Vol.2011-AL-135 No.1
2011/5/16

密グラフにおける全点対間最短経路アルゴリズムの高速化

柳澤 弘揮[†1]

通常の行列乗算におけるスカラー乗算と加算をそれぞれスカラー加算と最小値演算で置き換えた行列乗算は，全点対間最短経路問題などへの応用がある．最近，この演算に対し，最悪計算量は変わらないが実用的には通常の方法よりも高速な計算手法が提案された．本稿では，この手法をさらに改良し SIMD 命令を利用した高速化を可能にする．これを利用することにより，全点対間最短経路問題をフロイド・ワーシャル法よりも高速に解くことができることも示す．

A Faster All-Pairs Shortest Paths Algorithm for Dense Graphs

Hiroki Yanagisawa[†1]

Funny Matrix Multiplication (FMM) is a matrix multiplication operation in which the scalar addition and multiplication operations are replaced by the scalar minimization and addition operations, respectively. It is a fundamental computational task for matrices and its applications include the all-pairs shortest paths problem. Recently McAuley and Caetano have proposed a new algorithm whose expected computation time is significantly shorter than that of the straightforward FMM computation, while the worst-case time complexity remains unchanged. This paper gives an improved faster FMM algorithm, which exploits an instruction-level parallelism By using this new algorithm, the all-pairs shortest paths problem can be solved much more quickly than with the Floyd-Warshall algorithm.

†1 日本 IBM 株式会社 東京基礎研究所
IBM Research - Tokyo

1. はじめに

行列の乗算は，基本的な演算処理であり，これまで効率的な実装について数多く研究されてきた[11),17)]．これらの研究は，標準的な行列乗算に限らず，ブール行列上の乗算や Funny Matrix Multiplication (FMM) にも適用可能である．ブール行列乗算とは，通常の行列演算のスカラー加算と乗算がそれぞれブール値の論理和と論理積に置き換えられたもので，グラフの推移閉包の計算等に利用される．FMM は，スカラー加算と乗算がそれぞれ最小値演算と加算に置き換えられたもので，全点対間最短経路問題を解く際等に利用される．

本稿では，FMM の高速化とその全点対間最短経路問題への応用について考える．FMM の計算においては，長さ n の二つの配列 v_A と v_B が与えられたとき，下記の値を求めるのが主要な計算となる．

$$\min_{k=1,\ldots,n}\{v_A[k]+v_B[k]\}$$

通常，この計算には $O(n)$ 時間必要だが，McAuley と Caetano は，もし v_A と v_B 内の値をソートするような順列が事前に与えられていれば，(実用的には）もっと速く計算できることを示した[18)]．アイデアは単純で，k を $1,\ldots,n$ の順に調べていくのではなく，$v_A[k]$ や $v_B[k]$ の値が小さくなるような k から順に調べていくことにより，$v_A[k]+v_B[k]$ の値が大きくなるような k を調べる処理をスキップするというものである．このアイデアにより，最悪計算時間はこれまでと変わらないものの計算時間の期待値は短くなることが示された．FMM の入力として与えられる二つの行列の行と列をそれぞれソートする時間は，合計で $O(n^2 \log n)$ で十分なため，FMM 全体の計算が高速化できる．

本稿では，このアルゴリズムを，SIMD (Single Instruction Multiple Data) 命令を利用することによる高速化が可能になるように，改良した．McAuley と Caetano のアルゴリズムは，多くのランダムなメモリ・アクセスが伴うが，改良したアルゴリズムでは，多くのメモリ・アクセスが連続になるように工夫した．このことによって，SIMD 命令の活用が可能になり，４並列の SIMD 命令で３倍以上の高速化を達成した．

次に，この SIMD 命令で高速化したアルゴリズムを，密グラフにおける全点対間最短経路問題に適用した．全点対間最短経路問題は，与えられた重み付き有向グラフ $G=(V,E)$ について，全ての頂点間の最短経路長を計算する問題である．与えられたグラフが密のとき，二つの手法がよく知られている．一つは，フロイド・ワーシャル法[10),24)] である．このアルゴリズムは，最悪計算時間が $O(n^3)$ で，実用的には最も高速だと考えられており多くの

© 2011 Information Processing Society of Japan

実装が提案されている[3),12),13),15),19),22),23)]．もう一つは，分割統治法によるもので，FMM をサブルーチンとして利用する $O(n^3)$ 時間アルゴリズム[1),4),9),21)] である．本稿では，改良した FMM を後者の手法と組み合わせ，フロイド・ワーシャル法より高速に解けることを実験的に示す．

2. 関連研究

行列乗算は，全点対間最短経路問題と深く関係している．重みなしの有向グラフに対しては，Alon ら[2)] が，ブール行列乗算を利用して，$o(n^3)$ 時間アルゴリズムを提案している．重み付き無向グラフに対しては，通常の行列演算を利用した $o(n^3)$ 時間アルゴリズムが幾つか提案されている[20)]．重み付き有向グラフについては，この問題と FMM が時間計算量の観点で同等であることが知られている[1)]．

