2001 年:多倍長整数計算研究記録

梅谷 武

作成：2001-01-10 更新：2006-01-12

2001 年:多倍長整数計算研究記録
IMS:20010110001; NDC:418; keywords:多倍長整数計算, 整数;

目　　次

1　2000 年 10 月以降のまとめ (2001.1.20)

多倍長整数計算の研究はだいぶ間があいてしまいました。研究日誌の最後の日付を見ると 10 月 25 日になっています。この頃の状況を日誌とプログラムでたどってみたいと思います。まず、10 月 6 日に 32bit 語長のストラッセン-ショーンハーゲ法 (以後 SS 法と書く) をまず C++ で実装して正しく計算できることを確認しています。

10 月 11 日にこれがあまりにも遅いのでこれを高速化するために、古典算法からすべてをアセンブラで書き

直す作業を始めています。古典算法のアセンブラ化の過程で、いくつかのやり方を試して、クヌース本の算法をそのまま x86 に移植するのが一番速いことを実験で確認しています。古典算法のアセンブラ化は 10 月 13 日にとりあえず終了していますが、その日の日誌には除算だけはクヌース本のそのままの移植が難しいので、書きやすいやり方に変更したと書いてあります。32bit 語長 SS 法のアセンブラ化は 10 月 23 日に終了したのですが、その日の古典算法と SS 法との性能比較実験の記録は次のようになっています。

表 1　除算試験 480 回× 10 による性能比較

古典算法	**SS 法**
5 秒	98 秒

除算試験となっていますが、これは乗算と除算が同時に試験されるようになっているものです。除算は同じルーチンを使っていますので、この差は乗算の違いだけで生じていることになります。

この SS 法の遅さの原因を考えていくとやはり、SS 法の型が $K = 2^k$ として、

$$(n, m, \zeta) = (K, 2^{2K} + 1, 2^4)$$

となっていることにたどりつきます。(n, m, ζ) 型 の SS 法においては、長さ n の FFT と法 m の整数演算を行いますが、特に後者の演算に時間がかかりすぎるということが最大の原因です。10 月 25 日の日誌に最終的に次のような結論にたどりついたという記述があります。

命題 1.1 (ストラッセン-ショーンハーゲ法の計算量) ストラッセン-ショーンハーゲ法の計算量は、その型 *(n,m, ζ)* が与えられたときに

i. 長さ n の *FFT* 算法の計算量
ii. 法 m の整数演算の計算量

により決定される。

これはある意味ではあたりまえで、原論文はこの計算量を求めて乗算の最適な算法であると結論しているわけです。ただ、実装の対象となる計算機を決めたときに実行時間をクロック単位で見積もるという作業の必要性を認識するにいたったという意味で、これを強調しておくことにします。

仕事のスケジュールが入ったこともあって、結果的にここまでが昨年の研究成果ということになりました。

2　加減算試験 (2001.1.20)

世紀があらたまって、なおかつ昨年末から開発していた文書整形システム XeX がなんとか使えるレベルになったので、C++ による多倍長整数計算の研究をそろそろ再開しようと思います。

昨年 11 月以降も、仕事の合間に Internet 上の資源調査や文献調査は定期的にやっていたのですが、そこで幸運にも次のステップを踏み出すために非常に重要な資源である NTL[N1] を発見しました。これは C++ でかかれた数論のためのライブラリなのですが、１年ぐらい前に見たときには配布制限されていたので、これは使わないことにしていたのですが、今回見たら配布条件が変更されていて、GPL(GNU General Public License) で配布されるようになっていました。

今日は馴らし運転ということで NTL を実際に使ってみることにしました。1 を m 回だけ +1 した後、m 回だけ-1 するという簡単な加減算のプログラムで、UBASIC、NTL、S0 を比較するという実験を行いました。m=10,000,000 のときに結果を次に記します。

表 2　加減算試験１ (m=10,000,000)

	UBASIC	NTL	S0
1 回目	61 秒	42 秒	108 秒
2 回目	62 秒	42 秒	109 秒
3 回目	61 秒	42 秒	108 秒

UBASIC と NTL の差は、インタプリタとコンパイラの違いということで納得できますが、年が明けたばかりというのに、我 S0 の 108 という数字が煩悩の数のように見えます。加減算でこれだけ差がつくというのはまだまだ甘いということですね。一歩前へ進むための方向性がわかったということで今日は満足することにしましょう。

