

メニーコア CPU 環境における準同型暗号演算高速化を目的とする タスクスケジューリング手法の検討

鈴木拓也[†] 石巻優[†] 山名早人[†]

概要：近年利活用されているクラウドコンピューティングには、クラウド上にアップロードされたデータにおける機密情報が漏洩する危険性がある。この危険性を回避する一つの方法として、暗号文上での演算が可能な準同型暗号を利用することが挙げられる。しかし、準同型暗号は処理速度が遅いことが知られている。本研究では、メニーコア CPU を搭載したマシン上における準同型暗号処理の高速化を行う。特に、準同型暗号演算で構成された関数を静的および動的にスケジューリングすることで CPU コア使用率を高める。評価実験では、Xeon Phi 7230F (64 コア) 上で、複数の関数をそれぞれ特定時刻に開始し、個々の関数や全ての関数を実行し終えるまでにかかった処理時間を測定した。そして、1つの準同型暗号演算を実行するスレッド数を、静的に決定するナイーブ手法と CPU 使用状況や関数実行状況を考慮して動的に決定する手法とを比較し、動的に決定する手法の有効性を検証し、さらなる改善内容を検討した。

キーワード：準同型暗号、スケジューリング、メニーコア CPU

1. はじめに

近年、ユーザが高性能な計算資源を保有することなく、計算処理を行うことが可能なクラウドコンピューティングが利活用されている。ユーザは、データをクラウド上にアップロードし、目的の計算をクラウド上で実行し、結果をクラウドからダウンロードする。しかし、クラウド上にアップロードしたデータが漏洩する危険性があり、機密情報が流出する恐れがある。この危険性を回避する方法の一つとして、暗号化したまま演算可能な準同型暗号を用いることが挙げられる。しかし、準同型暗号演算は時間計算量と空間計算量が大いという問題点があるため、高速化する必要がある。

準同型暗号演算を高速化するためには、CPU アーキテクチャに合わせた実装や実行をすべきである。近年の CPU は、シングルスレッド性能の成長が停滞しており、コア数を増やすことで性能向上を図っている。このため、メニーコア CPU の性能を引き出すために、可能な限り多くのコアを使用し続ける必要がある。

メニーコア CPU のコアをできるだけ多く使用するために、静的スケジューリングと動的スケジューリングの2種類の方法が用いられる。静的スケジューリングは、実行前に予め確定している情報を用いて、オフラインで最適化する手法である。例えば、予めイテレーション回数が既知のループ処理を並列処理することで、メニーコア CPU の性能を引き出すことが可能である。一方、動的スケジューリングは、実行時に確定する情報を用いて最適化する手法である。外部から入力されたデータに対して何らかの処理を行うアプリケーションでは、変化する状況に応じて適切に処理を実行しなければ、メニーコア CPU の性能を活かしきれない。

本研究は、準同型暗号演算を対象に、静的・動的スケ

ジューリングを適切に行うことで、準同型暗号演算を高速に実行できるようにすることを目的とする。本稿では、計算資源に応じて、準同型暗号演算に適した動的スケジューリングを実現するために、準同型暗号演算に対して使用するスレッド数を静的に決定するナイーブ手法と動的に決定する手法を比較する。そして、動的に決定することの有効性を検証し、改善内容を検討する。

本稿の構成を以下に示す。2 節で準同型暗号に関して説明する。3 節で実験に使用した準同型暗号パラメータと計算機について説明し、4 節で予備実験を示す。5 節で本研究における解決したい課題に関して説明し、6 節で関連研究を示す。7 節で実験について説明する。8 節で実験結果と考察を示す。最後に 9 節でまとめを行う。

2. 準同型暗号について

2.1 準同型暗号の概要

準同型暗号には、暗号化したまま加算が可能な加法準同型性や、暗号化したまま乗算が可能な乗法準同型性という性質により分類できる。例えば、加法準同型性のみを持つ加法準同型暗号は Paillier 暗号[1]があり、乗法準同型性のみを持つ乗法準同型暗号は RSA 暗号がある。そして、加法準同型性と乗法準同型性の両方を持つ完全準同型暗号 (Fully Homomorphic Encryption, FHE) が 2009 年に Gentry によって発表された[2]。そして、現在も準同型暗号は改良が加えられている。以下では、Ring-LWE ベースの準同型暗号[3]を対象に述べる。

準同型暗号では、暗号文にノイズを含ませることで、解読困難性を高めている。そして、準同型暗号演算を行う度に暗号文が持つノイズが増加する。特に、準同型乗算を適用すると、暗号文のノイズが大きく増加する。そして、ノイズ量が規定値を超えると正しく復号することができなくなる。そのため、FHE は、1つの暗号文に対して数回程度

[†] 早稲田大学
Waseda University

の準同型乗算が可能な Somewhat Homomorphic Encryption (SwHE) や規定回数まで準同型乗算が可能な Leveled Homomorphic Encryption (LHE) [5]と、ノイズ量を減らす Bootstrapping と呼ばれる処理を組み合わせることで、任意の回数の準同型暗号演算を実行可能としている。Bootstrapping はパラメータによっては数十秒間から数分間以上かかる[4]1。したがって、1つの暗号文に対して適用される準同型乗算の回数に決まっている場合は、SwHE や LHE を用いることが望ましい。また、Bootstrapping を実行しない場合での準同型乗算を行える残り回数 l (以下、レベル)の初期値 L ($0 \leq l \leq L$)は、準同型暗号のパラメータに依存する。レベルが大きいほど各準同型暗号演算の時間計算量と空間計算量が大きくなる。

