完全準同型暗号による安全頻出パターンマイニングの NUMA マシン上での高速化

馬屋原 昂[†1]　今林 広樹[†1]　山名 早人[†2]

†1 早稲田大学基幹理工学研究科 〒169-8555 東京都新宿区大久保 3-4-1

†2 早稲田大学理工学術院〒169-8555 東京都新宿区大久保 3-4-1

E-mail: †{uma, imabayashi, yamana}@yama.info.waseda.ac.jp

あらまし 本稿では, データを秘匿した状態のまま計算できる完全準同型暗号（FHE: Fully Homomorphic Encryption）を NUMA マシン（マルチコア／マルチソケット）上で高速に実行するための仕組みとして，NUMA マシン上での各ソケットが持つローカルメモリへのデータ配置手法を提案する．昨今，様々なデータがクラウドに格納され処理されるようになると共に，遺伝子データや個人情報などの秘匿すべきデータが増大している．FHE は，こうしたデータを秘匿したまま計算できる暗号である．しかし，FHE は，膨大な時間・空間計算量を必要とする．一般的に FHE により暗号化された 1 データは数十から数百 MB となる．NUMA マシン上で，計算に必要となるデータが複数のソケット（メモリノード）に分割して格納されると性能低下につながる．本稿では，NUMA マシン上でのマルチコア並列処理において，ローカルメモリを最大限に利用するために，データ依存関係をもとに暗号文データを格納先するメモリノードを決定する手法を提案する．提案手法を完全準同型による Apriori アルゴリズムに適用したところ，約 3~4%の高速化を達成した．

キーワード FHE，HPC，NUMA，Apriori アルゴリズム

1. はじめに

近年のクラウドコンピュータ技術の発達により，クライアントが保持するデータを外部のクラウドサーバに計算委託する機会が増えている．クラウド上で医療データや商品の購買記録などの個人情報を扱う場合，クラウド側に具体的な情報を知られることなく計算委託できることが望まれる．これを実現する手法として秘密計算がある．秘密計算の中でも準同型暗号を利用することで，情報を秘匿しつつ，加法・乗法の演算を行うことができる．2009 年の Gentry の研究[1]を契機に加法・乗法の演算の回数制限がない完全準同型暗号（Fully Homomorphic Encryption, FHE）に関する研究が活発に行われている．しかし，FHE の演算には膨大な時間計算量と空間計算量を必要とする問題がある．

FHE が持つ膨大な時間計算量と空間計算量の問題を解決するアルゴリズムレベルでのアプローチとして，FHE のスキーム自体もしくは FHE のスキーム上で実装されたアルゴリズムの改良に焦点を当てる研究がある．また，アーキテクチャレベルのアプローチとして，GPU[2][3]や FPGA[4]を用いたオフロード計算よる高速化や独自のハードウェアデザイン [5][6][7]に関する研究がある．これに対し，本稿では，「FHE による暗号化によりデータが数十，数百 MB になること」に着目し CPU とメモリ間のデータ転送に着目した高速化を目指す．

FHE を高速化するためのメモリ最適化に関連する研究としては，Abozaid ら[6]の研究がある．Abzaid らは膨大な乗算に対する効率的なハードウェアデザインを提案している．通常，CPU の乗算器にはオペランドデータをワード単位で同時に入力し，その結果を出力する．しかし，FHE で用いる 1 データは，数十 MB となるため現在の CPU アーキテクチャでは同時入力できない．そこで，データをパイプライン化して入出力することにより，要求メモリバンド幅の削減とメモリの転送レイテンシの隠蔽を達成した．後に，FPGA での実装も発表している[4]．また，Cao ら[7]は，FHE で多用される FFT 演算高速化のために，非常に大きな整数に対する FFT 演算を低レイテンシで実現するためのアーキテクチャを提案している．

以上のように，Abozaid らや Cao らは，独自のハードウェアデザインによる高速化を目指している．これに対し，本研究では一般的に普及しているマルチコア／マルチソケットで構成される NUMA マシンを対象とし，データを各ローカルメモリへ最適配置することによる高速化を目指す．具体的には，FHE 利用時には，暗号文データが巨大となり，複数ソケットに分散されて格納されることを防ぎ，各コアの計算にローカルメモリ内に格納された暗号文データを用いることで，メモリアクセスレイテンシを削減し高速化を行う．また，同手法を FHE スキーム上の Apriori アルゴリズムに適

用する．

本研究の貢献は,FHE スキーム上の既存の Apriori アルゴリズムに関して，暗号文データの格納場所，計算コアを適切に設計した上で実際に実装を行い，高速化を達成したことである．

本稿の構成は，次に示す通りである．2 節にて関連研究について述べ，3 節にて予備実験を行う．4 節にて提案手法の詳細を説明する．5 節にて評価実験準備について説明した後に，6 節にて評価実験および結果の考察を行う．最後に，7 節にてまとめと今後の課題について述べる．

