_n$ $(c_j \in \Delta)$, $\mathcal{M}$ produces c_k before reading either of a_{k+1} or b_{k+1}, $0 \le k \le n-1$. We assume here also that the inputs are read in sequence, so for all k, a_k (b_k) is read before a_{k+1} (b_{k+1}).

$\mathcal{M}$ computes f *half-line* (with respect to the first argument) if $\mathcal{M}$ produces c_k before reading a_{k+1}, $0 \le k \le n-1$. a will be referred to as the *on-line argument* and b as the *off-line argument* of the half-line product.

2. The Off-Line to On-Line Conversion

2.1. Informal Description.

We first give a general description of an on-line multiplication algorithm *ONLINE*, independent of the particular machine model on which it is to be programmed, and omitting many of the bookkeeping and data management details. Section 2.2 gives the construction in detail.

The definition of *ONLINE* is in terms of two auxiliary procedures *ON* and *HALF*, each of which uses a given off-line multiplication procedure *OFF*.

ON(n) assumes that the first $n/2$ digits of the product of two n-digit integer inputs have already been computed by prior calls to *ON*. It produces the next $n/2$ digits on-line as the corresponding inputs are being read. It

then computes and stores the n high-order digits of the product in preparation for a subsequent call to *ON*($2n$).

HALF(n) forms the product of two n-digit integer inputs, producing the n low-order digits of the product half-line as the on-line argument is being read. It then computes and stores the additional n high-order digits.

Let a, b be two n-digit binary* integers, where n is a power of 2, and write $a = a_1 \cdot 2^{n/2} + a_0$, $b = b_1 \cdot 2^{n/2} + b_0$ where a_1, a_0, b_1, b_0 are $n/2$-digit. Suppose $ab = c$ and $c = c_2 \cdot 2^n + c_1 \cdot 2^{n/2} + c_0$, where c_0, c_1 are $n/2$-digit and c_2 is n-digit.

1. c_0 is the $n/2$ low-order digits of $a_0 \cdot b_0$. Let d_1 denote the $n/2$ high-order digits of this product.

2. c_1 is the $n/2$ low-order digits of $d_1 + a_0 \cdot b_1 + a_1 \cdot b_0$. Let d_2 denote the (roughly) $n/2$ high-order digits of this expression.

3. $c_2 = d_2 + a_1 \cdot b_1$.

These definitions are illustrated in Figure 1.

All that need be done is to perform the four $n/2$-digit multiplications $a_i \cdot b_j$ and the indicated additions in such a way as not to violate the on-line or half-line restrictions. Both procedures are of the same general form, differing only in input-output arrangements.

To *ON*(n):

1. Compute $d_1 + a_0 \cdot b_1 + a_1 \cdot b_0$ as a_1 and b_1 are being read by running two *HALF*($n/2$) procedures in parallel, performing additions as the output digits are produced. a_1, b_1 are the on-line arguments and a_0, b_0 the off-line arguments of these two half-line products.

2. After a_1 and b_1 have been read, compute $d_2 + a_1 \cdot b_1$ by one use of *OFF* and additions, and store the result.

---

*We assume base 2 numbers for convenience. Everything generalizes trivially to an arbitrary base.

Suppose now that a and b are to be multiplied half-line. Let a be the on-line and b the off-line argument. Assume the length of b is $\leq n$.

To *HALF*(n):

If $n = 1$,

1. Read a and print the digit ab.

else

2. Compute $a_0 \cdot b_0$ half-line by *HALF*($n/2$) as a_0 is being read.
3. After a_0 has been read, compute $a_0 \cdot b_1$ off-line by using *OFF*, and store these digits.
4. Compute $a_1 \cdot b_0$ half-line by HALF($n/2$) as the digits of a_1 are read, forming the digits of $d_1 + a_0 \cdot b_1 + a_1 \cdot b_0$ as the outputs are produced.
5. Compute $d_2 + a_1 \cdot b_1$ off-line by using *OFF* and additions.

The top-level procedure *ONLINE* multiplies two numbers on-line without needing to know their lengths in advance. It operates by calling *ON*(n) with n equal to successive powers of 2.

To *ONLINE*:

1. Read the first digit from each input and compute the first product digit.
2. $n \leftarrow 2$.
3. *ON*(n).
4. $n \leftarrow 2n$.
5. Go to 3.

