ハイパフォーマンスコンピューティング 60－6
（1996．3．4）

分散メモリ型並列計算機による高速多倍長計算

高橋大介† 金田康正††

東京大学大学院理学系研究科情報科学専攻†
東京大学大型計算機センター††

本稿では，分散メモリ型並列計算機により，多倍長桁数の計算を高速に行う方法について述べる．多倍長桁数の加減算においては，キャリーやボローの処理ををいかに並列化するかが鍵となるが，本稿で述べる工夫を行うことによって，この処理は並列化できる．

また，N桁の多倍長桁数の乗算はFFT(高速フーリエ変換)を用いれば$O(N \log N \log\log N)$のオーダーで求まるが，多倍長桁数の乗算の計算コンポーネントであるFFTの計算及び最終結果の正規化の部分を並列化することにより，高速に求めることができる．

32プロセッサのMIMD型並列計算機SR2001を用いて多倍長桁数の計算を行った結果，209万桁の加算において48.7倍，26万桁の乗算において49.8倍の加速率が得られた．

Fast Multiple-Precision Calculation on Distributed Memory Parallel Computers

Daisuke TAKAHASHI† Yasumasa KANADA††

Department of Information Science, Graduate School of Science, University of Tokyo†
Computer Centre, University of Tokyo††

This paper describes for the fast multiple-precision calculation on distributed memory parallel computers. In multiple-precision calculation, parallelization of carry and borrow propagation is the key component in the process speed. N digits multiple-precision multiplication can be realized the computing complexity of $O(N \log N \log\log N)$ with FFT(Fast Fourier Transform), we parallelized that FFT part and normalize part. Then we achieved the high speed multiple-precision multiplecation. According to the experimental results with MIMD parallel computer SR2001 of 32 PE, speed up ratio of 48.7 times at 2.09 million decimal digits addition, and 49.8 times speed-up ratio at 262 thousand decamal digits multiplication were attained.

1 はじめに

多倍長桁数を高速に計算するために，ベクトル計算機やRISCプロセッサにおいて，今までさまざまな試みがなされてきた[1][2].

本稿では，さらなる高速化を達成するために，分散メモリ型並列計算機により，多倍長桁数を高速に計算する方法について述べる．多倍長桁数の加減算においては，キャリーやボローの処理をいかに並列化するかが鍵となるが，本稿で述べるベクトル処理にも適応可能な工夫を行うことによって，この処理は並列化できる．

また，N桁の多倍長桁数の乗算はFFT(高速フーリエ変換)を用いれば$O(N \log N \log\log N)$のオーダーで求まるが，多倍長桁数の乗算の計算コンポーネントであるFFTの計算及び最終結果のFFTの計算及び最終結果の正規化の部分を並列化することにより，高速に求めることができる．

本稿では，分散メモリ型並列計算機により，多倍長桁数を高速に計算する方法について述べる．以下，2章で多倍長加減算および単精度定数乗算ルーチンの並列化について，3章で多倍長乗算の並列化について，4章で本稿で示す方法の評価について述べる．

2 多倍長加減算および単精度定数乗算ルーチンの並列化

多倍長桁数同士の加減算や多倍長桁数と単精度定数との乗算は，桁数をNとした場合，明らかに$O(N)$の計算量で行えることが分かる．

しかし，多倍長数同士の加減算や多倍長数と単精度定数との乗算において，並列化を阻害する要因は，キャリーおよびボローの処理である．例えば，足し算の場合を例にとって示すと次のようになる．

```
1  ICARRY=0
2  DO I=N,1,-1
3    ITEMP=MA(I)+MB(I)+ICARRY
4    IF (ITEMP .GE. 10000) THEN
5      MA(I)=ITEMP-10000
6      ICARRY=1
7    ELSE
8      MA(I)=ITEMP
9      ICARRY=0
10   END IF
11 END DO
```

ここでICARRYはキャリーを格納する変数で，ITEMPはテンポラリ変数である．また入力となるデータは各要素が10000以下に正規化されて配列MAとMBに入っているものとする．

このプログラムでは，IF文の中で定義されたICARRYが再帰的にITEMPの値を定める際に使用されているので，並列化は不可能である．

しかし，次に示すようなベクトル処理にも適応可能な工夫を行うことで，並列化が可能になる．HPF[3]で記述された，多倍長桁数の並列加算プログラムは，次のようになる．

```
1  MA(1:N)=MA(1:N)+MB(1:N)
2  DO WHILE (ANY(MA(2:N) .GE. 10000))
3    MC(N)=0.0D0
4    MC(1:N-1)=MA(2:N)/10000
5    MA(2:N)=MA(2:N)+MC(2:N)-MC(1:N-1)*10000
6    MA(1)=MA(1)+MC(1)
7  END DO
```