2. 関連研究

本節では， FHE スキーム上の既存の Apriori アルゴリズムとデータ配置・処理実行ノード決定アルゴリズムについて述べる．

2.1. 既存のアプリオリアルゴリズム

FHE を頻出パターンマイニングに応用した例として，Liu ら[8]の Privacy Preserving Protocol for Counting Candidate（P3CC）がある．今林ら[9]は Liu らの研究をもとに暗号文パッキング手法[10][11]を用いることで暗号文数を削減した．さらに，サポート値の計算結果をキャッシング・再利用することで，さらなる高速化を達成した．評価実験において，利用したスレッド数の記述はあるが，暗号文データの格納先メモリやスレッドの詳細な挙動に関しての議論は行っていない．

2.2. データ配置･処理実行ノード決定アルゴリズム

分散メモリマシンや分散共有メモリマシン（NUMA マシン）を対象とした並列プログラミングにおいては，データをどのメモリノードへ配置するか，そのデータをどの計算ノードで処理を実行するかが性能に大きな影響を与える．この課題への伝統的なアプローチとして Owner Computes Rule[12][13][14]がある． Owner Computes Rule では,演算結果を格納する先のメモリノードを基準として，処理を実行するコアを決定する．具体的には,データA, B, Cがそれぞれメモリノード 0，1，2 に格納されているとき，$C = A + B$という演算はメモリノード 2 をローカルメモリノードとするコアで実行される．

3. 予備実験－データ配置と処理性能

本節では，NUMA マシンにおいて，Owner Computes Rule を適用した際の性能を測定し，考察する．

ここで，NUMA マシンとは Non Uniform Memory Access マシンを意味しており，近年，多くの計算機サーバがこの構成をとっている．すなわち，一般的な NUMA マシンは 1 CPU が複数のコアで構成される（マルチコア）と共に，1 台の NUMA マシンに複数の CPU を備える（マルチソケット）．そして，NUMA マシン全体としてのメモリバンド幅を確保するために，メインメモリはソケットに分散して配置される．各ソケットが持つメモリのことをホームメモリもしくはローカルメモリと呼ぶ．したがって，例えば，CPU を 4 つ備える計算機サーバでは，ある CPU から別の CPU のローカルメモリにアクセスする場合と，自身のソケット内のローカルメモリへアクセスする場合では，メモリアクセスのレイテンシが異なる．以下では，ソケットをノードと呼び，ノードのホームメモリをローカルメモリと呼ぶこととする．

3.1. 測定手法

Algorithm 1 add_array を予備実験の測定対象の演算とする．要素数 n の 64bit 整数の配列 A, B, C を考え，それらの配列の添字を $\mathrm{i}(0 \le i < n)$とし，$\mathrm{C[i]} = \mathrm{A[i]} + \mathrm{B[i]}$を行う．この時，i の増加量（ストライド）を s とする．図 3-1 に n =5 のときの各ストライドにおける配列要素へのアクセス順序を示した．$\mathrm{s} = 1$のときは，シーケンシャルアクセスとなり，$\mathrm{s} > 1$の場合は s の値に依存してランダムアクセスに近い挙動となりうる． また，処理時間としてクロック数を採用し，m 回の測定時間の平均値を算出する．ループごとのキャッシュの影響を除去するために，ループの先頭でキャッシュをフラッシュする（loop head cache flush）．また，真偽値 f の値に応じて，さらにキャッシュフラッシュを行う（stride loop head cache flush）．このキャッシュフラッシュのタイミングは図 3-1 における下側の矢印による遷移時である．

Algorithm 1 add_array(*m*, *n*, *s*, *A*, *B*, *C*, *f*)

Input: # of loop *m*; array length *n*; stride *s*; array *A*, *B*, *C*; cache flush flag *f*;
Output: average clock cycle *ave*;

```
t := 0
for (j := 0; j < m; j ← j + 1) do
    flush_cache(A); # loop head cache flush
    flush_cache(B); # loop head cache flush
    for (k := 0; k < s; k ← k + 1) do
        if (f = true) then
            flush_cache(A); # stride loop head cache flush
            flush_cache(B); # stride loop head cache flush
        end if
        t1 := get_clock_cycle();
        for (i := k; i < n; i ← i + s) do
            C[i] ← A[i] + B[i];
        end for
        t2 := get_clock_cycle();
        t ← t + (t2 - t1);
        end for
    end for
ave ← t / m;
return ave;
```

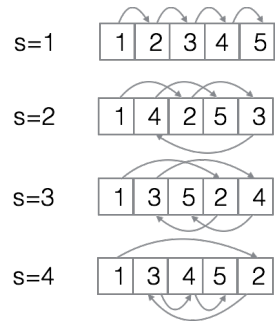

図 3-1 配列要素へのアクセス順序（n=5，s=1,2,3,4）

本予備実験ではデータサイズ別のシーケンシャルなメモリアクセス時の実行時間，ストライド別のランダムアクセスなメモリアクセス時の実行時間の比較を行う．なお，演算はノード単位ではなくコア単位での割り付けとした．データと演算の配置は 1)Owner Computes Rule を適用した場合（OCR case），2)実行コアが持つローカルメモリを用いない場合（worst case），3)実行コアが持つローカルメモリのみを用いる場合（best OCR case）であり，図 3-2 に示した．この時，各ノードに配置された「A」,「B」,「C」は配列データを,「run」は処理を実行するスレッドを表す．best OCR case は，データを同一のメモリノードに格納した場合に Owner Computes Rule を適用した挙動と同じである．