3 加減算試験 (その２) (2001.1.27)

NTL の圧倒的な速さにえらく刺激されてしまいました。少し気合を入れて、S0 のチューンアップ方法をあれこれ考えました。まず、NTL はすべて C++ で記述されているのですが、S0 では Pentium の最適化コードをアセンブラで書くと決めて、その方針をまとめました。

「多倍長整数算法の Pentium による最適化」

今週は足し算と引き算を NTL 並にしようというのを目標にしたのですが、実験していて気がついたのは、NTL ではデータが可変長であるということです。S0 は最初から静的にメモリ割り当てするという設計方針でしたので、データは固定長になっています。このため公平な評価を行なうことが難しいのです。評価プログラムによって結果が変わってくるということになります。最適化後の加減算試験１の結果を次に示します。

表 3　加減算試験１ (m=10,000,000)

	UBASIC	NTL	S0
1 回目	61 秒	42 秒	55 秒
2 回目	62 秒	42 秒	55 秒
3 回目	61 秒	42 秒	55 秒

UBASIC を抜いたのですが、まだ NTL よりも遅いように見えます。ところが S0 のデータ長を最短にすると大体 20 秒程度で計算できてしまいます。

いろいろなプログラムで実験してわかったことは、データ長を最適化した場合、S0 は NTL の倍ぐらい速く、データ長が冗長であった場合、NTL の方が速くなるときもある、ということです。ここまではあっさりとできてしまったのですが、次の乗除算はなかなか手ごわいと予想されます。あまり高望みすると行き詰まりそうなので、最初は古典算法だけを最適化することにしました。

4 構造改革 (2001.2.10)

NTL との性能比較をより精密にしようと新しいベンチマークプログラム SNBench を作成しました。これはある特定の処理だけの性能比較ができるようにしたものです。これにより何が原因で差が生じているのかがはっきりわかるようになりました。このテストをやっているうちに S0 の設計上の問題点が明らかになってきました。それは、データ配列の無効部分にすべて 0 をつめておくという原則を作り、その原則に依存するプログラムを作ってしまったことにあります。これにより桁上げや桁揃えのときに次数を大きくするだけで、大きくなった部分を 0 で埋める作業が必要がなくなったのですが、逆に、すべての処理の最初にデータ無効部分

を 0 で埋めるという処理が必要になってしまいました。そしてこの処理が性能にかなり影響していることが SNBench でわかってきたのです。

実はこのことは最初からわかっていたのですが、なるべく楽をしようという誘惑に打ち勝てなくてずっとそのままにしてあったことなのです。いわゆる「問題の先送り」と呼ばれているものと同類ことではなかったかと思います。今回 NTL の速さと完成度に大変な刺激を受けました。NTL のおかげでいままでになかったような高いモチベーションをもつことができるようになったと思います。これが「市場競争の原理」と呼ばれているものなのでしょう。やる気がでてきたおかげで、データ構造を変更し、全ソースを見直すと同時に必要な修正を加え、デバッグするという作業、すなわち「構造改革」を 2 週間かかって何とかやり遂げることができました。

構造改革後の SNBench の結果は次のようになりました。

表 4　SNBench Ver.0.01(3 回平均, 秒単位)

No.	項目	S0	NTL
1	加算	17.7	24.3
2	減算	17.3	19.0
3	加減算 1	19.3	20.0
4	加減算 2	7.5	15.3
5	乗算	37.8	42.3
6	除算	8.8	11.0
7	乗除算 1	11.0	22.6
8	乗除算 2	15.4	28.3
9	左シフト	13.2	19.6
10	右シフト	12.9	18.3
11	シフト 1	4.3	7.6
12	シフト 2	1.4	2.0
13	論理和	14.3	22.6
14	論理積	14.3	21.6
15	メルセンヌ 1	33.2	50.0
16	メルセンヌ 2	17.2	11.0

上の結果はある程度満足できるものだと思いますので、プログラムの最適化作業には一旦区切りをつけようと思います。項目 15 と 16 のメルセンヌ 1、2 は、p=4253 に対するメルセンヌ素数の判定を行なうもので、前者は割り算を使用し、後者は割り算をシフトと論理積に置き換えたものです。メルセンヌ 2 はほとんど乗算の性能試験といっていいものです。唯一この項目で劣っていますが、これは S0 が n ビットの乗算を $O(n^2)$ の計算量で行なう古典算法を使っていることによります。