Let $F(n)$ denote the number of steps required by *OFF* to multiply n-digit numbers, and assume F satisfies the conditions of Theorem 1. Let $N(n)$ and $H(n)$ denote the number of steps required by *ON*(n) and *HALF*(n) respectively, for n a power of 2. Then the following relations hold:

$$N(n) \leq 2H(n/2) + F(n/2) + c_1 n; \tag{1}$$

$$H(n) \leq \begin{cases} c_2 n \text{ if } n = 1; \\ 2H(n/2) + 2F(n/2) + c_2 n \text{ if } n \geq 2. \end{cases} \tag{2}$$

The terms $c_1 n$, $c_2 n$, where c_1 and c_2 are constants, bound the times required for additions and overhead. In the next section we suggest a Turing machine implementation in which these overheads are indeed bounded by $O(n)$.

The relation (2) solves as:

$$H(n) \leq \sum_{i=1}^{\log n} 2^i \cdot F(n/2^i) + c_2 n(1 + \log n) = O(F(n) \log n) \tag{3}$$

since $n \leq F(n) \leq F(2n)/2$ for all n a power of 2. (We assume the summation is zero when the upper limit $\log n = 0$.) From (1) and (3), we immediately get $N(n) = O(F(n) \log n)$.

Let $T(n)$ be the time when the n^{th} output digit is produced by *ONLINE* for arbitrary n, and let $r = \lceil \log n \rceil$. Then

$$T(n) \leq \sum_{i=1}^{r} N(2^i) + c_3 n \leq c_3' r F(2^r) + c_3 n = O(F(n) \log n)$$

for constants c_3 and c_3' by the assumptions on F.

2.2 *Detailed Description.*

The descriptions of the procedures *ONLINE*, *ON*, *HALF* and *OFF* in Section 2.1 leave somewhat vague both the order in which the computations are to be performed and the handling of the arguments to the functions. In this section, we define the routines more precisely to make clear that they are correct and that the on-line restriction is not violated. The implementation of these procedures on a multitape Turing machine to achieve the time bounds of Section 2.1 is straightforward.

The main procedure, *ONLINE*, takes two on-line arguments A and B and produces the product AB on-line. The inputs and outputs may be thought of as *streams*, that is, they are read and written sequentially, low-order digits first. READ(X) reads and returns the next digit of the on-line argument $X \in \{A, B\}$; PRINT(s) causes s to be produced as the next digit of output, $s \in \{0, 1\}$.

The other procedures take three kinds of arguments. An *on-line argument* just names an input stream. An *off-line argument* is a binary string. A *procedural argument* is another procedure which may be passed to a subprocedure or executed directly by the called procedure. In the descriptions that follow, A and B denote on-line arguments, P denotes a procedural argument, and lower case letters are used for off-line arguments and variables.

We first present the procedures and then prove them correct. The assertions of Lemma 3 describe exactly what *ON* and *HALF* are supposed to do, and Lemma 4 describes the effect of *ONLINE*.

OFF(x, y) is the assumed off-line multiplication procedure with running time $F(n)$ of Theorem 1. *SPLIT*(x, n) is a function which returns a pair of natural numbers (x_1, x_0) such that $x_1 \cdot 2^n + x_0 = x$ and $x_0 < 2^n$. *CAT*(x_1, x_0, n) is the inverse of *SPLIT* and returns $x_1 \cdot 2^n + x_0$. (When x_0 is a length n number, this just performs concatenation of binary strings.) Since our numbers are represented in binary notation, it should be clear that *SPLIT* and *CAT* both run in time proportional to the length of x.

To $ON(n, d, a_0, b_0, A, B)$:

1. $c \leftarrow 0$;
2. Compute in parallel, starting with (i) and switching alternately between (i) and (ii) every time a READ instruction is about to be executed:

   (i) $(p, a_1) \leftarrow HALF(n/2, d, b_0, A, (\lambda x \,.\, t \leftarrow x))$;

   (ii) $(q, b_1) \leftarrow HALF(n/2, 0, a_0, B, (\lambda x \,.\, (c, s) \leftarrow SPLIT(c + x + t, 1);$ PRINT$(s)))$;
3. $d_2 \leftarrow c + p + q$;
4. $c_2 \leftarrow d_2 + OFF(a_1, b_1)$;
5. Return (c_2, a_1, b_1).