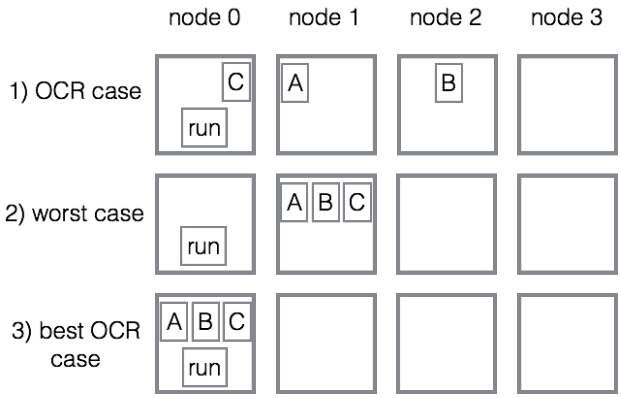

図 3-2 比較手法の配列・コアのノード配置

実験に使用する NUMA マシンのメモリノード数は 4, CPU モデル名は Intel(R) CPU E7-8880 v3 @ 2.30GHz, OS は CentOS 6.7，キャッシュ容量は L1，L2，L3 の順に 32KB，256KB，46,080KB である．また，コンパイラは g++(version 4.9.2)を用い，最適化オプションを「-O2」とした．

3.2. 結果と考察

本予備実験では m=10 として，10 回の測定結果の平均値を算出した．データサイズ別のシーケンシャルなメモリアクセス時の実行時間を表 3-1，ストライド別のランダムアクセスなメモリアクセス時の実行時間を表 3-2，表 3-3 に示した．表 3-2 では stride loop head cache flush により，直近のキャッシュヒットのみの結果となる．表 3-3 では stride loop head cache flush を行わないため，stride の値によっては直近と古いキャッシュヒットを含む結果となる．

表 3-1 実行時間
（データサイズ別，キャッシュフラッシュ有）(s=1)

メモリサイズ [8B]	クロック数 [Kcycle]		
n	OCR case	worst case	best OCR case
10^3	32	34	15
10^4	114	130	74
10^5	700	740	470
10^6	6,700	7,200	4,600
10^7	67,000	73,000	48,000
10^8	665,000	720,000	460,000

表 3-2 実行時間（キャッシュフラッシュ有）(n=10^8)

ストライド [-]	クロック数 [Mcycle]		
s	OCR case	worst case	best OCR case
10^0	649	698	446
10^1	6,280	7,230	4,390
10^2	14,000	16,400	8,600
10^3	14,100	15,400	8,630
10^4	15,000	16,100	8,860

表 3-3 実行時間（キャッシュフラッシュ無）(n=10^8)

ストライド [-]	クロック数 [Mcycle]		
s	OCR case	worst case	best OCR case
10^0	650	698	446
10^1	6,290	7,000	4,400
10^2	14,000	15,400	8,610
10^3	3,350	3,700	2,570
10^4	3,330	3,680	2,600

best OCR case では OCR case, worst case と比較して，表 3-1 では，処理時間が約 30~35%削減（$n = 10^6, 10^7, 10^8$），表 3-2，表 3-3 では，処理時間が順に約 40~45%，30~40%削減された（$s = 10^2, 10^3, 10^4$）．また，表 3-2 と表 3-3 を比較すると，全ての case において，キャッシュによって処理時間が約 70~80%削減された（$s = 10^3, 10^4$）．

結果をまとめると，best OCR case，OCR case，worst case の順で高速であり，シーケンシャルアクセス・ランダムアクセス共にローカルメモリを利用することで高い処理性能を得られる．なお，CPU のキャッシュにデータが残っている場合（表 3-3）は，リモートメモリへのアクセスが試行初回の 1 回のみになり高速になることが確かめられた．

4. 提案手法

本節では，NUMA マシンの CPU ノードに対応するローカルメモリに暗号文データを格納することで，メ

モリアクセス速度を改善する手法を提案する．予備実験の測定結果をもとに次の方針により,高速化を行う．

方針 1)演算に必要なデータ参照，演算結果の格納を同一ノード内で完結させる．
方針 2)同一データを用いる演算をできる限り同一ノードに配置し，L3 キャッシュを再利用する．
方針 3)リモートメモリアクセスが必要な場合，必要となる演算の前に，ローカルメモリへコピー（先行リード）する．

以下では,上記 3 方針に基づく提案手法を FHE スキーム上の Apriori アルゴリズムに適用することで高速化を行う．4.1 項にて暗号文データの計算依存性にもとづく task 生成とその実行に関して述べた後，4.2 項にて Apriori アルゴリズムへの適用への概要を述べる．4.3 項にて Apriori アルゴリズムにおける task 生成,4.4 項にて暗号文のマルチコアの並列乗算計算ついて述べる．