現在、Ring-LWE ベースの準同型暗号の方式には、BGV 方式[5,6]や B/FV 方式[7,8,9], CKKS 方式[4,10,11]が存在する。これらの方式では、パッキング (パッチング) [12,13]と呼ばれる、1つの平文や暗号文に複数の整数値や実数値を格納し、ベクトルのように扱う手法が利用できる。パッキングにより SIMD 演算でき、スループットが向上する。

暗号文 ct は多項式環上で表現され、2つ以上の多項式で構成される。したがって、各項の係数の法を Q とし、多項式環の次数を N とすると、実装上では暗号文 ct は例えば $\mathbb{Z}_Q^{2 \times N}$ の行列となる。ただし、 Q は数十〜数百[bit]の大きさである。 Q が CPU のワードサイズよりも大きい場合2は多倍長整数を用いる必要がある。以降、中国人剰余定理を適用し、ワードサイズ以下の大きさである q_i を用いて $Q = \prod_{i=0}^l q_i$ とすることを RNS (Residue Number System) と呼ぶ。ここで、 q_i の数を $n_{q_i} = l + 1$ と定義する。RNS を用いることで、各項の係数をシングルワードごとに独立に n_{q_i} 並列で計算でき、並列性が向上する[6, 9, 11]3。

2.2 CKKS 方式について[4,10,11]

本稿で使用する CKKS 方式を説明する。CKKS 方式は LHE をベースとした FHE の 1 つである。値に誤差が含まれることを許容することで、近似的に実数を表すことができるため、CKKS 方式は実数値を扱うのに最も適した方式である。また、格納した実数値の桁情報を持つスケールと呼ばれる値が各暗号文に備わっている。準同型暗号演算や復号化を正しく行うために、スケールを適切に管理する必要がある。

本稿の実験で用いた SEAL ライブラリ[14]では、RNS 表現を用いて CKKS 方式を実装しており、平文は $m \in R$, 暗号文は $ct = (ct^{(j)})_{0 \leq j \leq l} = \left((c_0^{(j)}, c_1^{(j)}) \right)_{0 \leq j \leq l} \in \prod_{j=0}^l R_{q_j}^2$ と表される。ただし、 $R = \mathbb{Z}[X]/(X^N + 1)$ を N 次の多項式環とし、

$R_a = (\mathbb{Z}_a[X])/(X^N + 1)$ は、 R の各項の係数を a で剰余した多項式環とする。また、 $c_k^{(j)} \in R_{q_j}$ である。SEAL ライブラリの実装では、各項の RNS 表現における係数値をデータとして保持している。そのため、平文 m は $n_{q_i} \times N$ の配列として扱うことができる。また、暗号文 ct は一般化すると $ct \in \prod_{j=0}^l R_{q_j}^K$ と表すことができ、 $K \times n_{q_i} \times N$ の配列として扱うことができる。 K の初期値は 2 であり、入力が 2 つの暗号文である準同型乗算の過程で 1 増加し、後述の Relinearization と呼ばれる処理を適用すると 1 減少する。

最後に、本稿で用いる準同型暗号演算について説明する。

- Relinearization : 暗号文のサイズ K を小さくすることを目的とした準同型暗号演算である。ただし、出力暗号文のサイズ K を 2 未満にすることはできない。準同型乗算の過程で Relinearization を適用しないと、暗号文のサイズが増加していき、準同型乗算を実行する度に時間計算量と空間計算量が増加する問題が生じる。
- Rescaling : レベルを l から $l-1$ に小さくすることで、暗号文の持つノイズを減らすこととスケールを調整することを目的に用いられる。
- Modulus Reduction : レベルの削減を行う。ある準同型暗号演算と入力の組が与えられた時、当該入力となる全ての平文4と暗号文のレベルは同一である必要があり、レベルを調整するために用いられる。

なお、以降では、SEAL ライブラリの関数名を参考に、準同型加算と準同型乗算に関して、下記のように定義する。

- AdditionPlain : 平文と暗号文の準同型加算
- Addition : 暗号文同士の準同型加算
- MultiplicationPlain : 平文と暗号文の準同型乗算 (MultiplicationPlain, Rescaling の順で適用) の一部
- Multiplication : 暗号文同士の準同型乗算 (Multiplication, Relinearization, Rescaling の順で適用) の一部
- Square : 暗号文の 2 乗 (Square, Relinearization, Rescaling の順で適用) の一部

3. 本稿で使用する計算機とパラメータ

計算機には、CPU は Xeon Phi 7230F (64 物理コア、同時マルチスレッディングによる 256 論理コア) が搭載されており、メモリは 6 チャンネルで合計 384GB の DDR4 メモリが搭載されている。なお、Xeon Phi 7230F に搭載されている 16GB の MCDRAM メモリは使用しない。

準同型暗号ライブラリである SEAL ライブラリで実装されている CKKS 方式を用いた。準同型暗号パラメータは、特に記述しない場合は、多項式の次数 N を 32768, レベルの初期値 L を 7 ($n_{q_i} = 8$), $\log_2 q_i = 60$ とした。