Step 2 of this algorithm requires a little explanation. Statements 2(i) and 2(ii) each have side-effects: 2(i) affects the value of the variable t, and 2(ii) affects the value of the variable c and causes printing. Also, both statements cause input to be read, 2(i) from stream A and 2(ii) from stream B. To insure that the on-line restriction is not violated and that the correct results are produced (since 2(ii) depends on 2(i)), the statements must be executed in parallel, subject to the synchronization constraint that the j^{th} execution of the procedural argument of 2(i) precedes the j^{th} execution of the procedural argument of 2(ii) which in turn precedes the $(j+1)^{st}$ READ from either input stream. The method of execution proposed in Step 2 guarantees this sequencing, for each of 2(i) and 2(ii) causes precisely one evaluation of its procedural argument between successive calls on READ. (Cf. Lemma 3, part B.4.)

To *HALF*(n, d, b, A, P):

If $n = 1$, then

1. $a \leftarrow$ READ(A);
2. $(c, s) \leftarrow SPLIT(d + ab, 1)$;
3. Call $P(s)$;
4. Return (c, a);

else

5. $(d_1, d_0) \leftarrow SPLIT(d, n/2)$;
6. $(b_1, b_0) \leftarrow SPLIT(b, n/2)$;
7. $(p, a_0) \leftarrow HALF(n/2, d_0, b_0, A, P)$;
8. $(d_2', d_1') \leftarrow SPLIT(p + d_1 + OFF(a_0, b_1), n/2)$;
9. $(q, a_1) \leftarrow HALF(n/2, d_1', b_0, A, P)$;
10. $c_2 \leftarrow q + d_2' + OFF(a_1, b_1)$;
11. $a \leftarrow CAT(a_1, a_0, n/2)$;
12. Return (c_2, a).

Lemma 3. Let n be a power of 2.

A. Assume $n \geq 2$ and let d, a_0 and b_0 be numbers of length $\leq n/2$. Suppose $(c_2, a_1, b_1) = ON(n, d, a_0, b_0, A, B)$, and *ON* produces the m-digit number c_1 as output. Then:

A.1. *ON* reads $n/2$ digits from A; these digits are a_1.

A.2. *ON* reads $n/2$ digits from B; these digits are b_1.

A.3. $m = n/2$.

A.4. *ON* obeys the on-line restriction.

A.5. $c_2 \cdot 2^{n/2} + c_1 = d + a_0 \cdot b_1 + a_1 \cdot b_0 + a_1 \cdot b_1 \cdot 2^{n/2}$.

B. Let d and b be numbers of length $\leq n$. Suppose (z_1, a) = *HALF*(n, d, b, A, P), and *HALF* calls P r times with the successive digits of z_0 as arguments. Then:

B.1. *HALF* reads n digits from A; these digits are a.

B.2. $r = n$.

B.3. *HALF* produces no output (except as a possible result of calls on P).

B.4. For every j, $1 \leq j \leq n$, *HALF* calls P for the j^{th} time before reading the $j{+}1^{\text{st}}$ digit of A and after reading the j^{th}.

B.5. $z_1 \cdot 2^n + z_0 = d + ab$.

Proof. The proof is by induction on n.

For $n = 1$, part A is vacuously true. Part B follows by inspection of the case $n = 1$ in the definition of *HALF*.

Now, let n be a power of 2, and suppose the statement of Lemma 3 is true for every $n' < n$ which is also a power of 2. Statements A.1 - A.3 follow from the definition of *ON* and from B.1 - B.3 for $n' = n/2$. A.4 follows from B.3 and B.4 for $n' = n/2$ and from the remarks following the definition of *ON* about the sequencing in Step 2. A.5 follows by direct calculation from B.5 for $n' = n/2$ and the following:

Claim: Let p_0 be the sequence of $n/2$ digits with which the procedural argument of Step 2(i) of *ON* is successively called, regarded as a binary integer, low-order digit first. Let q_0 be the corresponding sequence for 2(ii). Let c_1 be the sequence of digits printed by Step 2 of *ON*, and let c be the value of the variable by the same name at the completion of Step 2. Then $c \cdot 2^{n/2} + c_1 = p_0 + q_0$. We leave the proof of this claim to the reader.

B.1 and B.2 follow easily by induction, and B.3 follows by inspection, since *HALF* contains no PRINT statements. B.4 just says READs alternate with calls on P. This is again obvious since the only READ occurs in Step 1 of *HALF* and the only call on P is in Step 3. Finally, B.5 follows by induction using B.5 for $n' = n/2$ and direct calculation. □

We now define the main on-line multiplication procedure, *ONLINE*.