二つの $n \times n$ 行列に対しての，(標準的な) 行列乗算やブール行列乗算については $O(n^{3-\epsilon})$ 時間（ϵ は正の定数）アルゴリズムが知られている[7)] が，FMM を行う $O(n^{3-\epsilon})$ 時間アルゴリズムは知られていない．FMM の最悪時間計算量について理論的な観点からは非常に多くの研究がなされているが，現在の最良のアルゴリズム[5)] でも $O(n^3 \log^3 \log n / \log^2 n)$ である．その反面，FMM を実用的に高速に計算する方法については，これまであまり研究されてこなかった．我々の知る限りでは，McAuley と Caetano が実用的に高速な非自明なアルゴリズムを初めて提案した[18)]．このアルゴリズムの最悪時間計算量は $O(n^3)$ であるが，実用的にはそれよりも速く計算できる．我々のアルゴリズムは，このアルゴリズムをより簡単かつ高速にしたものであり，さらには，SIMD 命令を利用した高速化も行った．

全点対間最短経路問題に対しては，フロイド・ワーシャル法[10),24)] の高速な実装に関する研究が数多くなされてきた．Venkataraman ら[22)] は，ブロック化と呼ばれるテクニックを利用してキャッシュ効率を高めたアルゴリズムを提案し，Penner と Prasanna はそれをさらに高速化した[19)]．現在，通常の CPU 向けで最速の実装は，Han ら[12)] が提案した手法で，SIMD 命令の活用による高速化も含まれている．Bondhugula ら[3)] は，FPGA (Field Programmable Gate Array) 向けに高速な実装を提案し，Vinjamuri と Prasanna[23)] は，Cell Broadband Engine 向けに最適化した実装を提案した．Harish と Narayanan[13)] は GPU を利用して CUDA での実装を提案し，Katz と Kider[15)] はブロック化によりこの CUDA での実装をさらに高速化した．

フロイド・ワーシャル法以外では，分割統治法を利用したアルゴリズムも幾つか提案されている．古くは 1970 年代に出版された本にも現れた手法[1)] があり，最近では Tiskin によるブロック化したガウスの消去法に基づくアルゴリズム[21)]，D'Alberto と Nicolau による R-Kleene アルゴリズム[9)]， Buluç らが提案したアルゴリズム[4)] がある．

McAuley と Caetano[18)] もまた，彼らの FMM を利用した全点対間最短経路問題に対するアルゴリズムを示している．しかし，彼らは繰り返し二乗法（参考: 節 5）を全点対間最短経路問題を解くのに利用したため，最悪時間計算量が $O(n^3 \log n)$ になってしまっている．

3. 記　　法

特に断りが無い限り，本稿では n 行 n 列の正方行列のみを考え，簡単のため n は 2 のべき乗であるとする．行列 A の i 行 j 列目の要素は $A[i,j]$ と書く．K 並列の SIMD 加算命令とは，二つの長さ K の配列 v_A と v_B について，$v_C[i] = v_A[i] + v_B[i]$ を計算する一つの命令を意味する．通常の加算命令しか使えないと少なくとも K 個の命令が必要となり，一個の命令で同じ計算がことによる高速化の効果は大きい．単に K 並列の SIMD 命令と言った場合，乗算命令などの他の命令についても同様に定義した命令も含む集合全体を指すものとする．

4. Funny Matrix Multiplication

行列 A と B について，FMM (Funny Matrix Multiplication) $C = A \cdot B$ は下記の計算を全ての $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,n\}$ について行うものとして定義される．

$$C[i,j] = \min_{k=1,\ldots,n} \{A[i,k] + B[k,j]\}$$

素直に実装すると，この計算には $O(n^3)$ 時間を要する．

4.1 我々のアルゴリズム

長さ n の二つの配列 v_A と v_B について $\min_k\{v_A[k] + v_B[k]\}$ の計算をするとき，v_A と v_B の値をソートする順列が与えられれば，もっと速く計算できる．順列 $a_1, a_2, \ldots, a_n$ を v_A の値が昇順になるインデックスの順列とし（つまり，$v_A[a_1] \le v_A[a_2] \le \cdots \le v_A[a_n]$），順列 $b_1, b_2, \ldots, b_n$ を v_B の値が昇順になるインデックスの順列とする．このとき，図 1 に示されるアルゴリズム 1 によって，$\min_k\{v_A[k] + v_B[k]\}$ が計算できる．

このアルゴリズムが正しく動作することは容易に確かめられる．インデックス k^* を，$v_A[k] + v_B[k]$ の値が最小になる k とし，t^* を $v_A[k^*] + v_B[k^*]$ とおく．このとき，$v_A[k^*] \le t^*/2$ もしくは $v_B[k^*] \le t^*/2$ の少なくとも一方が成立する．アルゴリズム 1 では，変数 t が t^* の上限値を保持するため，k^* を見つけるためには，$v_A[a_i] \le t/2$ と

© 2011 Information Processing Society of Japan

Vol.2011-AL-135 No.1
2011/5/16

入力: 長さ n の二つの配列 v_A，v_B と，それらをソートする順列 $\{a_i\}$ と $\{b_j\}$
出力: $\min_k\{v_A[k]+v_B[k]\}$ を計算した値