4.1. データ依存に基づく task 生成とその実行

本項では，暗号文データ依存について述べた後に，そのデータ依存の解消方法とキャッシュフレンドリーな task の実行方法を述べる．まず，データ集合を C とする．$f: C^2 \to C,\ (x,y) \mapsto z = f(x,y)$を考える．FHE では任意の回数の加算と乗算が可能であるため,$f(x,y) = x+y$または$f(x,y) = x \cdot y$とする．この加算または乗算の処理を行う写像 f を task と呼ぶ．task の定義から，ある task の出力が他の task の入力となりうることは明らかである．ただし，相互依存関係にある task については考えず，DAG を想定する．図 4-1 に task の例を示す.この時,$task_{(a)(b)}$は入力データ C_a, C_b から $C_{(a)(b)}$ を出力する．$task_{(x)(y)}$は入力データ C_x，C_y から $C_{(x)(y)}$を出力する．$task_{((x)(y))(z)}$は $task_{(x)(y)}$の出力 $C_{(x)(y)}$と C_z を入力データとして，$C_{((x)(y))(z)}$を出力する．

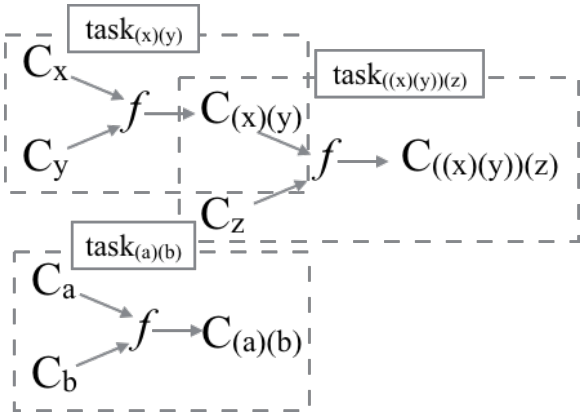

図 4-1 task の例

図 4-1 において，$task_{(x)(y)}$，$task_{(a)(b)}$間にはデータ依存がないため，並列に実行可能である．このように他の task とデータ依存を持たない task を task priority queue へ入れ，task priority queue から当該 task の優先度に基づき task を取り出し順次計算を実行する．以上を繰り返すことで計算を進める．具体的には，以下の手順により task の優先度を付与し，計算を行う．

手順 1) 計算に必要な task を生成し，この全ての task が終了するまで，手順 2)～手順 5)を繰り返す．
手順 2) データ依存を持たない，もしくは解消された task を task priority queue へ push する．
手順 3) queue 内に格納されている task が入力とする入力データ各々について，いくつの task から参照されているかを数え上げ，その数を当該入力データの優先度とする．
手順 4) 最も高い優先度を持つ入力データ（最優先データ）を入力とする task に最も高い優先度を付与する．
手順 5) 最も高い優先度の task を task priority queue から pop し実行する．該当する task が複数ある場合には，それらを並列実行する．

手順 5)において，並列実行時にはスレッドプールを用いて処理を行う．この時，各コアに 1 スレッドを割り当てるため,並列実行可能数はコア数と同じとする．該当する task 数が並列実行可能数を超える場合には，task を pop した順番にスレッドプールで実行する．上記の優先度決定方法を用いることで，同一のデータを参照する task を連続して pop することができる.次に，pop した task をその入力データと同一ノードのコアを用いて実行させることで,L3 キャッシュのヒット率向上による高速化を図る．これは方針 2 に該当する．

なお，上記の方法のみでは，task 間のデータ依存を考慮して優先度を付与していないために，プログラム依存グラフ(データ依存及び制御依存からなるグラフ)において，クリティカルパスにある task の実行が遅延させられる場合があるが，以下の Apriori では，全てデータ並列に計算可能であるため考慮していない．つまり，クリティカルパス問題については，本稿の対象外とし今後の検討課題とする．

4.2. Apriori アルゴリズムへの適用の概要

本実装では，今林ら[9]の Apriori アルゴリズムをベースとする．サーバ側の Apriori アルゴリズムの流れを図 4-2 に示し，提案手法にあたる部分を太字で示した．

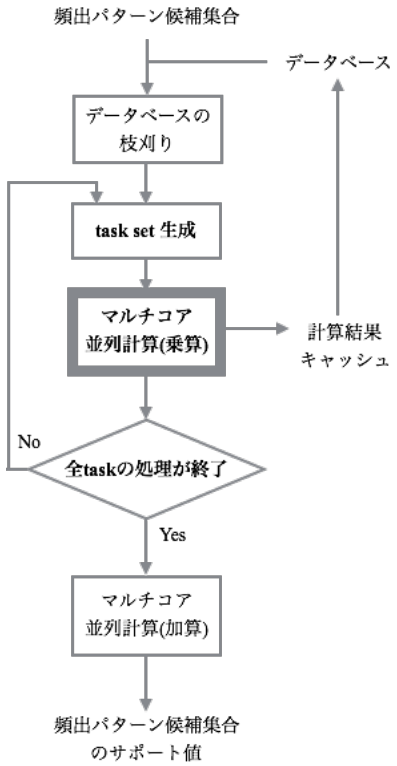

図 4-2 Apriori アルゴリズムの流れ（サーバ側）

まず，サーバはクライアントから頻出パターン候補集合を受け取る．サーバは，受け取った候補パターンに含まれないアイテムデータをデータベースから削除する（データベースの枝刈）．次に，トランザクション毎，候補パターン毎に，task を生成し，マルチコアで並列に暗号文データの乗算計算を行う．各 task では，あるアイテムが当該トランザクションに含まれるかどうかを判定するための前処理計算を行っており，task 間にはデータ依存関係及び制御依存関係が無いため並列に実行可能である．