1 平文が 2 値である FHEW 方式では 0.69 秒、TFHE 方式では 0.1 秒以下で実行可能である。なお、本稿では FHEW 方式と TFHE 方式は扱わない。

2 例えば、64bitCPU では $\log_2 Q \geq 64$ を満たす場合である。

3 例えば、準同型加算は RNS を用いない場合は $2N$ 回の多倍長整数加算に対し、RNS を用いる場合は $2N(l+1)$ 回のワードサイズでの加算である。

4 CKKS 方式では、平文にもレベルが存在する。

表1 シングルスレッド時での(レベル+1)の値である n_{q_i} に対する各準同型暗号演算にかかる実行時間[ms]

		n_{q_i}						
		2	3	4	5	6	7	8
準同型暗号演算名	AdditionPlain	1	1	2	2	2	3	3
	Addition	1	2	2	3	3	4	4
	MultiplicationPlain	6	9	11	14	17	20	22
	Multiplication	14	21	27	34	40	48	55
	Square	10	15	19	25	29	34	39
	Relinearization	33	65	101	151	203	271	349
	Rescaling	31	53	72	96	116	140	162

コンパイラはGCC6.3.1を使用した。コンパイルオプションは、SEALライブラリには「-O3 -DNDEBUG -fPIC -maes -fopenmp -pthread -std=gnu++14」とし、その他のプログラムには「-O3 -DNDEBUG -Wall -march=native -maes -fopenmp -pthread -std=gnu++14」とした。

4. 準同型暗号演算の並列性に関する予備実験

4.1 並列処理の必要性

SEALライブラリのサンプルプログラムを元に、(レベル+1)の値である n_{q_i} を2~8のそれぞれについて、Xeon Phi 7230F上で測定した各準同型暗号演算に対する実行時間を表1に示す。なお、SEALライブラリでは、1つの準同型暗号演算をシングルスレッドで実行するように実装されている。このサンプルプログラムでは、AdditionPlainは4回、Additionは3回、それ以外の準同型暗号演算は1回ずつ実行するのを1セットとし、10セット実行した後、各準同型演算1回あたりの平均実行時間を算出する。

表1より、RelinearizationとRescalingの実行時間が長いことが分かる。そのため、CPUの計算資源が余っている場合は、RelinearizationとRescalingのそれぞれを複数のスレッドを用いて並列実行することで実行時間を短縮することができる。なお、RelinearizationやRescalingの前に実行されるMultiplicationPlainやMultiplication, Squareも含め、準同型乗算全体を複数のスレッドで並列実行することで、動的スケジューリングによるオーバーヘッド短縮を狙う。一方、

表2 q_i の数 n_{q_i} と使用スレッド数 $n_{threads}$ に対する Relinearization 1回あたりの平均実行時間[ms]

		$n_{threads}$											
		1	2	3	4	5	6	7	8	9	10	11	12
n_{q_i}	2	33	17	18	18	17	17	18	18	17	17	17	18
	3	65	43	22	22	22	22	22	22	22	22	22	22
	4	101	51	53	27	27	27	27	27	27	27	27	27
	5	151	91	60	60	32	32	31	31	31	32	32	32
	6	203	106	69	71	69	37	37	36	36	37	37	36
	7	271	155	119	79	80	80	41	41	42	42	42	41
	8	349	177	131	89	89	90	88	47	47	46	47	46

準同型加算は、並列実行しても約1[ms]以下の短縮しか期待できないため、本稿の実験では準同型加算をシングルスレッドで実行する。また、Modulus Reductionや暗号文のコピーも、準同型加算と同様に実行時間が数[ms]以下であるため、シングルスレッドで実行する。

4.2 並列化の実装と実行時間測定

OpenMPを用いて、準同型乗算を複数スレッドで並列実行できるように、下記に示す通りにプログラムを変更した。

- MultiplicationPlain, Multiplication, Square : 係数への乗算と加算に関する $O(q_i)$ の for ループを並列化した。したがって、 n_{q_i} (レベル+1) 並列で実行できる。
- Relinearization : iNTT(inverse NTT(Number Theoretic Transform))に関する $O(n_{q_i})$ の for ループと、 K を削減する計算に関する $O(n_{q_i}^2)$ の 2重 for ループの内、依存関係を考慮して内側の for 文のみを並列化した。したがって、 n_{q_i} (レベル+1) 並列で実行できる。
- Rescaling : スケール値への調整に関する $O(K)$ の for ループ (本稿では Multiplication や Square の後に必ず Relinearization するため常に $K=2$) と、NTT と iNTT に関する $O(n_{q_i} \times K)$ の for ループを並列化した。したがって、スケール値調整は K 並列で実行、NTT と iNTT は $n_{q_i} \times K$ ((レベル+1)の K 倍) 並列で実行できる。

そして、4.1項と同じサンプルプログラムを用いて、(レベル+1)の値である n_{q_i} と準同型暗号演算1回分に使用するスレッド数 $n_{threads}$ を変化させた時の Relinearization と Rescaling のそれぞれにかかった実行時間を表2および表3に示す。

表2と図1より、Relinearizationにかかる実行時間は下記の式で近似できることが分かる。ただし、 a, b は定数である。

$$T_{rel}(n_{q_i}, n_{threads}) = \left\lceil \frac{n_{q_i}}{n_{threads}} \right\rceil \times (a \times n_{q_i} + b)$$