まず，1行目の部分で，キャリーを考慮せずに足し算を行ってしまう．次に，2行目のANY文で，配列MAの各要素に10000以上の値があるかどうかをチェックする．もしあれば，4行目で，キャリーを求めたのちに，5行目でキャリーの補正を行う．なお，MCはキャリーを格納する配列である．

ここで注意したいのは，このキャリーの補正においては，いわゆるキャリーの伝搬を考慮していないということである．したがって，キャリーは完全に補正されていない場合があるので，各要素が10000未満になるまでループが繰り返されることになる．このプログラムでは，$0.99999999\cdots9 + 0.00000000\cdots1$のようにキャリーの伝搬が頻繁に起こる場合は，並列化率が低くなり，性能が低下する恐れがある．

しかし，多倍長桁数の計算において，各要素の値はランダムで，正規化されているものと仮定すれば，キャリーが2回続けて出現する確率は，基数を10000とすれば，$\frac{1}{2} \times (\frac{1}{10000})^2 = 5 \times 10^{-9}$となり，
$0.99999999\cdots9 + 0.00000000\cdots1$などの特別な場合を除いては，事実上問題はないことが分かる．

キャリーの伝搬が頻繁に起こる場合は，桁上げ飛び越し(carry skip)方式[4]を用いるか，金田がベクトル計算機で行ったように[1]，桁上げ先見(carry look-ahead)方式を一次回帰演算で実現し，Recursive Doubling[5][6]によって

並列化することが考えられるが，4章での評価ではこれらの並列化を行わなかった．

ところで各プロセッサへのデータの割り当て方であるが，ブロック分割では，多倍長桁数の加算の計算量は桁数を N とすると，$O(N)$ であり，プロセッサ間通信はプロセッサ間でキャリーを転送するだけなので，プロセッサ数を P とすると，通信量は $O(P)$ となる．

また，サイクリック分割では，計算量は $O(N)$ でブロック分割の場合と変わらないが，プロセッサ間通信は各要素間で発生するので，通信量が $O(N)$ となってしまう．

したがって，プロセッサ間通信の量を減らすには，ブロック分割が良いが，ニュートン法によって，逆数や平方根を求めるような場合には，桁数を倍々にして求める必要があり，計算領域が次第に変化するので，ブロック分割ではプロセッサ利用率が悪くなる．

これは，通信量とプロセッサ利用率のトレードオフになるが，通信速度などの条件などで，状況は大きく変わるので，ブロック分割が良いか，サイクリック分割が良いかは一概には言えない．

今回は，通信量が増えても，プロセッサ利用率を高く維持出来る，サイクリック分割により，インプリメントを行い，実験を行った．

多倍長桁数同士の減算や多倍長桁数と単精度定数との乗算も，加算と同様にして実現が可能となる．この場合，計算量および通信量は多倍長数の加算とほぼ同じである．

3 多倍長桁数の乗算アルゴリズム

3.1 多倍長桁数の乗算

多倍長桁数の乗算のアルゴリズムは種々あるが [7]，筆算で行うように部分積を計算する方法（筆算では九九を用いた1桁×1桁，計算機では例えば4桁×4桁）では，計算桁数を N とすれば $O(N^2)$ の計算量が必要となり Z, 桁数が大きくなると膨大な計算量，計算時間が必要になる．高速のアルゴリズムとしては，積をとるべき数を上位と下位の桁に2分し再帰的なアルゴリズムを用いることによって，計算量を $O(N^{\log_2 3})$ に減らすアルゴリズム [7] や，$O(N \log N \log\log N)$ のシェーンハーゲストラッセンのアルゴリズム，そして有限体の上のフーリエ変換を使う手法 [8] がある．つまり，いくつかの素数についてモジュロ計算でフーリエ変換や畳み込み演算を行い，それらの最終結果から中国剰余定理 (Chinese Remainder Theorem) を用いて，本当の結果を構築する方法である*．また，複素フーリエ変換，すなわち浮動小数点演算を用いて畳み込み演算を行う方法もある．この場合，フーリエ変換プログラムとしては，計算センターに登録されている，高性能の FFT ルーチンを利用できるので，インプリメントが楽になるという特長がある．