To *ONLINE*(A, B):

1. $a_0 \leftarrow$ READ(A); $b_0 \leftarrow$ READ(B); PRINT($a_0 \cdot b_0$);
2. $n \leftarrow 2$; $d \leftarrow 0$;
3. $(c_2, a_1, b_1) \leftarrow ON(n, d, a_0, b_0, A, B)$;
4. $(n, d, a_0, b_0) \leftarrow (2n, c_2, CAT(a_1, a_0, n/2), CAT(b_1, b_0, n/2))$;
5. Go to 3.

Lemma 4. When Step 3 is about to be executed for the j^{th} time, let c_0 be the m-digit number which has been printed so far. Then:

1. $n = 2^j$.
2. $m = n/2$.
3. Exactly $n/2$ digits have been read from each of A and B. These digits are a_0 and b_0 respectively.
4. The on-line restriction has not yet been violated.
5. $a_0 \cdot b_0 = d \cdot 2^{n/2} + c_0$.

Proof. The proof is by induction on j. By inspection of the program *ONLINE*, the lemma holds for $j = 1$. The fact that it holds for $j > 1$ follows readily by direct calculation using the truth of the lemma for $j - 1$ and Lemma 3, part A. □

A few remarks are in order concerning the Turing machine implementation of these procedures. We may assume that the Turing machine has one work tape for each variable named in any of the programs. The recursive procedure calls are handled in the usual way using one additional tape as a pushdown store. At procedure entry, the old values of any local variables are saved on the pushdown store and the local variables are initialized; at procedure exit, the old values of the variables are restored.

The only possible difficulty is with the implementation of Step 2 of *ON*. However, *ON* itself is not recursive, so we simply make two independent copies of the Turing machine which implements *HALF* and run them in parallel as required.

3. Sparse Numbers

Call a natural number *k-sparse* if its binary representation contains at most k one-bits. The purpose of this section is to show that the time bound of Corollary 2 for on-line multiplication can be improved if either of the two multiplicands is sufficiently sparse. In particular, when one of the numbers is (log n)-sparse, the time reduces to just $O(n \log n)$. Paterson, M. Fischer and Meyer [7] show that any Turing machine $\mathcal{M}$ for on-line multiplication of a length n number by the particular (log n)-sparse constant

$$K_n = \sum_{2^i < n} 2^{2^i}$$

requires time $cn \log n$, where c depends only on $\mathcal{M}$. Thus, both bounds are optimal to within constant factors.

Let *ones*(b) denote the number of ones in the binary representation of b. We begin by describing an off-line procedure, *OFFSPARSE*(a, b), which forms the product of two length n numbers a and b by repeatedly shifting

and adding b into an accumulator. More precisely, let $(a_{n-1}\cdots a_0)_2$ be a binary representation of a, and let $I = \{ i \mid a_i = 1 \}$. Then $a = \sum_{i \in I} 2^i$, so $ab = \sum_{i \in I} b \cdot 2^i$. The last summation has $|I| = ones(a)$ terms, each of length at most $2n$, so it can be computed on a Turing machine in time at most $cn(ones(a) + 1)$ for some constant $c > 0$.

OFFSPARSE(a, b) always computes the correct answer but is fast (relative to other multiplication algorithms) only when a is sparse. Similarly, *OFFSPARSE*(b, a) is fast only when b is sparse. Both of these methods take time $O(n^2)$ in the worst case. By running them in parallel with any other (fast) multiplication procedure, we obtain a method whose running time is proportional to the minimum of the times for any of the three procedures, yielding:

Lemma 5. Let *OFF* be a multitape Turing machine which performs off-line integer multiplication of length n numbers within time $F(n)$, where F satisfies the conditions of Theorem 1. Let $G(n) = F(n)/n$. Then there is another multitape Turing machine *OFFSP* which performs off-line multiplication of two length n numbers a and b in time at most

$$c_1 n(\min\{ones(a), ones(b), G(n)\} + 1),$$

where $c_1 > 0$ is a constant.