上記の近似は下記の2点を示している。

1. スレッド数を増やしていくと、 n_{q_i} が $n_{threads}$ で割り切れる時に実行時間が短縮される。
2. $n_{threads} \geq n_{q_i}$ における最短時間 $T_{rel}(n_{q_i}, n_{q_i})$ は n_{q_i} に線形比例する。

表3 q_i の数 n_{q_i} と使用スレッド数 $n_{threads}$ に対する Rescaling 1回あたりの平均実行時間[ms]

		$n_{threads}$											
		1	2	3	4	5	6	7	8	9	10	11	12
n_{q_i}	2	31	16	16	12	11	11	12	12	12	11	12	12
	3	53	27	22	18	18	14	14	14	14	15	14	14
	4	72	37	30	25	25	21	21	17	17	17	17	17
	5	96	49	39	31	27	27	27	23	23	19	20	19
	6	116	59	46	38	34	30	30	30	30	27	27	22
	7	140	69	54	45	41	37	32	32	33	33	33	29
	8	162	80	64	52	48	43	39	35	35	35	35	35

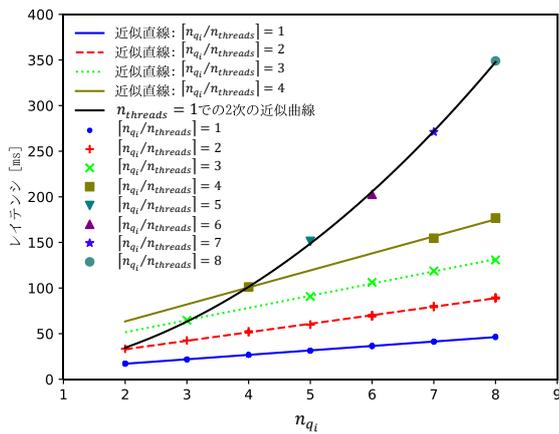


図1 $T_{rel}(n_{qi}, n_{threads})$ に関するグラフ

すなわち、できるだけ n_{qi} の約数のスレッド数で実行することでCPUの計算資源をできるだけ多く使用することになる。また、使用可能なコア数が多い場合では、可能な限り n_{qi} をできるだけ小さくしてからRelinearizationを行うことで実行時間の短縮が見込める。

表3のRescalingの結果について説明する。 $n_{threads} > 1$ において、 $\left\lfloor \frac{2 \times n_{qi}}{n_{threads}} \right\rfloor < \left\lfloor \frac{2 \times n_{qi}}{n_{threads} - 1} \right\rfloor$ を満たす $n_{threads}$ の時に、実行時間が短縮されており、特にスレッド数を1から2に増やすことで実行時間が半分になったことが分かる。このため、CPUのコアが余っている時は、 $K \times n_{qi}$ の約数のスレッド数で実行すると実行時間が短縮されるが、CPUのコアが十分には余っていない場合は K の約数のスレッド数で実行することが効率良いと考えられる。

4.3 CPUに負荷をかけた時の実行時間

メニーコアCPU上で、同時に複数の準同型暗号演算を実行した際の実行時間を示す。4.2項では、1つの準同型暗号演算に対して測定を行った。しかし、実際に準同型暗号を用いたアプリケーションは、同時に複数の準同型暗号演算を実行することもある。したがって、より良いスケジューリングを実施するために、CPUやメモリに負荷がかかる時の実行時間を参考にすべきである。

同時実行する準同型暗号演算の数 $M \in \{2^i | 1 \leq i \leq 8\}$ と $n_{threads} \in \{1, 2, 4, 8\}$ を組み合わせる時に、RelinearizationおよびRescalingのそれぞれ1回分にかかる平均レイテンシを表4および表5に示す。RelinearizationもしくはRescalingを $16 \times M$ 回行うことを1セットとし、10セット繰り返して1回あたりの平均レイテンシを算出した。また、Xeon表4 使用スレッド数 $n_{threads}$ と同時実行する準同型暗号演算数 M に対するRelinearizationでの平均レイテンシ[ms]

		M							
		2	4	8	16	32	64	128	256
$n_{threads}$	1	348	344	346	340	359	435	1,091	2,787
	2	174	179	178	192	224	519	1,388	—
	4	89	91	98	122	256	674	—	—
	8	51	52	69	131	322	—	—	—

Phi7230Fは256論理コアであるため、 $M \times n_{threads} > 256$ となる組み合わせは除外した。

表4より、Relinearizationは同時に64論理コアでの実行が最もスループットが高いことと、使用論理コア数が64の場合は、 $n_{threads} = 2$ での実行時間が短いことが分かる。また、表5より、Rescalingは同時に128論理コアでの実行が最もスループットが高いことと、使用論理コア数が128の場合は $n_{threads} = 2$ の時の実行時間が短いことが分かる。

5. 本研究における最終的に解決したい課題

本研究では、複数の準同型暗号演算を組み合わせることによって構成された関数 f を高速化することを目標とする。具体的には、使用可能な計算資源に応じて、関数 f における準同型暗号演算の実行順序や依存関係を変更し、計算資源をできるだけ活用することで、関数 f 全体のレイテンシを短くすることを目指す。