数千桁以上の乗算では，FFT を用いた乗算アルゴリズムが実際的には最も速いことから，本論文では FFT による乗算アルゴリズムを使用する場合の議論を行う．

FFT による乗算ルーチンの具体的な実現方法は次のようになる．まず入力となる二つの $m \cdot 2^n$ ビット ($= m \cdot (\log_{10} 2) \cdot 2^n$ 桁の 10 進数) の数をそれぞれ A と B としたとき，この A と B をかけた結果 C を求める計算プログラムは次のようになる．

第一ステップ：$2 \cdot 2^n$ の長さの倍精度配列を二つ用意する．

第二ステップ：A, B それぞれの $m \cdot 2^n$ ビットの情報を先ほど用意した二つの配列の前半分に均等にばらまく．すなわち各配列要素にしまう情報は m ビット分となる．

第三ステップ：各配列の後ろ半分を 0.0D0 とする．

第四ステップ：2^{n+1} 点順 FFT 演算をそれぞれの配列に対して行う．その結果得られた配列を A', B' と呼ぶことにする．

第五ステップ：A' と B' に対して畳み込み演算を行う．得られた結果を配列 C' と呼ぶことにする．

第六ステップ：C' に対して今度は 2^{n+1} 点逆 FFT 演算を行う．その結果得られた配列を C と呼ぶことにする．

第七ステップ：配列 C の各要素は正確な答え，整数，から少しだけずれている浮動小数点となっているはずである．それで FORTRAN の

*D.H.Bailey の円周率 2936 万桁計算に，この方法に基づく多倍長乗算ルーチンが使用されている [9]

DNINT(X)=DINT(X+0.5D0) 組み込み関数的な演算を，配列 C の各要素に対して行う．(もしそのずれの最大値が 0.5D0 に近ければ，この乗算は正確に行われたと言い難いのでエラー)．

第八ステップ：適当な基底で配列 C を正規化する．2 進表現では 2^m を基底に，10 進表現では $10^{m(\log_{10} 2)}$ を基底にすると良い．

以上が具体的な実現方法である．

3.2 多倍長乗算ルーチンの並列化

多倍長乗算ルーチンを並列化するにあたって，3.1節で述べた乗算ルーチンの具体的な実現方法の第一ステップから第三ステップまでと，第七ステップは，明らかに並列化が可能であることが分かる．

また，第五ステップの畳み込み演算においては，フーリエ係数どうしの積をとることになるが，これはデータの個数を N とすれば，$O(N)$ の計算量で行え，しかも並列化は容易である．

問題は，FFT および正規化の部分の並列化であるが，FFT に関しては，並列アルゴリズムが多く提唱されており，実際にライブラリとして提供されているものも多い．したがって，高性能な並列 FFT ルーチンを利用するのが，正しいプログラムを簡単に書くという点からは理想的である．

また，逆 FFT の後の正規化の処理は，多倍長加減算および単精度定数乗算ルーチンの並列化におけるキャリーの処理と本質的には同一であるので，この部分も同様にして並列化できる．

ただし，キャリーの値が 1 とは限らないので，正規化時にキャリーが出現する確率は，加減算の場合より高くなる．

4 並列化の評価

並列化の評価に際しては，多倍長桁数 $(0.999\cdots)$ の加算および多倍長桁数 $(0.111\cdots)$ の乗算を，プロセッサ数 P と桁数 N を変化させて，実行時間を測定することにより行った．

計算機としては，MIMD 型並列計算機 SR-2001(32PE，総主記憶 2GB) を用いた．計算時間の測定に際しては，32PE すべてをシングルユーザーで使用し，経過時間を測定した．

並列 FFT ルーチンとしては，メーカー提供の 2 基底の実数 FFT を用いた．通信ライブラリも，メーカー提供のメッセージパッシング型の通信ライブラリを用いた．

プログラムは，FORTRAN で記述し，コンパイラは，日立の最適化 FORTRAN77 を用い，最適化オプションとして-W0,'OPT(O(S))' を指定した．

多倍長桁数のデータ構造としては，多倍長桁数を 10 進 8 桁ごとに区切り，下の位から順に 32 ビット整数の配列に格納することとする．したがって，乗算において FFT を行う際には，32 ビット整数の配列のデータを 64 ビット浮動小数点数の配列にコピーすることになる．