なお，task 内の計算においては，同一の乗算が複数回発生する可能性があるため，乗算の計算結果をキャッシュし高速化する [9]. なお，ここで言うところのキャッシュはアプリケーションキャッシュであり，CPU が持つキャッシュではない．

全トランザクション，全候補パターンに対する前処理が完了したら，その結果に対して候補パターン毎に総和を求める（加算）ことで，各候補パターンのサポート値を求める．求められたサポート値（暗号文）はクライアントに戻され，クライアント側で頻出であったかどうかが判断され，アイテム長がインクリメントされた候補パターン集合が再びサーバに送られる．このようにして FHE による Apriori が実行される．

ここで，本稿の提案手法の対象となる演算は，太枠で囲んだ「マルチコア並列計算（乗算）」に該当する処理であり，サポート値計算におけるボトルネックとなる部分である．

以下，task 生成については 4.3 項，マルチコアの並列乗算計算については 4.4 項にて後述する．

4.3. Apriori アルゴリズムにおける task 生成

図 4-3 に Apriori アルゴリズムで用いるバイナリ化されたデータベースを示す．アイテム数を N_{item}，トランザクション数を N_{trans} とした．トランザクション ID を t, アイテム ID を i とし, 該当する平文を m_{t_i} とし，暗号文を C_{t_i} = Enc(m_{t_i})とする．m_{t_i} は，トランザクション t がアイテム i を含む場合 1, 含まない場合 0 となる．

	I_1	I_2	I_3	…	I_N_{item}
T_1	0	0	0	…	0
T_2	1	0	1	…	1
T_3	0	1	0	…	0
T_4	1	0	1	…	0
T_5	0	0	1	…	1
…	…	…	…	…	…
T_N_{trans}	0	1	0	…	1

図 4-3 Apriori アルゴリズムで用いるデータの例

頻出パターン候補として，{1，2，3}が与えられたとき，サポート値は$\sum_{t=1}^{N_{trans}}(C_{t_1} \cdot C_{t_2} \cdot C_{t_3})$により計算できる. つまり, 同一トランザクション ID のアイテムデータ全て（頻出パターン候補が持つアイテム全て）について乗算し，最後にその結果を足し合わせる．

Task 例を図 4-4 に示す．この時，写像 f はすべて乗算処理となるため，表記を省略した．トランザクション数は 2, 頻出パターン候補集合は{1, 2}, {1, 3}, {4, 5}が与えられたとした．また，頻出パターン候補のアイテム数が 3 以上の場合は，アイテム数が 2 の時と同様に task を生成することにとなる．つまり，頻出パターン候補{1，2，3}，{1，2，4}が与えられる場合には{1, 2}の計算結果キャッシュにより, {{1, 2}, 3}, {{1, 2}, 4}となる．また，計算結果キャッシュが存在しない場合にはパスが 2 段となる．

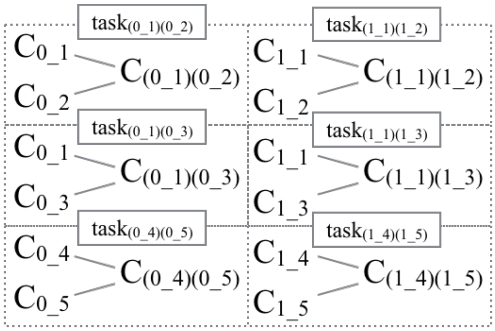

図 4-4 乗算処理の task 例

図 4-4 に示した task はトランザクション毎，候補パターン毎に作成されている．今，方針 1 により各 task の入力データは全て同一のノードに格納されることを前提とする．ここで，各 task は，トランザクション毎，アイテム毎に作成されているので，図 4-3 のデータに対して様々なメモリノードへの配置が考えられるが，ここでは，トランザクション ID が同じ全てのアイテムデータを同一のメモリノードに格納するとする．

例えば，4 個のノードにそれぞれに CPU とメモリがある場合を考える．$n = \lceil N_{trans}/4 \rceil$とし，適切なメモリノードに格納すると図 4-5(a)となる．この時，各処理の終了時間を均等にするために，メモリノードへの割当を均等にする．しかし，4 等分したデータをそのままとまりごとに順番に割り当てる必要性はない．つまり，データの入力がストリーム的な場合は，全体のデータ数が予め定まらず，図 4-5(b)に示した分け方となる．