関数 f において、条件を満たせば、準同型暗号演算の実行順序を変更することができる。例えば、準同型加算や準同型乗算は、平文空間での値を変えることを目的とした準同型暗号演算であるため、数式を適切に変更すれば、実行順序を変えることができる。また、RelinearizationやRescaling、Modulus Reductionなどは、平文空間での値を変更するのではなく、暗号文長やノイズ量を調整することを目的としているため、計算量やノイズ量を適切に管理することができれば、準同型加算や準同型乗算とは独立に実行順序を変更することが可能である。例えば、Modulus Reductionを行うことで n_{qi} の数が1つ減り、準同型乗算を行える回数が少なくなる。一方、Modulus Reduction実行後の各準同型暗号演算に対する実行時間は、表1で示した通りレベル(n_{qi})が小さくなるため短縮される。したがって、関数 f において、依存関係が変更された準同型暗号演算に影響を受ける準同型暗号演算の実行時間やパラメータが変化する。準同型暗号演算の特徴を適切に用いることで、実行時間短縮が期待される。

具体的な例を示す。図2(a)および図2(b)は、ある関数 f の一部を表したフローグラフである。図2(c)は図2(a)と図2(b)で用いる記号の定義である。この2つのフローグラフは、出力暗号文の要求レベルが $x - a \geq 0$ 、複数の準同型暗号演算で構成された「処理A」への入力暗号文の要求レベルが $x + 1$ 、「処理B」への入力暗号文の要求レベルが x である表5 使用スレッド数 $n_{threads}$ と同時実行する準同型暗号演算数 M に対するRescalingでの平均レイテンシ[ms]

		M							
		2	4	8	16	32	64	128	256
$n_{threads}$	1	162	159	160	158	167	202	335	968
	2	81	83	82	89	103	157	436	—
	4	53	53	57	67	94	205	—	—
	8	36	38	48	68	140	—	—	—

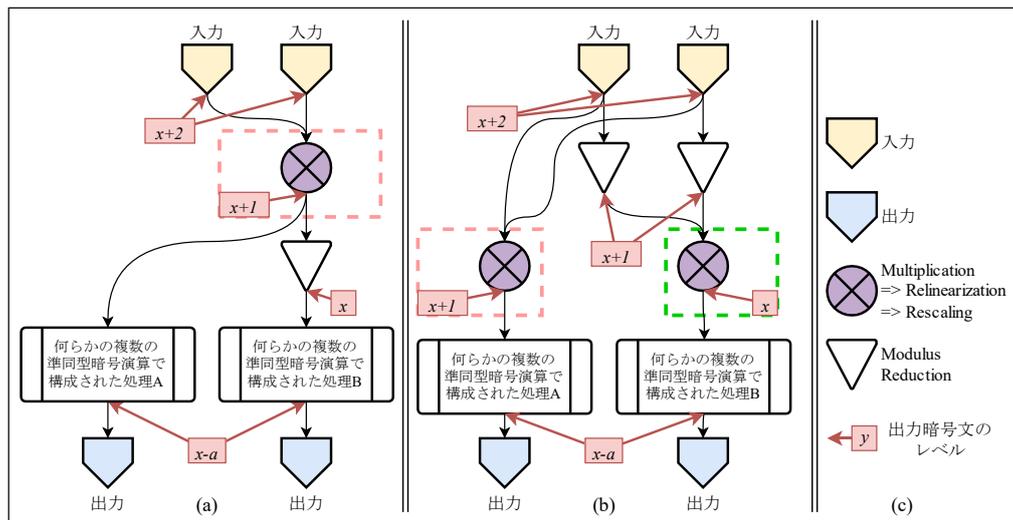


図2 逐次実行時の最適なフローグラフ(a), 計算資源に十分な余裕がある時に実行時間短縮が見込めるフローグラフの一例(b), 各記号の意味(c)

るとする。したがって、このフローグラフへの入力暗号文の要求レベルは $x + 2$ である。そして、「処理 A」にかかる実行時間は、「処理 B」にかかる実行時間よりも短いとする。

図 2(a)で示したフローグラフは、準同型暗号演算を逐次実行する場合に最適なフローグラフである。一方、図 2(b)で示したフローグラフは、CPU やメモリといった計算資源に十分な余裕がある時に、図 2(a)のフローグラフよりも短時間で実行完了できるフローグラフである。具体的なフローグラフの違いは、図 2(b)では、「処理 B」の直前に実行する Modulus Reduction を、準同型乗算の前に実行することである。メモリの観点では、図 2(b)は、Modulus Reduction の有無で 2 種類の暗号文のメモリ領域を確保する必要があるため、メモリ使用量やメモリアクセス回数が増加する。時間計算量の観点では、レベル $l = x$ での準同型乗算が 1 回増加するため、逐次実行する場合は実行時間が長くなる。

図 2(b)が図 2(a)よりも全体を通した実行時間が短くなる場合があることを示す。 $y (= 2y_1 + 1)$ スレッド使用可能な状況下において、 $x + 2 = y_1$ (フローグラフへの入力暗号文は $n_{q_i} = y_1 + 1$ を満たす) であるとする。この状況下では、図 2(a)および図 2 (b)における薄赤色点線で囲われた処理は、4.2 項で示した通り (レベル $l + 1$) の値である $y_1 + 1$ スレッドで実行すると最短実行時間となる。すなわち、 y_1 スレッド分の計算資源が活用できていない。一方、図 2(b)における濃緑色点線で囲われた処理は、入力暗号文のレベルが $x + 1$, すなわち $n_{q_i} = y_1$ であるため、余った y_1 スレッドを用いて実行すると十分高速に実行できる。この場合、図 2(a)における薄赤色点線で囲われた処理と 図 2(b)における左の薄赤色点線にかかる時間は同じである。しかし、右の濃緑色点線で囲われた処理中の Relinearization の時間が $x/(x + 1)$ 倍に短縮される。したがって、短縮した分だけ実行時間の長い「処理 B」を早い時刻に実行開始でき、全体での実行時間は図 2(b)のフローグラフの方が短くなる。