Theorem 6. Let *OFF*, $F(n)$ and $G(n)$ be as in Lemma 5. Then there is a multitape Turing machine *ONLINESP* which performs integer multiplication, obeys the on-line restriction, and produces the n^{th} output digit ($n \geq 2$) in

$$c(n \min\{ones(a(n)), ones(b(n)), (\log n)G(n)\} + n \log n)$$

computational steps, where $a(n)$ and $b(n)$ are the low-order n digits of the arguments, and $c > 0$ is a constant.

Proof. The new on-line procedure is nearly identical to the one described in Section 2. The only difference lies in the operation of the off-line subprocedure.

Let *OFFSP* be the off-line multiplication procedure of Lemma 5. Let *ONSP*, *HALFSP*, and *ONLINESP* be procedures defined exactly as *ON*, *HALF*, and *ONLINE* are in Section 2, except that *OFFSP* is used as the off-line multiplication subprocedure instead of *OFF*. Let $NS(n, p, q)$, $HS(n, p, q)$, and $FS(n, p, q)$ be the maximum number of steps required by $ONSP(n)$, $HALFSP(n)$, and *OFFSP* respectively when multiplying n-digit integers a and b with $p \geq ones(a)$, $q \geq ones(b)$, and n equal to a power of 2. (Note that $FS(n, p, q) \leq c_1 n(\min\{p, q, G(n)\} + 1)$ by Lemma 5.) For definiteness, assume a is the on-line argument for the half-line computation. For arbitrary n, let $TS(n, p, q)$ be the time when the n^{th} output digit is produced by *ONLINESP*, assuming that the first n digits of the two inputs have p and q ones respectively.

Split the n-digit integers a and b as $a = a_1 \cdot 2^{n/2} + a_0$ and $b = b_1 \cdot 2^{n/2} + b_0$, $a_0, b_0 < 2^{n/2}$, and let $p_i = ones(a_i)$ and $q_i = ones(b_i)$ for $i = 1, 2$. By the recursive definitions of $ONSP(n)$, $HALFSP(n)$ and *ONLINESP*, and maximizing over all such n-digit integers a and b, the following relations hold.

$$NS(n, p, q) \leq \max\{HS(n/2, p_1, q_0) + HS(n/2, q_1, p_0) + FS(n/2, p_1, q_1) + c_2 n \mid p_0 + p_1 = p, q_0 + q_1 = q\}; \tag{4}$$

$$HS(n, p, q) \leq \max\{HS(n/2, p_0, q_0) + FS(n/2, p_0, q_1) + HS(n/2, p_1, q_0) + FS(n/2, p_1, q_1) + c_3 n \mid p_0 + p_1 = p, q_0 + q_1 = q\}; \tag{5}$$

$$TS(n, p, q) \leq \sum_{i=1}^{\lceil \log n \rceil} NS(2^i, p, q) + O(n). \tag{6}$$

Lemma 7. There is a constant c_4 such that

$$HS(n, p, q) \leq c_1 n \min\{p, q, (\log n)\cdot G(n)\} + c_4 n \log n$$

for all $n \geq 2$ with n equal to a power of 2 and all $p, q \geq 0$.

Proof. Proof is by induction on j, where $n = 2^j$. The base $j = 1$ is immediate by choosing c_4 large enough.

The induction step follows from Lemma 5 and relations (4) and (5) by a straightforward calculation. Choose $c_4 \geq c_1 + c_3$. Assume the bound of Lemma 7 holds for $HS(n/2, p, q)$ where $p, q \geq 0$ are arbitrary. Then

$$\begin{aligned} HS(n, p, q) \leq \max\{ & c_1(n/2)\min\{p_0, q_0, (\log n/2)\cdot G(n/2)\} + c_4(n/2)\log n/2 \\ & + c_1(n/2)(\min\{p_0, q_1, G(n/2)\} + 1) \\ & + c_1(n/2)\min\{p_1, q_0, (\log n/2)\cdot G(n/2)\} + c_4(n/2)\log n/2 \\ & + c_1(n/2)(\min\{p_1, q_1, G(n/2)\} + 1) \\ & + c_3 n \mid p_0 + p_1 = p, q_0 + q_1 = q\}. \end{aligned}$$