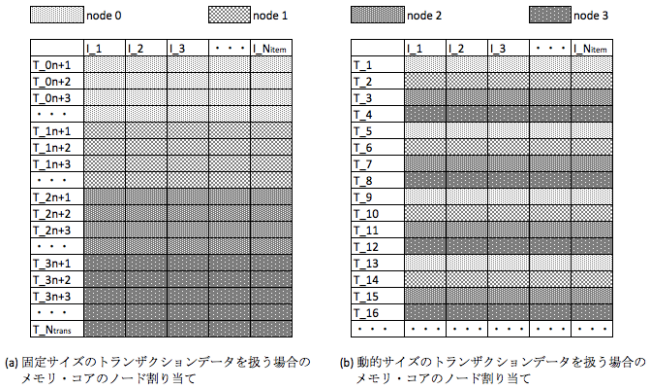

図 4-5 メモリ・コアのノード割り当て

4.4. マルチコアの並列乗算計算

図 4-4 の task において，4.1 項で示した task priority queue の優先度決定の具体的な流れを説明する．図 4-6 に task priority queue とデータ優先度について示した．

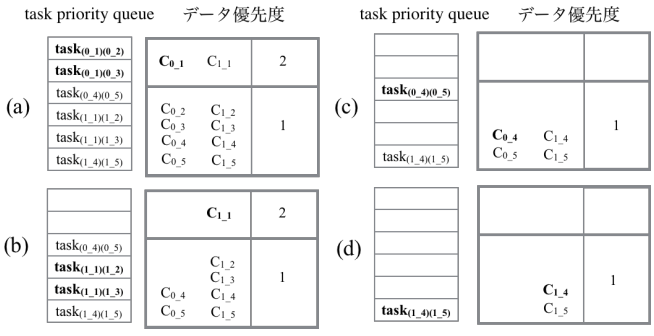

図 4-6 task priority queue とデータ優先度

まず，6 つの task は何れもデータ依存が存在しないため，手順 2 の通りに，全ての task を task priority queue に push する．そして，手順 3 の通りに，入力データの優先度を計算し，最優先データを太字で示した．手順 4 の通りに，最優先データを入力とする task においても太字で示した．手順 5 の通りに，task を並列実行する．上記の手順を繰り返すと，図 4-6 の(a)，(b)，(c)，(d)の順番通りになり，(d)の処理が終了することで，全ての task が実行される．この時，データの配置は図 4-5 に示す通りトランザクション毎に異なるノードに割り当てられており，トランザクション ID=0 の計算については，トランザクション ID=0 のデータをローカルに保存しているノードで実行される．一方，トランザクション ID=1 のデータを参照する 3 タスクは別のノードで実行される．

なお，暗号文の乗算では公開鍵を参照するため，方針 3 より，公開鍵は各ノードにコピーされ，どのノードのコアで実行してもローカルメモリへのアクセスができる状態となる．公開鍵は大量に存在する暗号文データと比較すると，コピーによるメモリ圧迫は無視できる程度である．

5. 評価実験準備

評価実験は BGV スキームによる FHE 実装の HElib[1] ライブラリをベースとする．これは C++で書かれたオープンソースライブラリである．本提案手法を適用する際に，必要な改変を HElib に施し，それを HElib+と呼ぶ．

評価実験では CentOS6.7 で行い，numactl コマンドを用いて，NUMA policy を設定する．NUMA policy の明示的な記述がない場合の設定は preferred node: current とする．この設定では，現在実行中のコアのローカルメモリを優先的に使用する．しかし，glibc の malloc(3)では，システムコールを経由せず，内部で確保されたメモリプールを用いる場合には NUMA policy を無視する．そのため，glibc 実装ではなく著者による実装の numalloc を用いる．

HElib の依存ライブラリとして，算術演算ライブラリの NTL[2] と GMP[3]が挙げられる．NUMA policy を preferred node: current と設定することで，この 2 つのライブラリを変更する必要がない[4]．

5.1. HElib+作成理由と変更箇所

HElib をそのまま利用した場合，以下の 2 点の問題があるため HElib+を作成した．

[1]http://shaih.github.io/HElib/
[2]http://www.shoup.net/ntl/
[3]https://gmplib.org/

[4]NUMA API を用いて，メモリノードを直接指定してメモリ確保を行う場合，既存ライブラリ書き換える必要があり，現実的ではない．

問題点 1 ：

HElib では，暗号文同士の演算を行う際に，一時変数を作成し，演算処理を行う．さらに，演算結果をもとの変数に代入する．具体的には，暗号文C_1，C_2の乗算結果C_{12}を取得したい場合，$C_{_tmp_1} = C_1 \cdot C_2$の後に，$C_1 = C_{_tmp_1}$を行い，$C_1$変数に$C_{12}$の値を格納する．

問題点 2 ：

HElib が提供する乗算メソッドのインタフェースとして，以下の 2 通りが存在する．

メソッド 1) Ctxt& Ctxt::operator*=(const Ctxt& other);

メソッド 2) void Ctxt::multiplyBy(const Ctxt& other);