以上をまとめると、計算資源の利用状況に適したフローグラフの変形方法を選択することで、全体での実行時間短縮を達成することができる。そのためには、可能な限り計算資源の利用状況を正確に判定し、効率良く準同型暗号演算を実行していく必要がある。

6. 関連研究

本節では、準同型暗号を用いた時のスケジューリングについての関連研究について述べる。

まず、Bootstrapping は他の準同型暗号演算と比べて数十倍から数万倍の実行時間を必要とするため、Bootstrapping の実行回数を減らすことで全体の実行時間を短縮できる。2013 年に Lepoint らが、Bootstrapping の最適実行位置を決定する問題は NP 完全であると示すとともに、準同型暗号演算を表すフローグラフと特定の初期レベルを元に最小となる Bootstrapping の回数を求める手法を示した[15]。佐藤らは、最近傍法における for ループをアンローリングした上で Bootstrapping の最良位置を決定することで、ナイーブ手法よりも 2.67 倍以上の高速化を達成した[16]。Bootstrapping の他に、Relinearization の回数を減らすことでも実行時間の短縮が可能である。Chen は、Relinearization の最適実行位置を決定する問題が NP 困難であることと、特別なケースでは多項式時間で解くことができることを示した[17]。佐藤らは、Bootstrapping と Relinearization の最適実行位置の厳密解を求める問題において、ナイーブ手法と比べて最適化にかかる時間を 3~4 割削減した[18]。

また、準同型暗号の知識が無い人でも、準同型暗号を用いたアプリケーションを作成できるようにするコンパイラを設計する研究が存在する[19, 20]。これらの研究では、各準同型暗号演算の実行位置や順序を自動で決定する必要がある。Carpov らは、平文空間の法が 2 である論理回路を対象とした準同型暗号演算を対象としており、実行時はリストスケジューリングを用いる[19]。Crockett らは、BGV 方

式における, Relinearization やノイズ量管理のための Modulus Switching の実行位置の決定を自動で行う[20].

7. 実験

5節で示した課題に関連して, CPU の利用状況を元に動的に決定したスレッド数で準同型暗号演算を実行し, 実行時間を短縮することへの検証を実施し, 改善点を考察する.

7.1 対象としたタスクグラフ

本実験では, 演算内容を下記の式で表される e^x のマクローリン展開を対象とした.

$$e^x \approx \sum_{k=0}^M \frac{x^k}{k!}$$

M の値を6とし, 作成したタスクグラフを図3に示す. なお, 本タスクグラフにおけるタスクは, 一連の準同型暗号演算に対して, 下記4種類のまとまりで分けた. また, タスク実行に伴うオーバーヘッドを削減するために, 実行時間が短く, 並列実行の効果が小さい Modulus Reduction と暗号文のコピーは, 任意のタスクの一部に含められるとした.

- Multiplication と Relinearization と Rescaling (図3の暗紫色の⊗)
- Square と Relinearization と Rescaling(図3の赤色の⊗)
- MultiplicationPlain と Rescaling (図3の明緑色の⊗)
- Addition と AdditionPlain(図3の明緑色や暗紫色の⊕)

各タスクの実行順序は, クリティカルパスを考慮し, 優先度付きのリストスケジューリングを用いる. 予めシングルスレッドで計測した各タスクの実行時間を用いて, 各タスク開始から最後のタスクまでのパスの内, 最も実行時間が長い時間をそのタスクの優先度とした. 例えば, 図3におけるタスク番号6の優先度は, タスク番号8, 9, 10の内, 最も優先度の高い値にタスク番号6の実行時間を足した値である. また, ジョブごとのレイテンシを短くするために, ジョブ間では, ジョブ番号が小さいジョブを優先して実行するようにした. 各準同型暗号演算をシングルスレッドで実行する条件下で, このタスクグラフの実行時間は, タスクを逐次実行した場合は4[ms]であり, タスクを動的に並列実行した場合は2[ms]であった.

7.2 実験の流れ

1つの暗号文に対して e^x のマクローリン展開を行うことを1つのジョブとした. 実アプリケーションでは, ネットワークを介してクライアントから計算サーバに入力暗号文データが送られてくる. しかし, 本実験では, 各ジョブにおける入力暗号文データは, 実験開始前に予め生成し, メモリ上に格納しておいた. そして, 各ジョブ開始時に対応する暗号文データを読み出した. これは, ネットワークを介して入力暗号文データがクライアントから計算サーバに送られてくるプロトコルを簡略化したものである.

各入力データに対するジョブの開始時刻の決定方法を

説明する. まず平均到着間隔 $D[\text{ms}] \in \{60, 70, 80, 90, 100, 120\}$ を決める. そして, 決定した D を用いて, 指数分布の累積分布関数の逆関数 $d_i = -D \ln(1 - x_i)$ を作成した. そして, 2,300個の一樣乱数 $\{x_i \in [0, 1] \mid 1 \leq i \leq 2300\}$ を生成し, 各入力データの到着間隔 $\{d_i \mid 1 \leq i \leq 2300\}$ を求めた. 最後に, $T_i = \sum_{j=1}^i d_j$ を用いて, i 番目の入力データの到着時刻, すなわち実験における i 番目のジョブの開始時刻 T_i を決定した. なお, ジョブ番号が小さいほどジョブ開始時刻が早い.