これからいよいよ高速乗算法の組み込みを行いたいと思いますが、離散フーリエ変換を使う方法はしばらく封印して、まず、簡単な割に効果的であると思われる Karatsuba 法から実験してみたいと思います。これがうまくいったらそれを Ver.0.01 として GPL で公開いたします。

5　Karatsuba 法の実験 (2001.2.16)

もっとも簡単な高速乗算法である Karatsuba 法 [S3] の実験を行いました。SNBench の結果を示します。

表 5　SNBench Ver.0.01(3 回平均, 秒単位)

No.	項目	S0	NTL
1	加算	16.5	24.3
2	減算	16.0	19.0
3	加減算 1	19.4	20.0
4	加減算 2	7.5	15.3
5	乗算	23.7	42.3
6	除算	8.5	11.0
7	乗除算 1	11.0	22.6
8	乗除算 2	20.0	28.3
9	左シフト	13.2	19.6
10	右シフト	12.9	18.3
11	シフト 1	4.2	7.6
12	シフト 2	1.3	2.0
13	論理和	15.5	22.6
14	論理積	15.6	21.6
15	メルセンヌ 1	27.2	50.0
16	メルセンヌ 2	10.8	11.0

メルセンヌ 2 が NTL と同等になったことでその効果がわかりますが、乗除算 2 が遅くなっているのを見てもわかるように必ずしも満足すべき結果とはいえません。実験していて気がついた問題点は、

1. 再帰関数による実装は簡単ではあるが実行効率が良くない。
2. 使用するスタック量が事前に予測できないため、大きな桁の乗算でスタック不足が発生する可能性がある。

ということです。現段階では Karatsuba 法を S0 に組み込むべきかどうかは判断できませんが、少なくとも現状の実験で使っている再帰関数による実装法は避けたいところです。

次は Knuth の本 [S2] の Toom-Cook 法を調べる予定ですが、その前にメルセンヌ素数の判定で使われている乗算は平方なので、平方専用の高速算法を組み込んでおこうと思います。NTL はおそらくこれを使っているのではないでしょうか。

6 再び古典算法 (2001.2.20)

古典乗法を元にして、平方算法を書いているあいだに古典乗法の最適化に関して新しいアイディアが湧いてきて、古典乗法を書き直しました。平方もそれをベースにして組み込みました。

表 6　SNBench Ver.0.01(3 回平均, 秒単位)

No.	項目	S0	NTL
1	加算	16.5	24.3
2	減算	16.0	19.0
3	加減算 1	19.3	20.0
4	加減算 2	7.0	15.3

5	乗算	32.0	42.3
6	除算	8.4	11.0
7	乗除算 1	11.0	22.6
8	乗除算 2	16.9	28.3
9	左シフト	13.2	19.6
10	右シフト	12.9	18.3
11	シフト 1	4.3	7.6
12	シフト 2	1.4	2.0
13	論理和	15.5	22.6
14	論理積	15.6	21.6
15	メルセンヌ 1	26.4	50.0
16	メルセンヌ 2	11.0	11.0

乗算試験で 15% 程度速くなりました。古典乗法で被乗数と乗数のポインタを比較して同じであれば古典平方算法を適用するようにして、メルセンヌ 1、2 はともに前回の Karatsuba 法並になってしまました。アセンブラによる最適化とはこんなものなのでしょう。ほんのちょっとしたアイディアで 1〜2 割の差がでてきてしまいます。次は Karatsuba 法による平方算法を組み込んで実験してみたいと思います。

7 再び Karatsuba 法 (2001.2.21)

Karatsuba 法に平方を組み込みました。

表 7 SNBench Ver.0.01(3 回平均, 秒単位)

No.	項目	S0	NTL
1	加算	16.6	24.3
2	減算	16.0	19.0
3	加減算 1	19.4	20.0
4	加減算 2	7.0	15.3
5	乗算	19.7	42.3
6	除算	8.4	11.0
7	乗除算 1	11.0	22.6
8	乗除算 2	17.5	28.3
9	左シフト	13.2	19.6
10	右シフト	13.0	18.3
11	シフト 1	4.2	7.6
12	シフト 2	1.2	2.0
13	論理和	15.6	22.6
14	論理積	16.7	21.6
15	メルセンヌ 1	23.7	50.0
16	メルセンヌ 2	7.9	11.0