Ctxt とは暗号文クラスであり，メソッド 2)の内部ではメソッド 1)を呼んでいる．この時，1 つの暗号文を複数回の演算で利用する場合に問題が生じる．具体的には，暗号文C_1，C_2，C_3に対して，$C_{12} = C_1 \cdot C_2$，$C_{13} = C_1 \cdot C_3$の演算結果を取得したいときを考える．前述の通り，演算結果をもとの変数に代入してしまうことから，メソッドを呼ぶ前に暗号文の一時変数を用意しなければならない．つまり，$C_{tmp\_1} = C_1$，$C_{12} = C_{tmp\_1} \cdot C_2$，$C_{13} = C_1 \cdot C_3$とする必要がある．さらに，今後$C_1$を別の演算で使用する可能性がある場合には，一時変数の個数が 1 個分増加する．

上記問題は，余計なコピーが増えるという点にある．これはコピーコストが増えるだけではなく，演算毎に，現在実行中のコアのローカルメモリに一時変数が確保されることになる．したがって，暗号文のデータが初期位置とは異なる場所に配置されてしまうため，提案手法の結果の比較・考察の妨げになる．これを防ぐため，暗号文の乗算を一時変数のコピーを用いない方法に変更した．具体的には，Ctxt& Ctxt:: operator*=(const Ctxt& other);メソッドを変更した．

また，4.4 項にて述べた公開鍵を各ノードにコピーして配置するために，公開鍵のアドレスの一致を確認している assert 機能を無効化した．

5.2. numalloc

glibc 実装の malloc(3)では NUMA マシン上で動作することを前提としていないため，著者による実装の numalloc を用いる．numalloc では内部のメモリプールからメモリを確保する際においても，NUMA policy の通りにメモリを確保する．numalloc の詳細については本稿とは関係ないため，省略する．

6. 評価手法

本実験では 次の 3 種類のメモリノードの利用方法を比較する．図 6-1 にそれぞれの利用方法を示した．

1. アイテムベース
   アイテムごとに同じメモリノードに格納する．
2. トランザクションベース
   トランザクションごとに同じメモリノードに格納する．これは提案手法の通りである．
3. ランダム
   暗号文データを格納するノードをランダムに決定する．

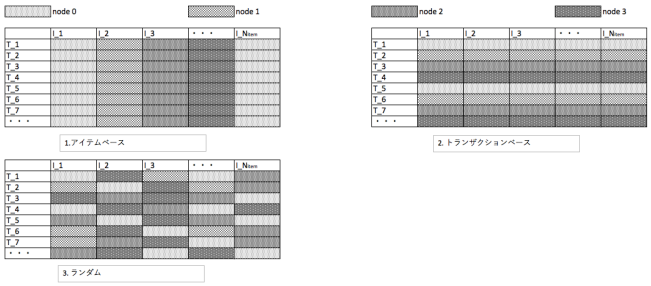

図 6-1 4 種類のメモリノードの利用方法

演算時のスレッド制御は提案手法におけるスレッド制御に合わせる．また，計測は 8 回の処理時間の最大値と最小値を除いた平均値を用いる．この時，キャッシュの影響を考慮し，繰り返しの計測を行う際には暗号文の生成からやり直す[5].

6.1. 実験環境

実験を行う環境について以下の表 6-1 にまとめた．

表 6-1 実験環境

名称	値
OS	CentOS 6.7（64bit）
Linux kernel version	2.6.32-504.30.3
CPU model name	Intel(R) CPU E7-8880 v3 @ 2.30GHz
# of CPU	4
# of core	72(18x4)
# of memory node	4
memory size	1TB(256GBx4)
L1d cache	32KB
L1i cache	32KB
L2 cache	256KB
L3 cache	46,080KB
glibc version	2.12-1
ulimit system resource setting	default
g++ version	4.9.2
g++ optimization option	-O2

6.2. パラメータ

評価実験に用いたパラメータについて表 6-2 に示した．括弧内はデフォルト値である．$p^r < N_{trans}$となるようにｐとｒの値を設定し，l は複数回の乗算が可能な値を設定した．また，N_{freq_cand}はN_{item}における 2 種類のアイテムの組み合わせ数とした．これは全てのアイテ

[5]しかし，一般的に FHE により暗号化された 1 データは数十から数百 MB となるため，前回のループのデータがキャッシュに残っている可能性は非常に低い．

ムが頻出であり，頻出パターン候補のアイテム長=2 の場合に相当する．本実験において，トランザクションデータの具体的な値は評価実験に影響を与えないため，glibc の random(3)の疑似乱数を用いて生成した 0 と 1 の値を用いた．

表 6-2 評価実験のパラメータ

パラメータ名	値	説明
p	11	p^r で平文空間
r	7	p^r で平文空間
k	(80)	セキュリティー パラメータ
l	8	FHE の回路の深さ（レベル）
c	(3)	キースイッチ行列の行数
w	(64)	秘密鍵に用いるハミング重み
N_{slot}	915	スロット数
N_{item}	18	アイテムの種類
N_{trans}	1,943,424	トランザクション数
N_{freq_cand}	153	頻出パターン候補数

