Распараллеливание алгоритмов умножения чисел многократной точности

Е.Г. Качко

Харьковский национальный университет радиоэлектроники

Операция умножения для длинных чисел остается объектом исследования математиков и программистов с точки зрения минимизации вычислительной сложности. Для оценки вычислительной сложности этой операции традиционно используется количество элементарных операций. При этом не учитываются свойства современных процессоров, такие как суперскалярность и многоядерность. В работе рассмотрены современные алгоритмы умножения с точки зрения возможности их распараллеливания, сделана теоретическая оценка вычислительной сложности и экспериментальная проверка с помощью Open MP и TBB. Выполнен анализ полученных результатов

1. Введение

Практически все современные несимметричные криптографические алгоритмы используют операцию умножения для длинных чисел. При этом длина сомножителя все время увеличивается. Так, для алгоритма RSA в качестве сомножителей могут использоваться числа длиной до 4096 бит, в дальнейшем и этот диапазон может быть расширен. Так как операцию умножения необходимо выполнять многократно при возведении в степень, а последняя операция используется и в алгоритмах RSA, и в алгоритмах на основе эллиптических кривых, то даже незначительное уменьшение вычислительной сложности для этой операции приведет к существенному уменьшению времени для формирования и проверки цифровой подписи.

Многие математики и программисты неоднократно возвращались к теме минимизации вычислительной сложности операции умножения. Из наиболее распространенных алгоритмов следует назвать алгоритмы: умножения в столбик (так называемый школьный метод), Карацубы [1], Шёнхаге и Штрассена [2], Фюрера [3]. В таблице 1 указаны значения вычислительной сложности для каждого из выше названных алгоритмов

Таблица 1. Вычислительная сложность основных алгоритмов умножения

Алгоритм	**Вычислительная сложность**
Вычисление в «Столбик»	$O(n^2)$
Карацубы и Офмана алгоритм	$O(n^{\log 3})$
Шёнхаге и Штрассена алгоритм	$O(nlognloglogn)$ $(2^{15}.. 2^{17})$
Фюрера алгоритм	$O(nlogn2^{O(\log^* n)})$[1]

Следует заметить, что последний алгоритм предполагает представление сомножителей как полиномы с системой счисления 2^{16} для длин чисел до 65536 бит включительно. Для 16 битных процессоров это удачное представление, для 32-битных и 64 битных – такое представление приведет к существенной потере производительности. В дальнейшем будет рассмотрен вопрос о возможности перехода на систему счисления 2^{32}, в данной работе этот алгоритм не рассматривается.

Для оценки вычислительной сложности алгоритмов все авторы используют число элементарных операций, которые необходимо выполнить. При этом не учитывается ни возможность выполнения самих операций параллельно за счет суперскалярности процессора, ни возможность выполнения отдельных функций параллельно, ни свойства кэша. Учет этих особенностей

[1] Обозначение log*n означает loglog…log n раз по количеству рекурсий, которые необходимо выполнить

может существенно влиять на вычислительную сложность, как в сторону уменьшения, так и в сторону увеличения.

Вопросы параллельного выполнения операции умножения рассматривались и ранее. В связи со спецификой вычислительных средств этот вопрос рассматривался в условиях использования распределенных вычислений для кластерных систем [4]. В работе [5] предложен параллельный алгоритм умножения, основанный на предвычислениях, который приводит к значительному увеличению количества циклов.

Цель данной работы – исследование основных методов умножения для чисел многократной точности с точки зрения возможности их параллельного выполнения.

2. Математическая постановка задачи

Даны 2 числа:

$X = \sum_{i=0}^{n-1} x_i * B^i$ $\qquad$ $Y = \sum_{i=0}^{n-1} y_i * B^i$,

где

B - основание системы счисления: x_i, y_i – "цифры" сомножителей.

Для современных процессоров $B=2^{32}$ или 2^{64}. Все экспериментальные данные получены при $B=2^{32}$.

Необходимо вычислить $Z=X * Y$, обеспечив минимальную вычислительную сложность за счет параллельного выполнения на многоядерном процессоре.