実験開始直後と実験終了直後は実行中のジョブ数が少なく, 設定したジョブの開始時刻の間隔と異なる負荷となる. そのため, 評価対象のジョブは, 301番目から1,900番目までの入力に対するジョブとした. 評価指標は, 各ジョブについて, ジョブを開始した時刻から完了するまでのレイテンシを対象とした. また, 評価対象の最初のジョブの開始から最後のジョブの完了までの時間も評価対象とした. 各平均到着間隔について5回実行した平均値を取得した.

7.3 実験に用いた手法

本実験では, 静的に使用するスレッドを設定しておくナイーブ手法と, CPU の利用状況を考慮して使用するスレッド数を決定する動的手法の2種類を試した. なお, 4.3項の結果より, 全体で128スレッドまで使用可能とし, ジョブ開始処理やスケジューリング用に1つの物理コア使い, 126論理コアを準同型暗号演算に使用可能とした. いずれの手法でも, 実行中の全てのジョブが使用しているスレッド数

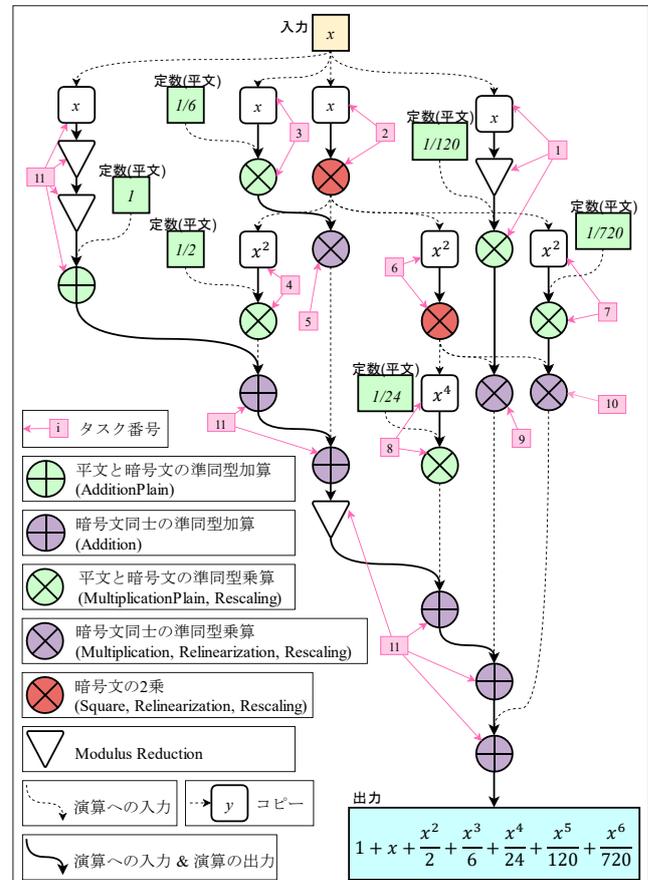


図3 使用した e^x のマクローリン展開のタスクグラフ

の総和が 126 スレッドを超えないようにした。

タスクが 1 つしか存在しない場合には、できるだけ多くの論理コアを用いて 1 つのタスク内における処理を並列実行することで、そのタスクにかかるレイテンシを短くすることができる。しかし、複数の実行可能なタスクが存在する場合は、1 つのタスクに多くの論理コアを割り当ててタスクを 1 つずつ逐次実行するよりも、複数のタスクにそれぞれ少ない論理コアを割り当てて、各タスクを並列に実行する方が計算資源を余らせにくい。

以上より、準同型加算に関するタスクには 1 スレッド割り当てるとし、準同型乗算に関するタスクに割り当てるスレッド数は $n_t \in \{1, 2, \lfloor n_{q_i}/2 \rfloor, n_{q_i}\}$ とした。ただし、 n_{q_i} はリストスケジューリングのキューの先頭にあるタスクの入力暗号文の (レベル+1) の値 (q_i の数) である。4.2 項で示した通り、計算資源が余っている状況下では、Relinearization は (レベル+1) の値である n_{q_i} スレッドで実行すると実行時間が最短になるためである。

7.3.1 ナイーブ手法

ナイーブ手法では、予めタスクごとに使用するスレッド数を $n_t \in \{1, \lfloor n_{q_i}/2 \rfloor, n_{q_i}\}$ と決めておき、受け取った入力データに対するジョブを実行する。

7.3.2 動的な手法

動的な手法では、CPU の利用状況に応じて、タスクごとに使用スレッド数を変化させる。あるタスクを実行する際に、使用するスレッド数を予め用意しておいた候補群の中から詮索する。この選択方法を以下に示す。まず、実行中の全タスクで使用しているスレッド数の総和を n_c とする。次に、リストスケジューリングのキューに含まれている、他の実行可能かつ実行待ちのタスクの数を r とする。また、他の各実行可能タスクについて、準同型加算および平文と暗号文の準同型乗算は 1 スレッド、暗号文同士の準同型乗算 (2 乗を含む) は 2 スレッドで実行するとみなし、総和を n_r とする。これは、リストスケジューリングのキューの先頭にあるタスクとキューに含まれる 2 番目以降の実行可能タスクとの間での計算資源の分配バランスを適切にすることを目的としている。そして、下記の順番で条件判定し、対象タスクに割り当てるスレッド数 n_t を決定する。なお、いずれかの条件を満たすまでタスクを実行しない。