1: $t=\infty$, $i=1$, and $j=1$
2: **repeat** // 一つ目のループ
3: $t=\min\{t, v_A[a_i]+v_B[a_i]\}$
4: $i=i+1$
5: **until** $i>n$ or $v_A[a_i]>t/2$
6: **repeat** // 二つ目のループ
7: $t=\min\{t, v_A[b_j]+v_B[b_j]\}$
8: $j=j+1$
9: **until** $j>n$ or $v_B[b_j]>t/2$
10: **return** t

図 1 アルゴリズム 1

$v_B[b_j]\le t/2$ を満たすインデックス a_i と b_j について調べれば十分である．よって，下記の補題 4.1 が成立する．

補題 4.1 アルゴリズム 1 は $\min_k\{v_A[k]+v_B[k]\}$ の値を正しく計算する．

アルゴリズム 1 の最悪計算時間は明らかに $O(n)$ であるが，計算時間の期待値はそれよりもずっと短い．たとえば，配列 v_A と v_B の各値が 0 以上 1 以下の互いに独立な乱数から選ばれた場合，McAuley と Caetano のアルゴリズムに対する証明[18] と同様にして，最悪計算時間が $O(\sqrt{n})$ であることを示すことができる．

McAuley と Caetano のアルゴリズムとの比較 アルゴリズム 1 は，McAuley と Caetano のアルゴリズム[18] を簡略化したものである．違いの一つは，アルゴリズム 1 ではインデックスを $a_1,\ldots,a_n,b_1,\ldots,b_n$ の順序で調べているが，彼らのアルゴリズムではインデックスを $a_1,b_1,a_2,b_2,\ldots,a_n,b_n$ の順序で調べている点である．他には，ループの終了条件も違っている．彼らのアルゴリズムでは，たとえば $v_A[a_i]+v_B[a_i]$ と $v_A[b_i]+v_B[b_i]$ の値を調べた後，$a'=b_i$ と $b'=a_i$ となるような a' と b' を見つけ，それをループの終了条件に利用している．そのような a' と b' を効率よく見つけるためには，$i=a^{-1}[a_i]$ と $j=b^{-1}[b_j]$ となるようなテーブル a^{-1} と b^{-1} を保持する必要があり余分な記憶容量が必要となってしまう．このような違いが，彼らのアルゴリズムの SIMD 命令での実装を難しくしている．次節では，SIMD 命令を利用してアルゴリズム 1 をどのように高速化するかについて述べる．

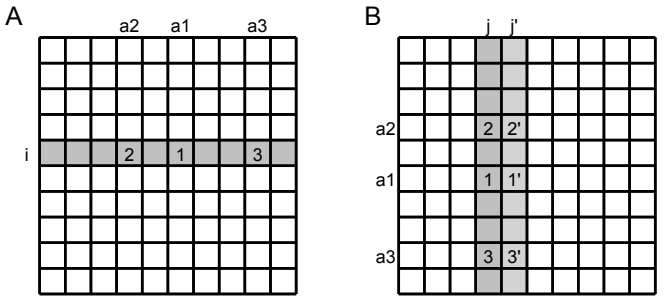

図 2 $C[i,j]$ と $C[i,j']$ を計算する際の，アルゴリズム 1 の一つ目のループのアクセス・パターン

4.2 SIMD 命令を利用した高速化

アルゴリズム 1 を直接 SIMD 命令を活用して高速化するのは容易ではない．しかし，少しアルゴリズムを調整することで SIMD 命令による高速化が可能となることを示す．

それにあたってまずは，$j'=j+1$ とおき，アルゴリズム 1 を利用した FMM 全体の計算で $C[i,j]$ と $C[i,j']$ が，どのような手順で計算されるかを詳しく見ていく．まず $C[i,j]$ を計算する際，アルゴリズム 1 の一つ目のループでは，行列 A の i 行目 $(A[i,a_1], A[i,a_2], \ldots)$ と行列 B の j 列目 $(B[a_1,j], B[a_2,j], \ldots)$ にアクセスする．次に，$C[i,j']$ を計算する際，アルゴリズム 1 の一つ目のループでは，行列 A の i 行目と行列 B の j' 列目 $(B[a_1,j'], B[a_2,j'], \ldots)$ にアクセスする．よって，アクセスパターンは，図 2 に示されるものとなり，以下の特徴が分かる．

- 行列 A の i 行目をアクセスするという点は同じである．
- 行列 B の j 列目と j' 列目へのアクセスは，隣り合っている（たとえば，$B[a_1,j']$ と $B[a_1,j]$，$B[a_2,j']$ と $B[a_2,j]$ のように）．

これらの特徴から，行列 B を列優先でメモリに配置することにより，$C[i,j]$ と $C[i,j']$ を計算する際のアルゴリズム 1 の一つ目のループを統合し，2 並列の SIMD 命令を利用して高速化するのは難しくない．具体的には，$B[a_i,j]$ と $B[a_i,j']$ へのアクセスを一つの SIMD 読込命令で行い，$A[i,a_i]+B[a_i,j]$ と $A[i,a_i]+B[a_i,j']$ の計算を一つの SIMD 加算命令で行う．なお，ループ統合を行う際，ループの終了条件を適切に設定する必要があることに注意されたい．$C[i,j]$ を計算する際は 変数 t について $v_A[a_i]>t/2$ が終了条件であったが，$C[i,j']$ を計算する際は別の変数 t' について $v_A[a_i]>t'/2$ が終了条件となるか