利用可能な主記憶容量の関係で計算桁数は，加算においては $(N = 2^{14} \sim 2^{21})$ 桁まで，乗算においては $(N = 2^{14} \sim 2^{18})$ 桁まで変化させ，プロセッサ数 P は，並列 FFT ルーチンが，2 のべきの台数のプロセッサでしか実行できなかったことから，$P = 1, 2, 4, 8, 16, 32$ と変化させて測定した．

ここで，乗算においては，FFT を行う作業領域が必要になるので，加算に比べて計算出来る桁数の上限が低くなっている．

結果は表 1，表 2 のようになった．

表 1. 多倍長桁数の加算の実行時間 (単位 秒)

N	P=1	P=2	P=4	P=8	P=16	P=32
2^{14}	0.004	0.002	0.002	0.001	0.001	0.001
2^{15}	0.010	0.004	0.003	0.002	0.001	0.001
2^{16}	0.022	0.011	0.005	0.003	0.002	0.002
2^{17}	0.042	0.021	0.011	0.005	0.003	0.002
2^{18}	0.089	0.044	0.023	0.012	0.006	0.003
2^{19}	0.212	0.088	0.046	0.023	0.012	0.006
2^{20}	0.592	0.214	0.093	0.046	0.024	0.012
2^{21}	1.208	0.585	0.218	0.094	0.048	0.025

表 2. 多倍長桁数の乗算の実行時間 (単位 秒)

N	P=1	P=2	P=4	P=8	P=16	P=32
2^{14}	0.219	0.154	0.064	0.056	0.052	0.078
2^{15}	0.528	0.326	0.127	0.097	0.069	0.088
2^{16}	2.322	0.821	0.288	0.195	0.104	0.121
2^{17}	5.956	2.247	0.613	0.386	0.169	0.167
2^{18}	13.094	6.222	1.619	0.901	0.313	0.263

多倍長桁数の加算，乗算のプロセッサ数 P に対する速度向上率 ($P = 1$ の場合の実行時間との比)$S(P)$ をそれぞれ図 1，2 に示す．これからも分かるように，並列化することにより，32 プロセッサで実行した場合，$2^{21}(\approx 209$ 万)

桁の加算で 48.7 倍，2^{18}(≈ 26 万) 桁の乗算で 49.8 倍の加速率が得られていることが分かる．

32 プロセッサで乗算を実行した場合，性能が頭打ちになっているように見えるが，これは今回使用した並列 FFT ルーチンが，2 の奇数乗のプロセッサ数の場合，性能が低下するためと考えられる．

実行するプロセッサ数の増加に従い各プロセッサの扱う問題のサイズが縮小されるため，キャッシュのヒット率が向上するが，一方でプロセッサ数の増加に伴う通信のオーバーヘッドも増加する．したがって，キャッシュのヒット率の向上と通信のオーバーヘッドの釣合が非常によくとれた結果がスーパーリニアスピードアップであるといえる．

しかし，加算においては，$N = 2^{17}$桁から，乗算においては，$N = 2^{16}$桁付近から，桁数が小さくなるにしたがって，並列化の効率が落ちている．これは，桁数が小さくなると，演算時間に対して通信の立ち上がり時間が無視できなくなってくるためと考えられる．

したがって，桁数 Nが小さい時は，並列化の効果はあまり期待できない．今回の結果では，加算，乗算共に 10 万桁を超えるあたりから並列化の効果が発揮できることが分かった．

また，32 プロセッサで計算した 2^{23}桁乗算の実行時間の内訳を表 3 に示す．これからも分かるように，乗算においては FFT の処理だけで 85%以上を占めている．

N桁の乗算においては，畳み込みや正規化の計算量は，$O(N)$ であるが，FFT の計算量は $O(N \log N \log\log N)$ であるので，Nが大きくなればなるほど，実行時間における FFT の比率が高くなる．