Note that by distributivity of + over min, it follows that for all integers $p_0, p_1, q_0, q_1, z_0, z_1$, $\frac{1}{2}(\min\{p_0, q_0, z_0\} + \min\{p_0, q_1, z_1\} + \min\{p_1, q_0, z_0\} + \min\{p_1, q_1, z_1\}) \leq \min\{p_0 + p_1, q_0 + q_1, z_0 + z_1\}$. Therefore,

$$\begin{aligned} HS(n, p, q) \leq & c_1 n \min\{p, q, (\log(n/2) + 1)\cdot G(n/2)\} \\ & + c_4 n \log n + (c_1 + c_3 - c_4)n. \end{aligned}$$

Since $F(n) \geq 2F(n/2)$, then $G(n) \geq G(n/2)$, so the induction step is proved. □

It is now easy to verify that there are constants c and c' such that

$$NS(n, p, q) \leq c'(n \min\{p, q, (\log n)\cdot G(n)\} + n \log n)$$

for all $n \geq 2$ and n equal to a power of 2; and

$$TS(n, p, q) \leq c(n \min\{p, q, (\log n)\cdot G(n)\} + n \log n)$$

for all $n \geq 2$. By the discussion of Section 2.2, it is clear that this on-line procedure can be implemented on a Turing machine whose running time satisfies conditions (4) - (6). This completes the proof of Theorem 6. □

Corollary 8. There is a multitape Turing machine which performs on-line integer multiplication within time

$$c(n \min\{ones(a(n)), ones(b(n)), (\log n)^2 \operatorname{loglog} n\} + n \log n)$$

where $a(n)$ and $b(n)$ are the first low-order n bits of the two inputs.

An interesting open question is whether or not the $n \log n$ overhead term can be eliminated from Theorem 6.

4. Other Applications

4.1. Generalized Linear Products.

The off-line to on-line conversion can be applied to other computations which can be loosely described as convolutional in nature. The generalized linear products defined in [3] are one such class. Let $a = (a_0, a_1, \ldots, a_m)$ and $b = (b_0, b_1, \ldots, b_n)$ be two vectors. The *linear product with respect to* $\otimes$ *and* $\oplus$, written $a \left[\begin{smallmatrix}\otimes\\ \oplus\end{smallmatrix}\right] b$, is a vector $c = (c_0, c_1, \ldots, c_{m+n})$, where

$$c_k = \bigoplus_{i+j=k} a_i \otimes b_j ,$$

$k = 0, \ldots, m+n$. For this to be meaningful, $a_i, b_j \in D$, $c_k \in E$ for some sets D and E, and $\otimes$ and $\oplus$ are functions,

$$\otimes : D \times D \to E,$$

$$\oplus : E \times E \to E, \quad \oplus \text{ associative.}$$

The only property needed by the on-line conversion is that the product of two length n vectors A and B be obtainable by additions ($\oplus$) from the four products $A_0 \boxed{\begin{matrix}\otimes\\ \oplus\end{matrix}} B_0$, $A_0 \boxed{\begin{matrix}\otimes\\ \oplus\end{matrix}} B_1$, $A_1 \boxed{\begin{matrix}\otimes\\ \oplus\end{matrix}} B_0$, and $A_1 \boxed{\begin{matrix}\otimes\\ \oplus\end{matrix}} B_1$, where $A_0 = (a_0,\ldots,a_{n/2-1})$, $A_1 = (a_{n/2},\ldots,a_{n-1})$, $B_0 = (b_0,\ldots,b_{n/2-1})$, and $B_1 = (b_{n/2},\ldots,b_{n-1})$. Generalized linear products clearly have this property, and in fact the on-line conversion is even simpler than for integer multiplication since there are no carries.

The result thus obtained is that a generalized linear product of two length n vectors can be computed on-line with at most $O(OP(n) \log n)$ uses of the basic operations $\otimes$ and $\oplus$, where $OP(n)$ bounds the number of basic operations needed to compute off-line the linear product of two length n vectors, providing that OP is monotone and satisfies $n \leq OP(n) \leq OP(2n)/2 \leq k\ OP(n)$ for some constant k.

Polynomial multiplication is an example of a **linear product**. Since polynomials of degree n with complex coefficients can be multiplied off-line using the Fast Fourier Transform [2] with only $O(n \log n)$ complex additions and multiplications, the corresponding on-line problem can be done with at most $O(n\ (\log n)^2)$ such scalar operations.

In case D and E are finite, the basic operations $\otimes$ and $\oplus$ can be computed in a constant amount of time on a Turing machine, and Theorem 1 then applies. A straightforward application is to the problem of on-line multiplication of polynomials with coefficients from some finite field.