© 2011 Information Processing Society of Japan

らである．よって，正しい解を得るためには終了条件を $v_A[a_i] > \max\{t/2, t'/2\}$ に設定する必要がある．この終了条件の変更に伴いループ回数が増えてしまうが，実際には，これによる速度低下よりも SIMD 命令の活用による高速化の寄与のほうが大きい．

ループの統合は二つのループに限定されず，K 個のループを統合し K 並列の SIMD 命令を活用するのも容易である．K の値を大きくしすぎるとループ回数の増加による速度低下が起こるが，それでも最悪計算量は $O(n^3)$ で変わらない点にも注意されたい．

これまでの議論により，アルゴリズム 1 の一つ目のループを統合することには成功したが，全く同じ方法では，二つ目のループの統合はできない．$C[i,j]$ と $C[i,j']$ の計算を統合しようとすると，$B[b_j,j]$ と $B[b_{j'},j']$ へのアクセスがメモリ上の隣同士に配置されているとは限らないからである．

しかし，行列 C の（列方向ではなく）行方向に連続した要素の計算に着目すると，行列 A を列優先でメモリに配置することで，二つ目のループを統合することができる．つまり，$i = i' + 1$ としたとき，$C[i,j]$ と $C[i',j]$ をアルゴリズム 1 で計算すると，二つ目のループでは，行列 A の i 行目 $(A[i,b_1], A[i,b_2], \ldots)$ と i' 行目 $(A[i',b_1], A[i',b_2], \ldots)$ と行列 B の j 列目 $(B[a_1,j], B[a_2,j], \ldots)$ にアクセスする．よって，アルゴリズム 1 を二つのループに分解し，一つ目のループでは行列 C について行方向に計算をしていき，二つ目のループでは行列 C について列方向に計算していけばよい．

実装の詳細　我々の実装では，4 並列の SIMD 命令を活用するため，まずは四つのループを統合した上で，さらにレイテンシの隠蔽による高速化のため，統合されたループをさらに二つ統合した．また，順列 $a_1, \ldots, a_n$ と $b_1, \ldots, b_n$ が事前にわかっているため，キャッシュ・プリフェッチ命令も利用した高速化も行った．

それから，SIMD 命令を利用した実装では，一つ目のループを処理する際には行列 C を行優先で記憶する必要があり，二つ目のループを処理する際には行列 C を列優先で記憶する必要があるため，途中で行列の転置操作も必要となることに注意されたい．この転置操作には，6) で提案されているアルゴリズム 3 を利用した．実験においては，この転置操作に要する時間を計測したが，FMM 全体の計算時間に占める割合は非常に小さかった．

5. 全点対間最短経路問題

ここでは，FMM を利用して全点対間最短経路問題を解く方法について述べる．与えられたグラフ $G = (V, E)$ の隣接行列を W とおく．つまり，$W[i,j]$ には頂点 i から頂点 j への距離が格納されているものとする．頂点 i から頂点 j に到達する経路のうちで最大で ℓ 本の枝を通る最短経路の長さを $L_\ell[i,j]$ に設定した行列を L_ℓ とおく．このとき，頂点 i から頂点 j への最短経路のうち丁度 $\ell+1$ 本の枝を通るものは，頂点 i から頂点 k までの ℓ 本の枝を通る最短経路と頂点 k から頂点 j に向かう辺に分解できるため，$L_{\ell+1} = L_\ell \cdot W$ が成り立つことが容易に確かめられる．定義より $L_1 = W$ なので，全点対間最短経路問題は $L_n = W^n$ を計算することに対応する．$n-1$ 回の FMM を適用することで W^n を計算することは可能だが，1 回の FMM に $O(n^3)$ 時間かかるとすると全体で $O(n^4)$ 時間かかってしまう．$W^{2k} = W^k \cdot W^k$ の関係を利用する繰り返し二乗法[8] と呼ばれる方法により，$O(n^3 \log n)$ 時間にまで削減することはできる．

分割統治法を使う[1),4),9),21)] と，さらにこれを $O(n^3)$ 時間まで削減できる．本稿では，図 3 に示される Buluç らによるアルゴリズム 2[4] を使用する．このアルゴリズムでは，入力として与えられる隣接行列 W を以下のように $n/2$ 行 $n/2$ 列の部分行列 A, B, C, D に分解して利用する:

$$W = \begin{pmatrix} A & B \\ C & D \end{pmatrix},$$

出力は，W に上書きして出力される．アルゴリズム 2 の 4 行目と 8 行目では，行列の要素毎の最小値演算を利用する．1 行目と 5 行目では，アルゴリズム 2 自身に対する再帰呼び出しがある．再帰呼び出しは途中で別の全点対間最短経路アルゴリズムに置き換えても問題ないので，我々の実装では，再帰呼び出し行列のサイズが 256×256 以下になったときにはフロイド・ワーシャル法を利用した．