3. Определение показателей эффективности для алгоритмов умножения

3.1 Вычисление в «столбик»

3.1.1 Алгоритм 1 (А1)

1. Обнуление произведения.
2. Для всех «цифр» второго сомножителя
   a. Вычисление произведения на первый сомножитель
   b. Сложение с текущим значением произведения.

Для простоты будем считать, что количество "цифр" для обоих сомножителей одинаково и равно n. В этом случае потребуется n операций умножения n-разядного числа на одно разрядное и сложение полученных результатов. Обозначим время выполнения операции умножения t_m, операции сложения t_a. Время выполнения алгоритма вычисления в столбик в последовательном режиме составляет:

$$T_{A1} = n^2 * (t_m + t_a) \qquad (1)$$

3.1.2 Алгоритм 2 (А2). «Быстрый столбик» [6]

Для всех цифр произведения вычисляем значение цифры по формуле:

$$Z_s = \sum_{i=0}^{i=s} x_{s-i} * y_i; \qquad s = 0..n-1;$$
$$Z_s = \sum_{i=s-n+1;k=n-1}^{i=n-1,k=n-i} x_k * y_i; \quad s = n..2n-1; \qquad (2)$$

Фактически «быстрый» столбик не что иное, как вычисление значения свертки для всех цифр результата, если «цифры» - это коэффициенты многочлена.

Для этого алгоритма необходимо выполнить такое же количество операций сложения и умножения, как для Алгоритма 1, но в другом порядке поэтому $T_{A2} = T_{A1}$.

Так как количество ядер для современных процессоров небольшое, а цель данной работы – дать практические рекомендации по использованию параллельных вычислений при умножении длинных чисел, в работе не рассматривается вариант неограниченного параллелизма, т.е. бесконечного числа параллельных ветвей.

Пусть число параллельных ветвей p, причем $p < n/2$ и n кратно p.

3.1.3 Параллельный алгоритм 1 (PA1).

Для обеспечения максимальной грануляции и равномерного распределения нагрузки разделим один из сомножителей на порции равной длины. В этом случае каждый поток может выполняться параллельно. Если не учитывать накладных расходов, связанных с использованием потоков, то общее время выполнения PA1 равно:

$$T_{PA1} = n^2 * (t_m + t_a) / p \quad (3)$$

Значения ускорения и эффективности соответственно равны:

$$S_{PA1} = \frac{T_{A1}}{T_{PA1}} = p; \quad E_{PA1} = \frac{S_{PA1}}{p} = 1. \quad (4)$$

3.1.4 Параллельный алгоритм 2 (PA2)

Так как сложность вычисления «цифры» зависит от ее номера, будем использовать динамический режим распределения итераций цикла. В этом случае потребуется специальный механизм учета переносов.

3.3 Алгоритм Карацубы и Офмана (KO)

3.3.1 Последовательный алгоритм (KO)

Для определения числа операций запишем формулы для вычисления: значения произведения по этому методу. Пусть XL - младшая часть первого сомножителя, а XH - его старшая часть, т.е $X = XH * B^{n/2} + XL.$. Пусть YL - младшая часть первого сомножителя, а YH - его старшая часть, т.е $Y = YH * B^{n/2} + YL.$. Обозначим ZL, ZH младшую и старшую часть произведения *Z*. Тогда:

$$Z = C * B^n + E * B^{n/2} + A \quad (5)$$

где:

$$A = XL * YL; \quad C = XH * YH; D = (XL + XH) * (YL + YH); \quad E = D - (A + C). \quad (6)$$

Заметим, что значение коэффициента E не отрицательно. В работе [7] предложена формула вычисления значения произведения:

$$Z = (B^n + B^{n/2})XH*YH + B^{n/2}* (XH-XL)*(YL-YH) + (B^{n/2}+1)*XL*YL, \quad (7)$$

Формула (7) эквивалентна (5), но второе слагаемое в (7) может быть как положительным, так и отрицательным, первое и третье слагаемое вычисляется сложнее (дополнительные операции сложения), поэтому в дальнейшем рассматривается формула (5). Для исключения рекурсии для вычисления *A, C, E* используем умножение «в столбик» (*A1*). Общее время выполнения в последовательном режиме равно:

$$T_{KO} = (3n^2/4 + n + 1)t_m + (3n^2/4 + 11n/2 + 6)t_a. \quad (8)$$

3.3.1 Параллельный алгоритм (PKO)

Схема параллельного вычисления для $p \geq 4$ представлена в таблице 2.