6.3. 結果と考察

実験結果を表 6-3 に示した．まず，HElib, glibc malloc ではなく HElib+, numalloc を用いることで，各暗号文データ配置パターンにおいて，約 60〜70 倍の高速化を達成した．次に，提案するライブラリを適用した結果における各暗号文データ配置パターンの考察を行う．トランザクションベースのパターンでは，他のパターンと比較して，約 3~4%の高速化を達成した．ただし，この処理時間の差は純粋なリモートメモリアクセスのみによるものではない可能性がある．具体的には，暗号文データ配置が異なることから生じる副作用として，メモリアロケートのタイミングが異なることが挙げられる．

表 6-3 実験結果

ライブラリ	暗号文データ 配置パターン	処理時間[s]
HElib glibc malloc	アイテムベース	521.6
	トランザクション ベース	480.2
	ランダム	601.5
HElib+ numalloc	アイテムベース	8.109
	トランザクション ベース	7.775
	ランダム	7.996

7. まとめ

本稿では，NUMA マシン上のマルチコア並列処理において，ローカルメモリを最大限に利用するために，データの計算依存性をもとに暗号文データの格納先のメモリノードを決定する手法を提案した．提案手法を FHE スキーム上の Apriori アルゴリズムに適用することで，約 3~4%の高速化を達成した．

今後の研究課題として，1)FHE スキーム上で構成される Apriori アルゴリズム以外のアルゴリズムに対して適用し，実験をすること，2)複数のメモリノードに格納されたデータを参照する必要のある計算に対応できる手法への拡張，3)暗号文データがメモリ・ディスク間を移行する場合においても，メモリの使用量を制限しつつ，処理時間の増大を抑えることが挙げられる．

謝辞 本研究は，科学技術振興機構（JST）CREST の支援を受けたものである．

参 考 文 献

[1] Gentry, C.: "A fully homomorphic encryption scheme", PhD Thesis, Stanford University, 2009, https://crypto.stanford.edu/craig/.

[2] Khedr, A., Glenn, G., and Vinod, V.: "SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers.", IEEE Trans. on Computers, vol. 65, issue 9, pp. 2848-2858, 2015.

[3] Dai, W., and Berk, S.: "cuHE: A Homomorphic Encryption Accelerator Library.", Proc. of the 2nd Int'l Conf. on Cryptography and Information Security in the Balkans (BalkanCryptSec), LNCS, vol. 9540, pp. 169-186, 2015.

[4] Abozaid, G., and Ahmed, E.: "Design Space Exploration for a Co-Designed Accelerator Supporting Homomorphic Encryption.", Proc. of the 20th Int'l Conf. on Control Systems and Computer Science (CSCS), 2015.

[5] Doröz, Y., Erdinç, Ö., and Berk, S.: "Accelerating fully homomorphic encryption in hardware.", IEEE Trans. on Computers, vol. 64, issue 6, pp. 1509-1521, 2015.

[6] Abozaid, G., Ahmed, E., and Yasutaka, W.: "A scalable multiplier for arbitrary large numbers supporting homomorphic encryption," Proc. of the 2013 Euromicro Conf. on Digital System Design (DSD), 2013.

[7] Cao, X., Ciara M., Máire O., et al.: "Optimized Multiplication Architectures for Accelerating Fully Homomorphic Encryption", IEEE Trans. on Computers, vol. 65, no. 9, pp. 2794-2806, 2016.

[8] Liu, J., Li, J., Xu, S., et al.: "Secure Outsourced Frequent Pattern Mining by Fully Homomorphic Encryption", Proc. of the 17th Int'l Conf. on Big Data Analytics and Knowledge Discovery (DaWaK), LNCS, vol. 9264, pp. 70–81, 2015.

[9] Imabayashi, H., Ishimaki, Y., Umayabara, A., et al.: "Secure Frequent Pattern Mining by Fully Homomorphic Encryption with Ciphertext Packing", Proc. of the 11th Int'l Workshop on Data Privacy Management (DPM) , LNCS vol. 9963, pp. 181-195, 2016.

[10] Smart, N.P. and Vercauteren, F.: "Fully homomorphic encryption with relatively small key and ciphertext sizes", Public Key Cryptography-PKC 2010, LNCS, vol. 6056, pp. 420-443, 2010.

[11] Smart, N.P. and Vercauteren, F.: "Fully homomorphic SIMD operations", Designs, Codes and Cryptography. vol. 71, no. 1, pp. 57-81, 2014.

[12] Zima, H.P., Bast, H.-J., and Gerndt, M.: "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization", Parallel Computing, vol. 6, issue 1, pp. 1-18, 1988.

[13] Callahan D., and Kennedy, K.: "Compiling programs for distributed-memory multiprocessors.", J. of Supercomputing, vol. 2, issue 2, pp. 151-169, 1988.

[14] Rogers, A., and Pingali, K.: "Process Decomposition Through Locality of Reference", Proc. of the ACM SIGPLAN 1989 conf. on Programming Language Design and Implementation (PLDI), pp. 69-80, 1989.