アルゴリズム 2 の計算時間が $O(n^3)$ であることを示す．$T(n)$ を n 頂点のグラフに対する計算時間，$M(n)$ を n 行 n 列の行列同士の FMM 演算にかかる時間とする．アルゴリズム 2 は，大きさ $n/2$ のグラフに対する二つの再帰呼び出しと，$n/2$ 行 $n/2$ 列の行列同士の六つの FMM 演算から構成されているため，$T(n) \leq 2T(n/2) + 6M(n/2) + O(n^2)$ の関係が成り立つ．$T(1) = O(1)$, $M(1) = O(1)$, $M(n) = O(n^3)$ なので，$T(n) = O(n^3)$ となる．また，正の定数 $\epsilon > 0$ について，$M(n)$ が $O(n^{3-\epsilon})$ に改善されると，$T(n) = O(n^{3-\epsilon})$ となることにも注意されたい．

6. 実　　験

ここでは，我々の提案するアルゴリズムについての実験結果を示す．全ての実験は，Intel Quad Core Xeon (E5570) 2.93 GHz, 24 GB RAM, 2×8 MB L3 キャッシュを搭載した

© 2011 Information Processing Society of Japan

入力: 与えられたグラフ $G = (V, E)$ の隣接行列 W
出力: 最短経路長を格納した行列 W^n

1: $A = APSP(A)$
2: $B = A \cdot B$
3: $C = C \cdot A$
4: $D = \min\{D, C \cdot B\}$
5: $D = APSP(D)$
6: $C = D \cdot C$
7: $B = B \cdot D$
8: $A = \min\{A, B \cdot C\}$

図 3 アルゴリズム 2

マシンで行われ，OS には Red Hat Enterprise Linux 5.4 を利用した．全てのプログラムは，C++ で記述し，`g++` に `-O2 -march=x86-64` オプションを指定してコンパイルした．行列の各要素の値は単精度浮動小数点を用い，SIMD 命令は Intel の SSE イントリンシックで記述した．実行時間の単位は全て秒で，入力を読み込む時間は含まれていない．

6.1 Funny Matrix Multiplication

スカラー実装 まず，アルゴリズム 1 (FMM-SEQ) の効果を示すため，通常の FMM 計算 (straightforward) や McAuley と Caetano のアルゴリズム (FMM-MC) との比較実験を行った．ここでは，行列 A と C は行優先で配置し，行列 B は列優先で配置したうえで，$C = A \cdot B$ の計算を行った．

まず一つ目の実験として，行列の各値を 0 以上 1 以下の独立な乱数で設定し，FMM の計算に要する時間を比較した．行列 A, B の n の値をさまざまに変化させたとき，実行時間は図 4 に示すように変化した．FMM-MC に関してはすでに McAuley と Caetano が示している[18] ように，通常の FMM 計算よりも高速に動作することが分かる．(オリジナルの実装[18] では Python が使われたが，我々は C++ で実装したため，我々の実装のほうが高速であることに注意されたい．) そして，アルゴリズム 1 (FMM-SEQ) は，FMM-MC よりもさらに速く動作することもわかる．これは，$v_A[k] + v_B[k]$ を一回計算することを「和計算」と呼ぶことにしたとき，FMM-SEQ は FMM-MC と比較すると，その和計算の数が少ないことによるものである（図 5 を参照）．また，通常の FMM では，和計算の数はちょうど n になるのに対し，FMM-SEQ では和計算の数は n よりも小さい速度で上昇するこ

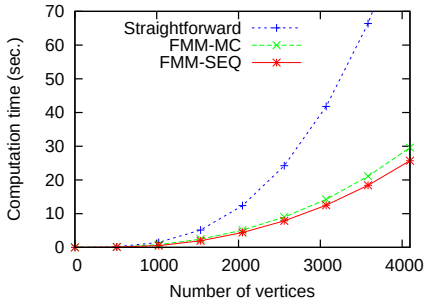

図 4 ランダムな行列に対する，通常の FMM, FMM-MC, FMM-SEQ の実行時間比較

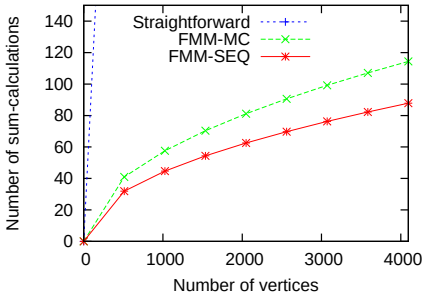

図 5 ランダムな行列に対する，通常の FMM, FMM-MC, FMM-SEQ の和計算の数の比較

とにも注意されたい（この実験では，$\sqrt{n}$ と $2\sqrt{n}$ の間くらい）．

FMM-SEQ も FMM-MC も共に，行列 A と B が互いに相関のある場合は高速に動作し，そうでない場合 (反相関な場合）には動作が遅くなる．これを確かめるために，非常に反相関な行列 A と B を以下のようにつくる

$$A[i, j] = (i - 1) + e_{i,j} \text{ and } B[i, j] = (n - j) + e'_{i,j}$$

ここで，$e_{i,j}$ と $e'_{i,j}$ は互いに独立な正規分布 $N(0, n^2/100)$ から選んだ乱数である．こうしてつくった都合の悪い行列に対しては，図 6 に示すように，FMM-SEQ と FMM-MC は，和計算の数が増加してしまい（参考: 図 7），動作が遅くなる．ただし，ここで作成した都合の悪い行列は，非常に極端なものであり，通常はこうした行列が入力として与えられる可能性は低い，と考えられる．また，たとえこうした行列が入力として与えられたとしても，最悪実行時間は $O(n^3)$ で変わらず，定数時間の速度低下にとどまることに注意されたい．