Таблица 2. Параллельное вычисление произведения. Алгоритм Карацубы Хофмана

Ядро процессора			
1	2	3	4
$ZL = XL * YL$	$ZH = XH * YH$	$XL + XH$	$YL + YH$
$ZL + ZH$		$D = (XL + XH) * (YL + YH)$	
$E = D - (ZL + ZH)$			
$Z += E$			

Операции, которые выполняются параллельно, определены в одной строке таблицы. В этом случае время выполнения параллельного алгоритма равно:

$$T_{PKO}^4 = (n^2/4 + n + 1)t_m + (n^2/4 + 5n + 5)t_a \quad . \qquad (9)$$

Ускорение и эффективность для алгоритма Карацубы и Офмана для $p \geq 4$:

$$S_{PKO}^4 = \frac{T_{KO}}{T_{PKO}^4} = \frac{(3n^2/4 + n + 1)t_m + (3n^2/4 + 11n/2 + 6)t_a}{(n^2/4 + n + 1)t_m + (n^2/4 + 5n + 5)t_a}, E_{PKO}^4 = S_{PKO}^4/4; \qquad (10)$$

Для 2-х ядер соответствующие формулы для времени выполнения и эффективности:

$$T_{PKO}^2 = (n^2/2 + n + 1)t_m + (n^2/2 + 4n + 4)t_a$$

$$S_{PKO}^2 = \frac{T_{KO}}{T_{PKO}^2} = \frac{(3n^2/4 + n + 1)t_m + (3n^2/4 + 11n/2 + 6)t_a}{(n^2/2 + n + 1)t_m + (n^2/2 + 4n + 4)t_a}, E_{PKO}^2 = S_{PKO}^2/2 \qquad (11)$$

3.4 Алгоритм Шёнхаге и Штрассена (SS). Последовательный и параллельный режимы

Этот метод основан на замене умножения больших чисел на умножение полиномов, для вычисления которых используется дискретное преобразование Фурье. Для вычисления каждой цифры полинома достаточно в качестве модуля использовать число, больше или равное $2nB^2$. Для того чтобы не выполнять вычисления с числами, превосходящими 2^{64}, будем использовать простые числа, произведения которых превосходят заданный модуль. Пусть максимальное число бит сомножителя $n_{Bits} = 16384$[2]. Такая длина выбрана из практических соображений. В криптографии числа большей длины не используются. Минимальный размер модуля, достаточный для таких сомножителей, равен 2^{74}. Этот модуль гарантируется произведением трех простых чисел, меньших, чем 2^{32}, но близких к этому значению. Выбор простых чисел гарантирует наличие корней и обратного элемента. Так как значение модуля зависит только от длины сомножителей, эти числа могут быть заранее вычислены для каждой из используемых длин.

3.4.1 Алгоритм SS (Последовательный режим)[7]

1. Предвычисления (вычисление простых составляющих модуля, их инверсий, корней и их степеней, и коэффициентов для обратного преобразования числа из смешанной системы счисления).
2. Для всех простых чисел
   a. приведение первого сомножителя по модулю;
   b. формирование преобразования Фурье для 1 сомножителя;
   c. приведение второго сомножителя по модулю;
   d. формирование преобразования Фурье для 2 сомножителя;
   e. покомпонентное умножение по модулю;
   f. формирование обратного преобразования Фурье.
3. Преобразование для получения результата