次は Toom-Cook 法を調べる予定です。

8 Toom-Cook 法 (2001.3.12)

Toom の原理を確認するために作譜実験を行なって Toom-Cook 法を S0/math/uint32 に組み込むのは構造上難しいという結論に達しました。Cook の方法を使わなくても実装できるのですが、その場合、大量に消費するメモリを節約する算法を独自に考えなければなりません。理論上最適な算法でないものにそれだけの時間を使うのはもったいないと判断しました。

9 ガロア理論の復習 (2001.8.17)

仕事場を移すという大仕事のために研究をしばらく中断していましたが、やっと一段落して落ち着いてきたので再開することにします。新しい仕事場ではこれまであちこちに分散していた本や資料を一箇所にまとめて整理しました。これが最大の目的だったのですが、実際やってみると実に気持ちがいいものです。またコンピュータ環境を新しくし、それと同時にかねてからの懸案であった過去のファイル資産の分類・整理をやっと実行することができました。これでだいぶ仕事や研究がし易くなります。

多倍長整数計算の研究の今年のこれまでの成果を簡単に振り返ってみますと、まずプログラムの構造を変えて Pentium による最適化を行ない、乗算以外はある程度満足できる性能になりました。次に高速乗算法の実装研究に移り、最初の Karatsuba 法の実装はかなりうまくいきました。これによって中くらいの大きさの整数の乗算については目標としていた NTL を超えることができました。さらに大きい整数の高速乗算法として Toom-Cook 法について実験したのですが効率のいい実装は難しいのではないかという結論になりました。

次に何をやるべきかということなのですが、少し先を見渡したときに高速乗算法の実装がうまくいった段階で挑戦すべきテーマの一つとして、大きな整数の素因数分解があります。これに関する参考書はすでにいろいろ集めているのですが、この研究を行なうにはどうしても代数的整数論に通じる必要があります。代数学の基礎は学生時代にひととおり学んでいるので、感覚的に代数的整数論を理解し使いこなすにはかなり高度なスキルが要求されるということがわかっています。そこで、今年の夏は代数的整数論を学ぶための第一歩として、代数学の復習を行なうことにしました。実はこれは 7 月ぐらいからやっていて、今は目標であったガロア理論の復習が大体終わったところです。この夏の残りは高速乗算法の実装研究に使い、ある程度満足できる成果が得られたら、その次の段階として代数的整数論を集中的に学習する予定です。

10 高速乗算法 (1) の設計完了 (2001.9.1)

この夏はどういうわけか絶好調で、特に理論研究上の収穫が多かったのですが、その最大のものは高速乗算法とは次のような条件を満たす素数 p を見つけることであるという認識に至ったことかもしれません。

i. $\mathbf{F}_p$ の演算が実装対象となる CPU で高速に計算できる。
ii. 目的となる数の乗算がいくつかの準同型写像を経由することによって $\mathbf{F}_p$ の計算に帰着できる。

$p = 2^6 4 - 2^3 2 + 1$ の場合の高速乗算法の設計が何とか完了しました。現在コーディングの途中なのですが、完成するのにまた 1 ヶ月ぐらいかかりそうなので、この段階で文書にまとめたものを公開します。(→ [S5] 「高速乗算法の設計と実装 (1)」)

高速乗算法 (1) という番号を付けた理由ですが、今回いろいろ文献を調べているうちに、上の ( ) の条件は、

> (') 目的となる数の乗算がいくつかの準同型写像を経由することによって $\mathbf{F}_p$ の拡大体の計算に帰着できる。

というように拡張できそうだということがわかってきたからです。

２進計算機で計算するときの理想的な素数 p は、2^q-1 の形のメルセンヌ素数です。剰余演算が AND と加算だけで計算できるからです。残念なことに p をメルセンヌ素数としたとき、$\mathbf{F}_p$ 上では離散フーリエ変換の長さ n に制約があってそのままでは実用にはなりません。