Another application yields an $O(n \log n)$ method for on-line recognition of palindromes [3]. The on-line conversion is applied to a subroutine for performing the $\boxed{\begin{matrix}=\\ \wedge\end{matrix}}$ linear product off-line in time O(n) which in turn uses as a subroutine the pattern matching algorithm of Morris and Pratt [6]. Details can be found in [3].

4.2. Multiplication by a Constant.

One further problem we consider is that of on-line multiplication by a constant. Viewed as a function of two arguments, this is an example of a half-line computation. All the digits of the constant multiplier are available off-line. The digits of the multiplicand are read subject to the on-line restriction. The product can clearly be computed in time proportional to n, the length of the on-line input, where the constant of proportionality may depend on the length of the off-line input. Our methods give a constant of proportionality smaller than would be obtained using classical methods.

Let y be a k-bit constant (the off-line argument), and let x be an n-bit multiplicand (the on-line argument). We divide the bits of x into k-bit blocks and consider each block to represent a single digit of the base $b = 2^k$ representation of x. y can be regarded as a single digit in base b, and we form the product xy in the straightforward way by multiplying the single digit y by the successive base b digits of x, and adding in the carries.

The above method is on-line with respect to base b digits, for the i^{th} block of k bits (the i^{th} digit base b) is produced before any bits of the $i{+}1^{\text{st}}$ block are read. To make the method on-line with respect to the binary numbers, we need to specify how to compute the values $u + v$ and $y \cdot u$ on-line, where u and v are arbitrary k-bit numbers. The former may be computed by the ordinary method for addition and requires time $O(k)$. To do the latter on a Turing machine, even when we are allowed to do arbitrary preprocessing on y and k, we know of no way better than to use the on-line algorithm of Corollary 2, which requires time $O(k\ (\log k)^2\ \text{loglog}\ k)$. This has the property that when $n < k$, the time drops to $O(n\ (\log n)^2\ \text{loglog}\ n)$.

x contains $\lceil n/k \rceil$ blocks of at most k bits each, so we have proved:

Theorem 9. There exists a Turing machine M and a constant c such that for each $k \geq 4$ and every n, if x is a number of length n and y is a number of length k, then M computes xy half-line with respect to x and runs in time $\leq cn(\log k)^2 \operatorname{loglog} k$.

Acknowledgement

The authors are grateful to M. Paterson and A. Meyer for several helpful discussions.

References

1. Cook, S.A. and Aanderaa, S.O. On the minimum computation time of functions. *Trans. Amer. Math. Soc. 142* (1969), 291-314.

2. Cooley, J.W. and Tukey, J.W. An algorithm for the machine calculation of complex Fourier series. *Math. Comp. 19* (1965), 297-301.

3. Fischer, M.J. and Paterson, M.S. String matching and other products. *MAC Technical Memorandum 41*, M.I.T. Project MAC, Cambridge, Mass. (January 1974); *Proc. Am. Math. Soc. Symp. on Complexity of Real Computational Processes* (1974), to appear.

4. Hartmanis, J. and Stearns, R.E. On the computational complexity of algorithms. *Trans. Amer. Math. Soc. 117* (1965), 285-306.

5. Hennie, F.C. On-line Turing machine computations. *IEEE Trans. Electronic Computers EC-15*, 1 (1966), 35-44.

6. Morris, J.H. and Pratt, V.R. A linear pattern-matching algorithm. *TR-40*, Computer Center, Univ. of Calif. at Berkeley (June 1970).

7. Paterson, M.S., Fischer, M.J. and Meyer, A.R. An improved overlap argument for on-line multiplication. *MAC Technical Memorandum 40*, M.I.T. Project MAC, Cambridge, Mass. (January 1974); *Proc. Am. Math. Soc. Symp. on Complexity of Real Computational Processes* (1974), to appear.