[2] Современные криптографические алгоритмы не используют больших длин

В виду громоздкости общей формулы для определения времени выполнения, она не приведена. Рассмотрим способы параллельного вычисления для алгоритма SS. При наличии не менее трех ядер каждое ядро может выполнить полный цикл обработки для одного простого числа. Так как предварительная обработка данных не учитывается, а последовательная часть программы в этом случае невелика, то ожидаемое ускорение приблизительно равно 3. Для двух ядерного процессора первое ядро выполняет преобразования для первого и второго простого числа, а второе – для третьего. В этом случае ожидаемое ускорение примерно равно 1.5

4. Реализация алгоритмов умножения

Для реализации описанных выше алгоритмов использовался процессор Intel (R) Core (TM)2 Duo CPU E6850 @3.00GHz. Среда разработки: Microsoft Visual C++ 2008.Компиляторы: Intel(R) C++ Compiler Professional Edition 11.1, Microsoft Visual C++ 2008. Для обоих компиляторов использовался режим оптимизации по времени. Среды для разработки параллельных программ: Open MP, версия 3.0 (май 2008), TBB 3.0. Были реализованы функции для всех описанных выше алгоритмов в последовательном и параллельном режимах. Для сравнения временных характеристик в качестве эталонной библиотеки использовалась библиотека Miracl[3] (версия 5.4.3 от 29.10.10). Обе библиотеки откомпилированы в режиме без использования ассемблерных вставок. Все функции реализованы для сомножителей одинаковой длины в диапазоне от 512 до 16384 бита. Длина сомножителей в таблицах задается в 32-битных словах. В таблицах результаты определяют время выполнения функций (секунды). Результаты приведены в таблицах 3-6.

Таблица 3. Вычисление «в столбик»

Len	Intel(R) C++ Compiler				Visual C++			
	Miracl	A1	PA1, Open MP	PA1, TBB	Miracl	A1	PA1, Open MP	PA1, TBB
16	2.79E-06	2.23E-06	2.23E-06	6.12E-05	3.91E-06	3.07E-06	3.35E-06	7.29E-05
32	6.94E-06	4.75E-06	5.31E-06	7.57E-05	8.94E-06	7.82E-06	7.26E-06	6.03E-05
64	2.43E-05	1.53E-05	1.68E-05	8.41E-05	3.10E-05	2.74E-05	2.23E-05	9.92E-05
128	9.33E-05	5.64E-05	6.22E-05	0.00012	0.000119	0.000109	8.18E-05	0.000151
256	0.00037	0.00022	0.00024	0.00033	0.000472	0.000415	0.00032	0.000318
512	0.00154	0.00085	0.00098	0.00113	0.00187	0.0017	0.00126	0.001037

Таблица 4. «Быстрый столбик»

Len	Intel(R) C++ Compiler			Visual C++		
	A1	A2	PA2,Open MP	A1	A2	PA2,Open MP
16	2.23E-06	4.75E-06	1.54E-05	3.07E-06	2.79E-06	8.10E-06
32	4.75E-06	5.59E-06	1.62E-05	7.82E-06	6.42E-06	1.03E-05
64	1.53E-05	5.03E-06	1.20E-05	2.74E-05	1.87E-05	1.93E-05
128	5.64E-05	5.59E-05	3.88E-05	0.000109	6.42E-05	4.55 E-05
256	0.00022	0.000209	0.000142	0.000415	0.00024	0.00015
512	0.00085	0.000814	0.00055	0.0017	0.00094	0.00057

Таблица 5. Алгоритм Карацубы и Офмана

Len	Intel(R) C++ Compiler			Visual C++		
	A1	KO	PKO,Open MP	A1	KO	PKO,Open MP
16	2.23E-06	2.79E-06	4.47E-06	3.07E-06	3.63E-06	9.77E-06
32	4.75E-06	5.59E-05	5.59E-06	7.82E-06	6.42E-06	1.42E-05
64	1.53E-05	1.54E-05	1.23E-05	2.74E-05	1.65E-05	1.98E-05
128	5.64E-05	5.17E-05	3.72 E-05	0.000109	5.59 E-05	4.61E-05
256	0.00022	0.000208	0.000137	0.000415	0.000208	0.000150
512	0.00085	0.000751	0.000510	0.0017	0.000802	0.000542