SIMD 実装 さらに，アルゴリズム 1 のスカラー実装 (FMM-SEQ) と，それを SIMD 命令で実装したもの (FMM-SIMD) について，さきほどと同様に 0 以上 1 以下の乱数で埋めた行列を入力として比較実験を行った．その結果，**表 1** に示すように，FMM-SIMD は FMM-SEQ に比べて和計算の数は 1.8 倍くらいに増えてしまうが，それでも実行時間は 3.4 〜 6.0 倍速くなった．つまり，4 並列の SIMD 命令を利用して，4 倍以上の高速化が達成できている．その理由としては，メモリ・アクセスのパターンが連続的なものに変わったことによりキャッシュ・ヒット率が向上した可能性や，キャッシュ・プリフェッチ命令の効果が考えられる．実際に，キャッシュ・プリフェッチ命令を除去すると，除去する前の FMM-SIMD に比べて 1.3 〜 1.9 倍遅くなった．他に考えられる理由としては，SIMD 命

© 2011 Information Processing Society of Japan

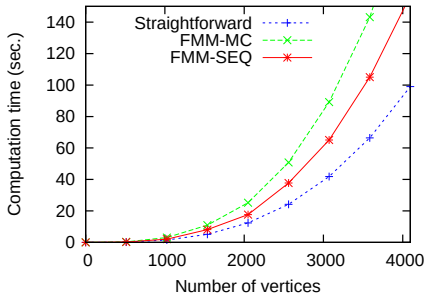

図 6 FMM-SEQ にとって都合の悪い入力行列に対する，通常の FMM, FMM-MC, FMM-SEQ の実行時間比較

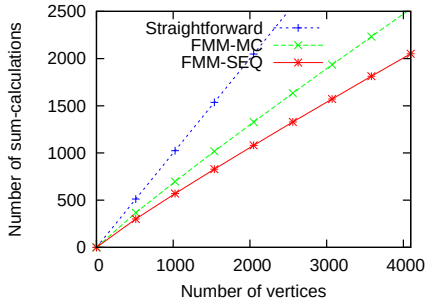

図 7 FMM-SEQ にとって都合の悪い入力行列に対する，通常の FMM, FMM-MC, FMM-SEQ の和計算の数の比較

	FMM-SEQ		FMM-SIMD			速度向上
n	実行時間	和計算の数	実行時間	実行時間（プリフェッチ付き）	和計算の数	（プリフェッチ & SIMD）
1024	0.509	44.654	0.192	0.151	79.109	3.4
2048	4.421	62.525	1.967	1.124	111.615	3.9
4096	25.713	87.840	12.208	6.281	157.680	4.1
8192	153.931	123.577	45.966	25.506	222.558	6.0

表 1 FMM-SEQ と FMM-SIMD の比較

令の活用により条件分岐命令が削減されたために，分岐ミスによるペナルティが少なくなった可能性が考えられる．

FMM-SIMD では，ソートに要する時間は無視できないため，高速なソート法が必要である．FMM-SIMD では，(詳細は述べないが) コムソート[16] を SIMD 命令の活用により 4 並列で行うアルゴリズムを実装し利用した．これは，井上らが提案した AA-sort[14] を簡略化したものである．まず，このソート法の実装が高速であることを示すため，標準ライブラリに入っている `qsort` と比較した．入力としては長さ n のランダムな配列を n 個用意し，それら全てをソートする時間を比較した．**表 2** に示す結果から，我々のコムソート実装は十分に高速であることが確かめられる．次に，FMM-SIMD による一回の FMM 計算に占めるソート時間の割合を計測した．入力にランダムな行列を与えたとき，結果は **表 3** に示す結果となり，計算時間のうちソート時間が占める割合は 21% 以下であることがわかる．よって，ソートに占める時間は大きくは無く，たとえより高速なソートの実装を利用したとしても，FMM 計算の時間は最大でも 21% しか削減できないため，大きな問題ではない．また，表 3 に示すように，FMM-SIMD において行列の転置にかかる時間は無視できるほど小さいことも分かる．

n	`qsort` の実行時間	我々のコムソートの実行時間	速度向上
2048	0.48	0.11	4.3
4096	2.07	0.45	4.7
8192	9.92	2.17	4.1
16 384	38.39	10.15	3.8

表 2 我々のコムソートの実装と qsort の実行時間比較

n	ソートに要する時間割合 (%)	行列転置に要する時間割合 (%)
2048	20.2	2.7
4096	14.6	1.4
8192	17.5	1.3

表 3 FMM-SIMD の実行においてソートと行列の転置に要する時間が占める割合

6.2 全点対間最短経路問題

ここでは，我々の FMM アルゴリズムの効果を示すため，全点対間最短経路問題に対して応用した結果を示す．FMM-SIMD と，繰り返し二乗法とを組み合わせたもの (APSP-SQUARE) と分割統治法であるアルゴリズム 2 とを組み合わせたもの (APSP-DC) の二つを実装した．