8. Schönhage, A. and Strassen, V. Schnelle Multiplikation grosser Zahlen. *Computing 7* (1971), 281-292.

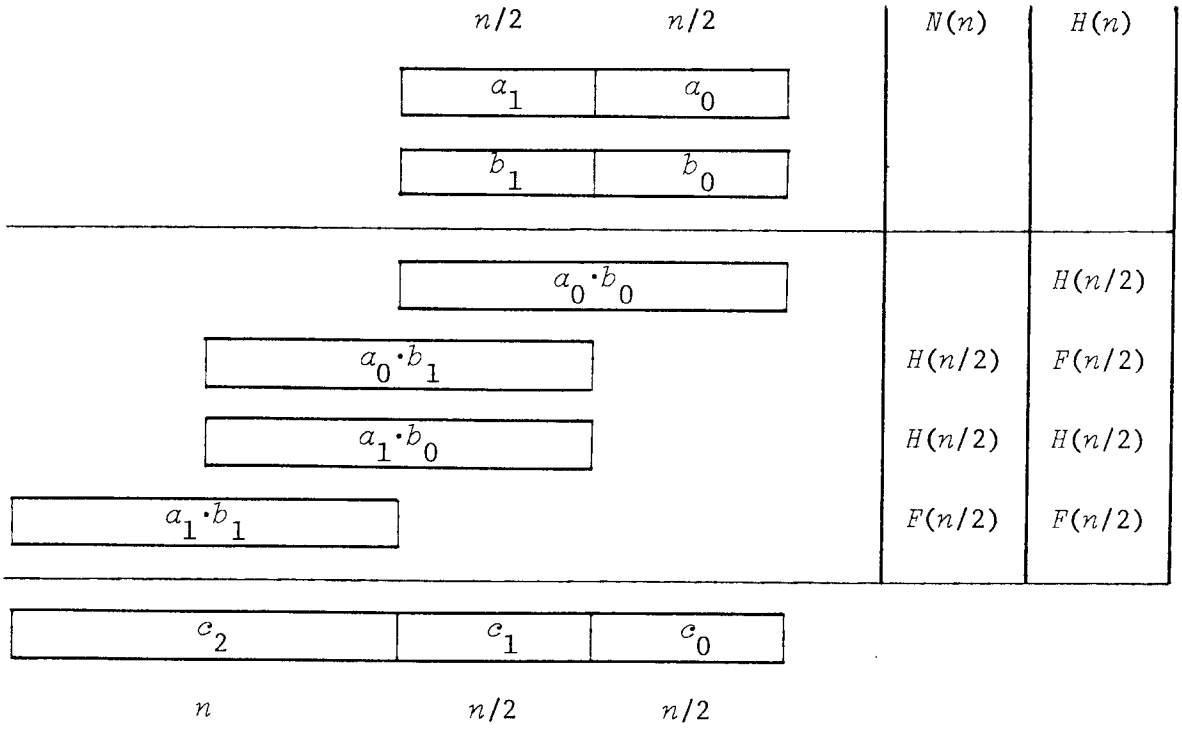

Figure 1. Recursively computed multiplication.

BIBLIOGRAPHIC DATA SHEET	1. Report Nos. GJ34671 + N00014-70-A-0362-0006 MAC TM- 45	2.	3. Recipient's Accession No.
4. Title and Subtitle Fast On-Line Integer Multiplication			5. Report Date : Issued May 1974 6.
7. Author(s) Michael J. Fischer and Larry J. Stockmeyer			8. Performing Organization Rept. No. MAC TM- 45
9. Performing Organization Name and Address PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139			10. Project/Task/Work Unit No. 11. Contract/Grant Nos. GJ34671 N00014-70-A-0362-0006
12. Sponsoring Organization Name and Address Office of Naval Research Department of the Navy Information Systems Program Arlington, Va 22217 Associate Program Director Office of Computing Activities National Science Foundation Washington, D. C. 20550			13. Type of Report & Period Covered : Interim Scientific Report 14.
15. Supplementary Notes			

16. Abstracts : A Turing machine multiplies binary integers on-line if it receives its inputs low-order digits first and produces the jth digit of the product before reading in the (j+1)st digits of the two inputs. We present a general method for converting any off-line multiplication algorithm which forms the product of two n-digit binary numbers in time F(n) into an on-line method which uses time only O(F(n) log n), assuming that F is monotone and satisfies n - F(n) - F(2n)/2 - kF(n) for some constant k. Applying this technique to the fast multiplication algorithm of Schonhage and Strassen gives an upper bound of $O(n (\log n)^2 \log\log n)$ for on-line multiplication of integers. A refinement of the technique yields an optimal method for on-line multiplication by certain sparse integers. Other applications are to the on-line computation of products of polynomials, recognition of palindromes, and multiplication by a constant.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Approved for Public Release; Distribution Unlimited	UNCLASSIFIED	24
	20. Security Class (This Page) UNCLASSIFIED	22. Price