[3] http://www.shamus.ie/index.php?page=Downloads

Таблица 6. Алгоритм Шёнхаге и Штрассена

Len	Intel(R) C++ Compiler			Visual C++		
	Miracl (SS)	SS	PSS, Open MP	Miracl (SS)	SS	PSS, Open MP
16	3.88E-05	3.57E-05	3.24E-05	3.94E-05	3.6E-05	3.18E-05
32	8.46E-05	7.23E-05	5.11E-05	8.57E-05	7.96E-06	5.89E-05
64	0.000185	0.000160	0.000109	0.000194	0.000176	0.000125
128	0.000409	0.000357	0.000241	0.000433	0.000392	0.000267
256	0.000915	0.000786	0.000531	0.000957	0.000866	0.000587
512	0.00202	0.00174	0.00116	0.00210	0.0014	0.00127

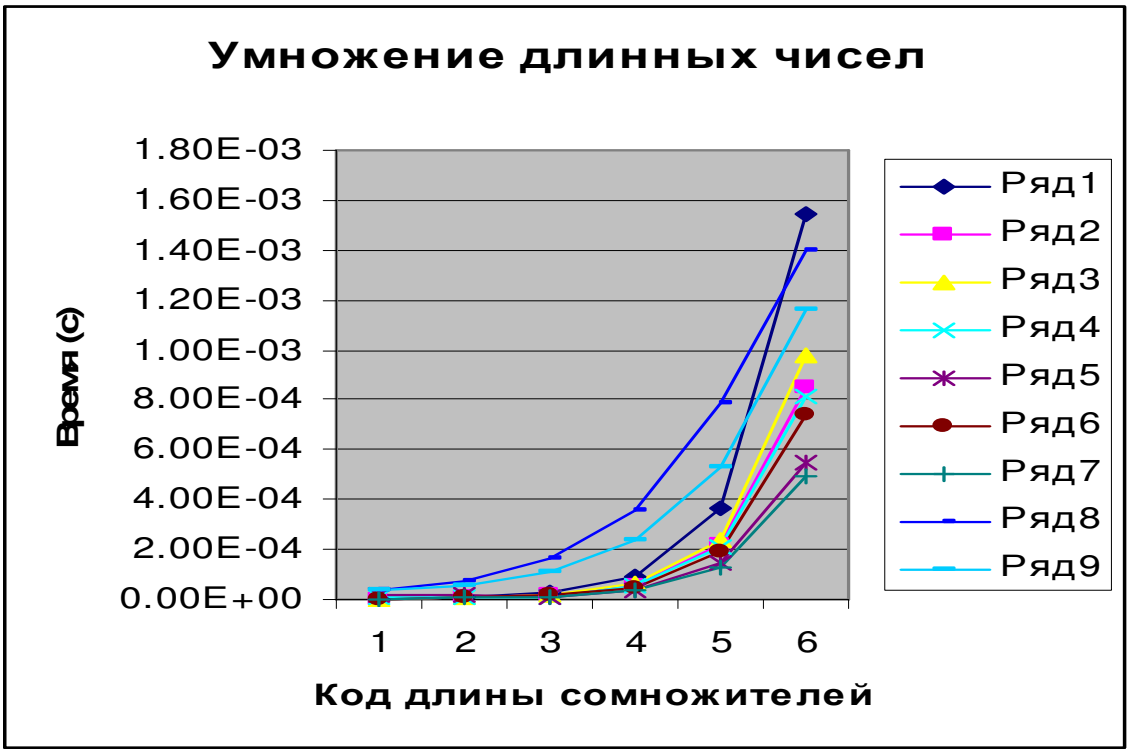
Как следует из таблицы 3, разработанная библиотека (алгоритм A1) более эффективна, чем эталонная (Miracl). Ускорение составляет от 32 до 62% для компилятора Intel C++ и от 13 до 78% для компилятора Visual C++. Сравнение эффективности кодов, построенных с помощью двух компиляторов, показывает, что первый компилятор строит более эффективный код (ускорение составляет от 9 до 63%). Параллельное выполнение кода, реализованное с помощью Open MP и TBB, не дает выигрыша. Так как использование TBB не дает выигрыша по сравнению с Open MP для этого типа задач, в дальнейшем результаты использования этой технологии для рассматриваемого класса алгоритмов не приводятся.