これらの実装と，フロイド・ワーシャル法を比較する．フロイド・ワーシャル法は，密グラフに対して最も高速であると考えられており，Han ら[12] による最速の実装が著者のウェブサイトに公開されている．ただ，この実装をダウンロードして手元のマシンに合わせてチューニングをした上で，(最大 4096 頂点の) サンプル入力に対する実行時間を計測することには成功したものの，これに対して任意の入力を与えて実行する方法を見つけることができなかった．そのため，比較相手としては，22) で提案されているより簡易なフロイド・ワーシャル法を，我々が実装したもの (APSP-FW) を利用した．これはブロック化技術を用いてキャッシュ効率を高めたものであり，ブロックサイズとして 64×64 を用い，SIMD 命令も活用した．この実装は，我々のマシンでは最大 6332 FLOPS (FLaoting point

© 2011 Information Processing Society of Japan

頂点数	APSP-FW	APSP-SQUARE	APSP-DC
1024	0.341	1.946	0.403
2048	2.727	15.081	1.810
4096	21.707	84.470	11.669
8192	173.234	410.675	66.890
16 384	1385.72	3706.31	369.667

表 4 グラフ RAND-G に対する全点対間最短経路アルゴリズムの実行時間比較

頂点数	APSP-FW	APSP-SQUARE	APSP-DC
1024	0.340	3.083	0.668
2048	2.726	53.380	5.165
4096	21.725	727.846	78.054
8192	173.212	7544.17	846.751
16 384	1385.59	-	7789.78

表 5 グラフ HYPERCUBE-G に対する全点対間最短経路アルゴリズムの実行時間比較.

number Operations Per Second) の性能があり，Han らの最速の CPU 実装[12] が最大で 9753 FLOPS であったのと比較すると，まずまずの性能であると言える．実験に使用したマシンは，1 クロックあたり四つの浮動小数点演算が可能で，最大周波数が 2.93 GHz であるから，ピーク時の性能は 11720 FLOPS となる．つまり，我々の APSP-FW は，マシンのピーク性能の 54% を達成している．

これらの実装の実行時間を比較するため，まず，ランダム・グラフ (RAND-G) を生成した．このグラフは完全グラフで，各辺の重みは互いに独立に $\{1, 2, \ldots, 1000\}$ からランダムに選んだ．各辺の重みが乱数で設定されているが，全点対間最短経路アルゴリズム (APSP-SQUARE と APSP-DC) からの FMM サブルーチン呼び出しでは，入力となる行列は必ずしも相関がないとは限らないことに注意されたい（あるときは相関関係があるかもかもしれないし，またあるときは反相関関係があるかもしれない）．

表 4 に，グラフ RAND-G に対する結果を示す．この結果より，APSP-DC は APSP-FW や APSP-SQUARE より高速に動作することがわかる．また，図 8 に，n を入力グラフの頂点数としたとき，実行時間を n^3 で割った値の変化を示している．これより，APSP-DC の n^3 あたりの実行時間は n の増加と共に減少していくことが確かめられる．また，フロイド・ワーシャル法がたとえマシンのピーク性能を達成したとしても，APSP-DC の方が高速に動作することも分かる．(ここでいうマシンのピーク性能は，フロイド・ワーシャル法にのみ適用可能なものである.)

次に，APSP-DC にとって都合の悪い入力についての実験結果を示す．そのようなグラフとしては，ハイパーキューブ・グラフが考えられる．グラフ RAND-G では多くの頂点間で最短経路が一本しかないのに対し，ハイパーキューブ・グラフでは各頂点間に非常に多くの最短経路が存在する．そのようなグラフに対しては，APSP-DC アルゴリズムの中で呼び出される FMM 計算において，入力行列が反相関関係になりやすくなる．これを確かめるために，n 頂点のグラフとして $\log n$ 次元のハイパーキューブ (HYPERCUBE-G) を生成し，グラフの各枝の重みを全て 1 に設定した．

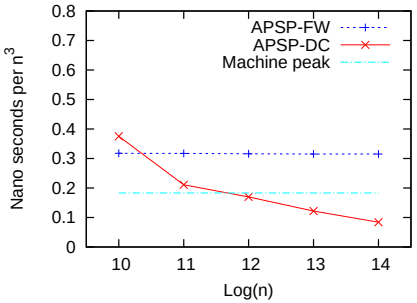

図 8 グラフ RAND-G に対する全点対間最短経路アルゴリズムの実行時間比較

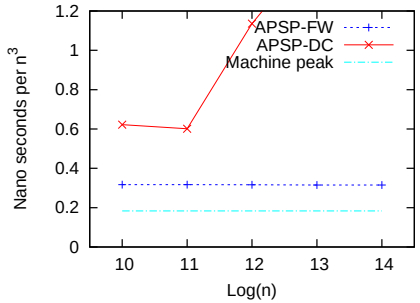

図 9 グラフ HYPERCUBE-G に対する全点対間最短経路アルゴリズムの実行時間比較

グラフ HYPERCUBE-G に対する実験結果を**表 5** と図 9 に示す．これらの結果から，我々のアルゴリズムは，グラフ HYPERCUBE-G に対しては必ずしも高速に動作しないことが分かる．

7. おわりに

本稿では，FMM 計算に対する高速なアルゴリズムを提案し，それをさらに SIMD 命令を活用して高速化した．そして，これを全点対間最短経路問題に適用し，分割統治法と組み合わせることで，フロイド・ワーシャル法より高速に解けることも示した．