長さ n の離散フーリエ変換が存在することと 1 の原始 n 乗根がその係数体に含まれることは同値です。複素数の場合は代数的閉体ですのでつねに存在します。

$p=2^q-1$ がメルセンヌ素数のとき、$\mathbf{F}_p$ の２次拡大体である $\mathbf{F}_p[i], i^2=-1$(ガウス整数) にはその長さ n の自由度が実用に十分なくらいあるということを、I. S. Reed と T. K. Truong が 1975 年に IEEE Trans-IT に書いた論文で示しています。(リード-ソロモン符号のリードです。まだご健在のようです。)

それ以後、このことに言及している文献はほとんど見つけることはできないのですが、1990 年代後半になって、J. Buhler, M. Shokrollahi, V. Stemann がこれをより一般的な代数体にするような研究を発表しています。彼らの一連の論文はかなり代数学を知らないと読めそうもないのでまだ手をつけていませんが、１年ぐらいかけて代数的整数論を勉強して挑戦してみようと考えています。

今回の高速乗算法 (1) の実装実験が終わったら、次の段階は高速乗算法　として $\mathbf{F}_p[i]$ の計算に帰着させるようなものを実際に設計してみたいと考えています。

11　高速乗算法 (1) の性能評価 (その１) (2001.9.9)

１年前に作ったショーンハーゲ・ストラッセン法のコードを再利用して、高速乗算法　をコーディングすることができました。剰余演算はアセンブラで書きましたが、FFT を含むキャリーフラグによる桁上げ以外の部分は C++ で書きました。デバッグに時間がかかるかなとも思ったのですが、数箇所修正する程度である程度複雑な乗算試験にパスしてしまいました。

古典算法や Karatsuba 法と同様に普通の乗算の他に平方専用の算法も組み込みました。古典算法や Karatsuba 法は Pentium 用に最適化しているので、現段階では勝負にならないのですが、大体の感じを探るために性能比較試験をやってみました。

表 8　]

乗算法の性能比較 [msec]

語長	CL 法	KO 法	KO/CL	M1 法	M1/CL
$2^4=16$	0.022	0.022	1.00	0.291	13.2
$2^5=32$	0.066	0.055	0.83	0.692	10.5
$2^6=64$	0.176	0.126	0.72	1.873	10.6
$2^7=128$	0.980	0.440	0.45	5.550	5.66
$2^8=256$	2.310	1.370	0.59	17.85	13.0
$2^9=512$	12.60	4.40	0.35	61.0	4.84
$2^{10}=1024$	34.10	14.80	0.44	223.0	6.54
$2^{11}=2048$	165.0	44.0	0.27	845.0	5.12
$2^{12}=4096$	539.0	148.0	0.27	3394.0	6.30

古典算法 (CL 法)、Karatsuba 法 (KO 法)、高速乗算法 (1)(M1 法) による平方をなるべく同じ条件で計測し、その時間と CL 法との比を並べました。語長が 2^{12} で終わっているのは、M1 法はかなり大量にメモリを消費するために事前に自由メモリ領域からメモリを確保しているのですが、これ以上の語長についてはこの段階でメモリ不足になってしまうからです。また、M1 法の実行時にどうもスワップが発生しているようです。

性能評価に使ったマシンの主な仕様は、

- CPU:Pentium-133MHz
- メモリ:48MB
- OS:Windows98 SE

ですが、このマシンでは M1 法の性能評価は無理かもしれません。少なくとも語長が 2^{20} 程度とれないと肝心な部分の性能が測定できません。

とりあえずメモリの節約と簡単にできる部分の最適化を検討して、実験をメモリ 64MB のマシンでやろうと思いますが、最終的にはメモリを 1GB 程度積んだマシンを用意しないと性能評価は難しいものと思われます。工房を建てたばかりで、すっからかんになっているので頭が痛いところです。

12 高速乗算法 (1) の性能評価 (その２) (2001.9.10)

余白に 0 を埋めるてすべての係数を計算するのをやめて語長でその係数が 0 かどうかを判断するように変更し、かなり効果がありました。

表 9]

乗算法の性能比較 [msec]

語長	CL 法	KO 法	KO/CL	M1 法	M1/CL
$2^4 = 16$	0.022	0.027	1.23	0.275	12.5
$2^5 = 32$	0.071	0.050	0.70	0.615	8.66
$2^6 = 64$	0.181	0.126	0.70	1.599	8.83
$2^7 = 128$	0.990	0.440	0.44	4.440	4.48
$2^8 = 256$	2.360	1.370	0.58	13.13	5.56
$2^9 = 512$	12.10	4.40	0.36	42.80	3.54
$2^{10} = 1024$	34.00	14.90	0.44	148.0	4.35
$2^{11} = 2048$	165.0	50.0	0.30	543.0	3.29
$2^{12} = 4096$	532.0	149.0	0.28	2092.0	3.93