したがって，高速な並列 FFT ルーチンが使えるかどうかで乗算の速度がほとんど決まってしまうことが分かる．

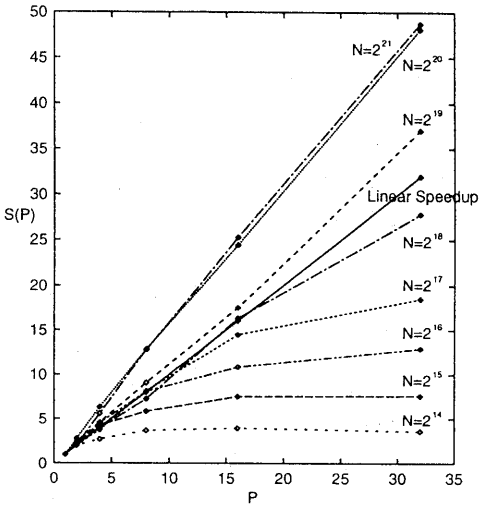

図 1. 多倍長桁数の加算の並列化効果

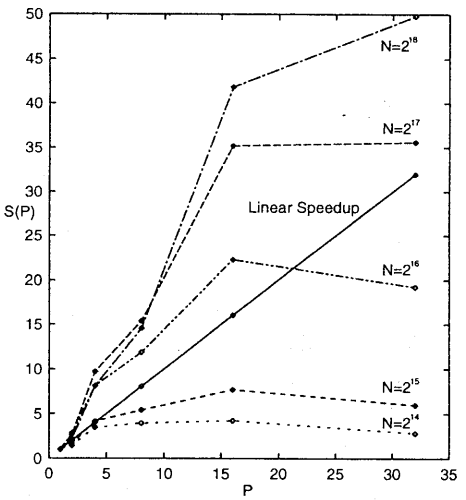

図 2. 多倍長桁数の乗算の並列化効果

表 3. 2^{23}桁乗算 (32PE) の実行時間の内訳

	実行時間 (秒)	比率
初期化	0.119	0.8%
順 FFT	7.316	54.6%
畳み込み	0.216	1.6%
逆 FFT	4.273	31.8%
整数化	0.298	2.2%
正規化	1.173	8.7%
合計	13.395	100.0%

5 まとめ

本稿では，分散メモリ型並列計算機により，多倍長桁数を高速に計算する方法について述べた．多倍長桁数の加減算においては，キャリーやボローの処理をいかに並列化するかが鍵になるが，本稿で述べた工夫を行うことによって，この処理を並列化できた．

また，N桁の多倍長桁数の乗算はFFT（高速フーリエ変換）を用いれば$O(N \log N \log\log N)$のオーダーで求まるが，多倍長桁数の乗算の計算コンポーネントであるFFTの計算及び最終結果の正規化の部分を並列化することにより，高速に求めることができた．

32プロセッサのMIMD型並列計算機SR2001を用いて，多倍長桁数の計算を行った結果，加算では209万桁において48.7倍，乗算では26万桁において49.8倍の加速率が得られた．

参考文献

[1] Y.Kanada: Vectorization of Multiple-Precision Arithmetic Program and 201,326,000 Decimal Digits of π Calculation, Supercomputing 88: Volume II, Science and Applications, Ed. by Joanne L. Martin and Stephen F. Lundstrom, IEEE Computer Society Press, pp. 117-128, 1989, IEEE Conference held at Kissimmee FL, Nov. 14-18 '88.

[2] 太田昌孝，前野年紀：多倍長計算のHPC技術，情報処理学会研究報告 94-HPC-54, pp. 61-65 (1994).

[3] High Performance Fortran Language Specification Version 1.0, High Performance Fortran Forum, (1993).

[4] M.Lehman and N.Burla: Skip Techniques for High-Speed Carry Propagation in Binary Arithmetic Units, IRE Trans. Elec. Comput., Col. Ec-10, pp. 691-698 (1961).

[5] S.Lakshmivarahan, Sudarshan K.Dhall: Analysis And Design of Parallel Algorithms; Arithmetic and Matrix Problems, McGraw-Hill Series in Supercomputing and Parallel Processing, pp. 337-379, (1990).

[6] H.Stone: An Efficient Parallel Algorithm for the Solution of Tradiagonal System of Equcations, JASM, Vol. 20, No. 1, pp. 27-38, (1973).

[7] D.E.Knuth: The Art of Computer Programming, Vol. 2 : Seminumerical Algorithms, Addison-Wesley, Reading, Massachusetts (1981).

[8] 後保範，長堀文子：ベクトル計算機による整数上のFFT計算，情報処理学会第27回全国大会，pp. 1293-1294 (1983).

[9] D.H.Bailey: The Computation of π to 29,360,000 Decimal Digits Using Borwein's Quartically Convergent Algorithm, Math. Comp., Col, 50, No. 181, pp.283-296 (1988)