今後の研究としては，まず，GPU 向けの実装を試みたい．フロイド・ワーシャル法では GPU を利用することで大幅な高速化が達成できることが示されており[15]，我々のアルゴリズムでも大幅な高速化が期待できるためである．他には，我々の FMM アルゴリズムの全点対間最短経路問題以外への適用も興味深い研究である．Williams と Williams[25] による

© 2011 Information Processing Society of Japan

と，非常に多くの問題（たとえば，非負重みを持つグラフに対して最小重みの三角形を見つける問題）が FMM と時間計算量の意味では等価と示されているからである．

謝　　辞

この研究は，総務省の地球温暖化対策 ICT イノベーション推進事業 (PREDICT) の助成により行われました．

参 考 文 献

1) Aho, A.V., Hopcroft, J.E. and Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).
2) Alon, N., Galil, Z. and Margalit, O.: On the exponent of the all pairs shortest path problem, *Journal of Computer and System Sciences*, Vol.54, No.2, pp.255–262 (1997).
3) Bondhugula, U., Devulapalli, A., Fernando, J., Wyckoff, P. and Sadayappan, P.: Parallel FPGA-based all-pairs shortest-paths in a directed graph, *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2006).
4) c, A.B., Gilbert, J.R. and Budak, C.: Solving path problems on the GPU, *Parallel Computing*, Vol.36, No.5-6 (online), DOI:10.1016/j.parco.2009.12.002 (2010).
5) Chan, T. M.: More algorithms for all-pairs shortest paths in weighted graphs, *Proc. of the 39th annual ACM Symposium on Theory of Computing (STOC)*, pp. 590–598 (2007).
6) Chatterjee, S. and Sen, S.: Cache-efficient matrix transposition, *Proc. of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, pp.195–205 (2000).
7) Coppersmith, D. and Winograd, S.: Matrix multiplication via arithmetic progressions, *Journal of Symbolic Computation*, Vol.9, No.3, pp.251–280 (1990).
8) Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithms, Third Edition*, The MIT Press (2009).
9) D'Alberto, P. and Nicolau, A.: R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks, *Algorithmica*, Vol.47, No.2, pp.203–213 (2007).
10) Floyd, R.W.: Algorithm 97: Shortest path, *Communications of the ACM*, Vol.5, No.6, p.345 (1962).
11) Goto, K. and vande Geijn, R.A.: Anatomy of high-performance matrix multiplication, *ACM Transactions on Mathematical Software*, Vol.34, No.3, p.Article No.12 (2008).
12) Han, S.-C., Franchetti, F. and Püschel, M.: Program generation for the all-pairs shortest path problem, *Proc. international conference on Parallel Architectures and Compilation Techniques (PACT)*, pp.222–232 (2006).
13) Harish, P. and Narayanan, P. J.: Accelerating Large Graph Algorithms on the GPU Using CUDA, *Proc. international conference for High Performance Computing (HiPC)*, pp.197–208 (2007).
14) Inoue, H., Moriyama, T., Komatsu, H. and Nakatani, T.: AA-sort: A new parallel sorting algorithm for multi-core SIMD processors, *Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp.189–198 (2007).
15) Katz, G.J. and Kider Jr., J.T.: All-pairs shortest-paths for large graphs on the GPU, *Proc. ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware*, pp.47–55 (2008).
16) Lacey, S. and Box, R.: A fast, easy sort, *Byte Magazine*, pp.315–320 (1991).
17) Marker, B., Zee, F. G. V., Goto, K., Quintana-Ortí, G. and van de Geijn, R.: Toward scalable matrix multiply on multithreaded architectures, *Proc. of 13th European Conference on Parallel and Distributed Computing (Euro-Par)*, pp.748–757 (2007).
18) McAuley, J. J. and Caetano, T. S.: An expected-case sub-cubic solution to the all-pairs shortest path problem in *R*, Technical report, CoRR (2009).
19) Penner, M. and Prasanna, V.K.: Cache-friendly implementations of transitive closure, *ACM Journal of Experimental Algorithms*, Vol.11, pp.185–196 (2006).
20) Takaoka, T.: All pairs shortest paths via matrix multiplication, *Encyclopedia of Algorithms*, Springer (2008).
21) Tiskin, A.: Synchronisation-efficient parallel all-pairs shortest paths computation, Technical report (2004).
22) Venkataraman, G., Sahni, S. and Mukhopadhyaya, S.: A blocked all-pairs shortest-paths algorithm, *ACM Journal of Experimental Algorithmics*, Vol. 8, p. Article No.2.2 (2003).
23) Vinjamuri, S. and Prasanna, V.K.: Transitive closure on the cell broadband engine: A study on self-scheduling in a multicore processor, *Proc. of the 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–11 (2009).
24) Warshall, S.: A theorem on Boolean matrices, *Journal of the ACM*, Vol.9, No.1, pp.11–12 (1962).
25) Williams, V.V. and Williams, R.: Subcubic Equivalences Between Path, Matrix, and Triangle Problems, *Proc. of the 51st IEEE Symposium on Foundations of Computer Science (FOCS)*, pp.645–654 (2010).

© 2011 Information Processing Society of Japan