Как следует из таблицы 4, Алгоритм A2 работает медленнее алгоритма A1 для всех длин. И, хотя параллельный режим (алгоритм PA2), позволяет уменьшить время выполнения, этот метод продолжает быть медленнее, чем A1.

Метод Карацубы и Офмана (таблица 5) опережает обычный метод на длине 64 слова (2048 бит). Параллельный режим позволяет получить ускорение до 50%.

Алгоритм Шёнхаге и Штрассена (Таблица 6) проигрывает даже простому столбику на всех рассмотренных длинах. Наилучшим сточки рения вычислительной сложности является первый алгоритм для длин сомножителей меньше 2048 и параллельный алгоритм Карацубы и Офмана для всех остальных длин.

Для всех алгоритмов компилятор Intel C++ создает более эффективный код. На рисунке 1 представлены графики зависимостей времени вычислений от длины сомножителей для компилятора Intel C++ для всех рассмотренных выше алгоритмов.



Ряд 1 соответствует функции умножения («в столбик») базовой библиотеки (Миракл). Эта функция менее эффективна по сравнению со всеми функциями, кроме варианта использования функции для алгоритма Шёнхаге и Штрассена для длины сомножителей не более 8192.

Ряды 2 и 3 соответствуют функции умножения («в столбик»), последовательному и параллельному вариантам соответственно. Эффективности обоих методов практически совпадают и выше эффективности функции умножения (ряд 1)..

Ряды 4 и 5 соответствуют функции умножения («быстрый столбик»), для последовательного и параллельного режимов. Эти графики находятся ниже графиков для рядов 2, 3 для всех длин.

Ряды 6 и 7 соответствуют алгоритму Карацубы и Офмана (последовательный и параллельный режим). Этому алгоритму соответствует минимальная вычислительная сложность для длин сомножителей не менее 2048 бит.

Ряды 8 и 9 – графики для алгоритма Шёнхаге и Штрассена. Эти графики пересекают график для функции умножения (Миракл) для длины сомножителей 16384 для последовательного режима и для длины 8192 для параллельного режима.

Ожидаемые зависимости скорости вычислений от длин не совпадают с полученными. Так, для последовательного «столбика» ожидается зависимость (1), а получается близкая к (3). Это связано с суперскалярностью современных процессоров, когда одновременно выполняется несколько команд. Тоже для других алгоритмов.

Литература

1. Карацуба А., Офман Ю. Умножение многозначных чисел на автоматах // Доклады Академии Наук СССР. — 1962. — Т. 145. — № 2
2. Arnold Schönhage and Volker Strassen, Schnelle Multiplikation grosser Zahlen, Computing 7 (1971), 281{292
3. Martin Förer. Faster integer multiplication. Proceedings of the 39th ACM Symposium on Theory of Computing, pages 57–66, 2007.
4. *Fagin B. S.* Large integers multiplication on massively parallel processors. http://barryfagin.name/Papers/FMPC90.htm.
5. Efficient Parallel Multiplication Algorithm for Large Integers. http://www.springerlink.com/content/hqup5uxp5np2dhmy/fulltext.pdf
6. Василенко О. Н. Теоретико-числовые алгоритмы в криптографии. http://window.edu.ru/window_catalog/files/r23845/book.pdf
7. Г. Нуссбаумер. Быстрое преобразование Фурье и алгоритмы вычисления сверток http://www.toroid.ru/nussbaymerG.html
8. Кнут Д. Искусство программирования для ЭВМ. т. 2. Получисленные алгоритмы. Мир, М. 1977., с. 314.