剰余演算がちょうど浮動小数点演算器で誤差無く計算できる範囲であることに気が付きました。これで劇的に速くなる可能性がありますので、明日にでもやってみようと思います。

13 高速乗算法 (1) の浮動小数点演算による実装検討 (2001.9.11)

昨日の剰余演算がちょうど浮動小数点演算器で誤差無く計算できる範囲であることに気が付いたというのは錯覚でした。Pentium の浮動小数点演算器では整数は 2 の補数表現で扱っているので、その変換処理が必要になります。またこれを効率よく実装するためには、データを最初から浮動小数点形式でもつことが必要になり、最初の設計方針に反してしまいます。

高速化する上での問題点がいろいろ見えてきたので、高速乗算法　の実装と性能評価には一旦区切りをつけて、次の高速乗算法　の検討に入りたいと思います。

14 高速乗算法 (2) の設計完了 (2001.9.12)

高速乗算法 (2) は高速乗算法 (1) の欠点を改良したもので、本来は (1) という番号にすべきなのですが、ソースファイルや文書の管理上の　という番号付けにしました。メルセンヌ素数 $M_{31} = 2^{31} - 1$ について $\mathbf{Z}_{M_{31}}$ の 2 次拡大体である $\mathbf{Z}_{M_{31}}[i], i^2 = -1$ 上の離散フーリエ変換を高速乗算法に応用してみようという試みです。

(→ [S6] 「高速乗算法の設計と実装 (2)」)

15 高速乗算法 (2) の性能評価 (その１) (2001.9.17)

$\mathbf{Z}_{M_{31}}$ の２次拡大体 $\mathbf{Z}_{M_{31}}[i], i^2 = -1$、これはガウスの複素整数 $\mathbf{Z}[i]$ をイデアル (M_{31}) で剰余したものと同型になりますが、この２次体の演算ライブラリを整備して、原始根やその冪乗である 1 の原始 N 乗根を計算する実験にだいぶ時間がかかってしまいました。整数の剰余環と違って、やや複雑な構造をもっているのでコーディングや検算には注意が必要です。

２次体の計算を実際にコーディングしたのは長いプログラマ人生でも初めての体験でした。特に原始根の探索には時間がかかりました。有限体といってもかなりの大きさですし、なんといっても２次元ですので探索ループが２重になってしまいます。最初は２重ループでしらみつぶしに探索するプログラムを書いて、寝ている間に見つけようとマシン２台で１２時間程度検索したのですが見つかりませんでした。実行状況を調べるてみると一順目のループも終わってなかったので一時はあきらめかけました。

探索プログラムを見直しているうちにミスを発見して、それを修正し、いろいろな値で検算しているうちに偶然 $12 + i$ という原始根を発見しました。原始根が見つかれば、後は高速乗算法 (1) のソースを少し追加・修正するだけですので、２日ぐらいでコーディングとデバッグを終わらせることができました。次表は最初の性能試験の結果です。

表 10]

乗算法の性能比較 [msec]

語長	CL 法	KO 法	KO/CL	M2 法	M2/CL
$2^4 = 16$	0.022	0.028	1.27	0.368	16.7
$2^5 = 32$	0.071	0.055	0.77	0.780	10.9
$2^6 = 64$	0.176	0.126	0.72	1.786	10.1
$2^7 = 128$	0.980	0.390	0.40	4.060	4.14
$2^8 = 256$	2.360	1.320	0.56	9.060	3.84
$2^9 = 512$	12.70	4.400	0.35	19.70	1.55
$2^{10} = 1024$	34.10	15.40	0.45	44.50	1.30
$2^{11} = 2048$	159.0	50.00	0.31	99.00	0.62
$2^{12} = 4096$	516.0	148.0	0.29	220.0	0.43

非常にいいカーブを描いています。2048 語でついに古典算法を追い抜きました。

16 高速乗算法 (3) (2001.9.20)

高速乗算法 (3) は高速乗算法 (2) において係数を２つに分解しているのをやめて、係数を少し大きなメルセンヌ素数 $M_{89} = 2^{89} - 1$ で評価して、$\mathbf{Z}_{M_{89}}[i], i^2 = -1$ 上の離散フーリエ変換で１回で計算してみようという試みです。(→ [S7] 「高速乗算法の設計と実装 (3)」)

表 11]

乗算法の性能比較 [msec]

語長	CL 法	KO 法	KO/CL	M3 法	M3/CL
$2^4 = 16$	0.016	0.017	1.06	0.796	49.8

$2^5 = 32$	0.050	0.054	1.08	1.818	36.4
$2^6 = 64$	0.186	0.127	0.68	4.218	22.7
$2^7 = 128$	0.720	0.540	0.75	9.890	13.7
$2^8 = 256$	2.420	1.370	0.57	22.30	9.21
$2^9 = 512$	11.00	6.100	0.55	51.10	4.65
$2^{10} = 1024$	34.00	16.00	0.47	113.1	3.33
$2^{11} = 2048$	160.0	49.00	0.30	247.0	1.54
$2^{12} = 4096$	517.0	148.0	0.29	549.0	1.06

高速乗算法 (2) は 4 語と 2 語の剰余演算に分解していて、高速乗算法 (3) は 6 語の剰余演算を行なっているのですが、やはり細分したほうが速いという結果になっています。

17 高速乗算法 (4) (2001.9.21)

高速乗算法 (4) は係数を 3 語で評価するところは、 と同じですが、演算が 3 語でもいいところが大きな違いです。この詳細は論文にして投稿する予定ですのでしばらく非公開とします。メモリ消費量が減ったため 2^{13} 語まで演算することができるようになりました。

表 12]

乗算法の性能比較 [msec]

語長	CL 法	KO 法	KO/CL	M4 法	M4/CL
$2^4 = 16$	0.022	0.022	1.00	0.247	11.2
$2^5 = 32$	0.071	0.049	0.69	0.506	7.13
$2^6 = 64$	0.181	0.126	0.70	1.181	6.52
$2^7 = 128$	0.880	0.490	0.56	2.750	3.13
$2^8 = 256$	2.360	1.370	0.58	6.100	2.58
$2^9 = 512$	12.60	4.400	0.35	13.80	1.10
$2^{10} = 1024$	34.00	15.40	0.45	31.30	0.92
$2^{11} = 2048$	165.0	50.00	0.30	65.0	0.39
$2^{12} = 4096$	516.0	148.0	0.29	149.0	0.29
$2^{13} = 8192$	2114.0	500.0	0.24	335.0	0.16

18 高速乗算法 (5) (2001.9.25)

高速乗算法 (5) は係数を 2 語と 1 語に分解したものです。この詳細は論文にして投稿する予定ですのでしばらく非公開とします。

表 13]

乗算法の性能比較 [msec]

語長	CL 法	KO 法	KO/CL	M5 法	M5/CL
$2^4 = 16$	0.016	0.017	1.06	0.263	16.4
$2^5 = 32$	0.055	0.049	0.89	0.550	7.17
$2^6 = 64$	0.170	0.132	0.78	1.219	7.17

$2^7 = 128$	0.720	0.600	0.83	2.800	3.89
$2^8 = 256$	2.310	1.370	0.59	6.150	2.66
$2^9 = 512$	10.40	5.500	0.53	13.80	1.33
$2^{10} = 1024$	33.50	15.40	0.46	30.70	0.92
$2^{11} = 2048$	154.0	49.00	0.32	66.00	0.43
$2^{12} = 4096$	532.0	149.0	0.28	148.0	0.28
$2^{13} = 8192$	2054.0	494.0	0.24	324.0	0.16

参考文献

数論

[N1] Victor Shoup, *NTL: A Library for doing Number Theory*

算法

[S1] 野下 浩平, 高岡 忠雄, 町田 元, “基本的算法”, 岩波書店, 1983

[S2] D. E. Knuth(中川 圭介訳), “準数値算法/算術演算”, サイエンス社, 1986

[S3] 梅谷 武, *Karatsuba* 法

[S4] 梅谷 武, *Toom-Cook* 法

[S5] 梅谷 武, 高速乗算法の設計と実装 *(1)*

[S6] 梅谷 武, 高速乗算法の設計と実装 *(2)*

[S7] 梅谷 武, 高速乗算法の設計と実装 *(3)*