1. $n_c + n_{q_i} + n_r \leq 126$ を満たす場合: $n_t = n_{q_i}$
2. $n_c + \lfloor n_{q_i}/2 \rfloor + r \leq 126$ を満たす場合: $n_t = \lfloor n_{q_i}/2 \rfloor$
3. $n_c + 2 + r \leq 126$ を満たす場合: $n_t = 2$
4. $n_c + 1 \leq 126$ を満たす場合: $n_t = 1$

8. 実験結果・考察

各ジョブのレイテンシについて、スレッド数を 1 としたナイーブ手法を図 4 に、スレッド数を $\lfloor n_{q_i}/2 \rfloor$ とナイーブ手法を図 5 に、スレッド数を n_{q_i} としたナイーブ手法を図 6 にそれぞれ示す。動的な手法での各ジョブのレイテンシを図 7

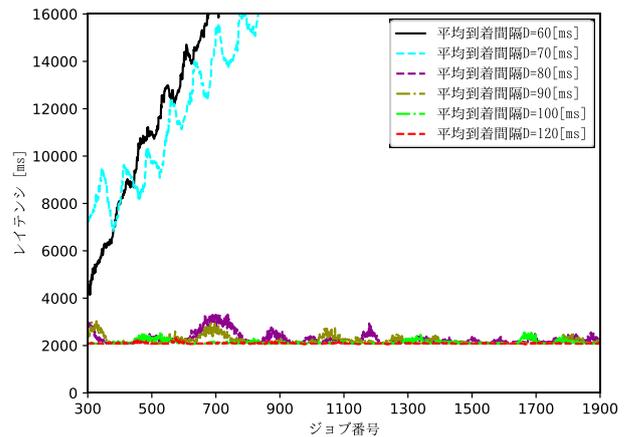


図 4 ナイーブ手法 (1 スレッド) でのレイテンシ

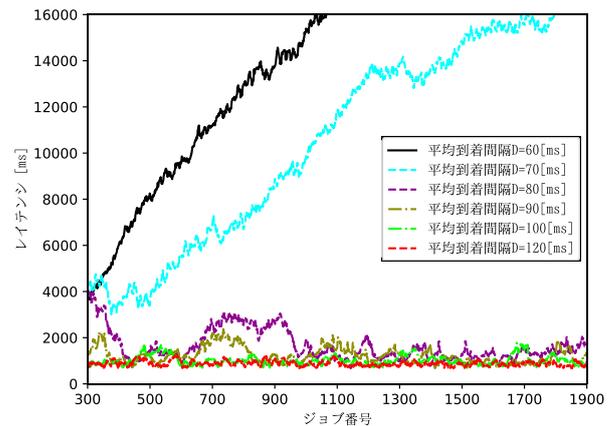


図 5 ナイーブ手法 ($\lfloor n_{q_i}/2 \rfloor$ スレッド) でのレイテンシ

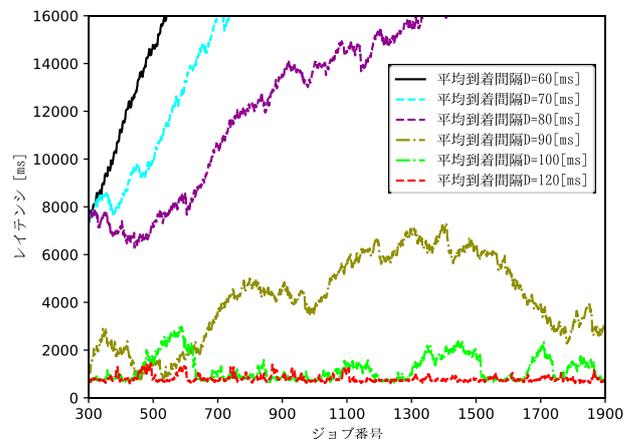


図 6 ナイーブ手法 (n_{q_i} スレッド) でのレイテンシ

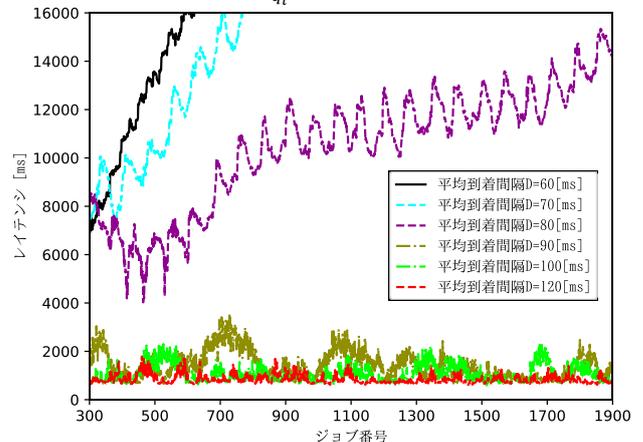


図 7 動的な手法でのレイテンシ

表 6 最初のジョブの開始から最後のジョブの完了までの時間

手法	平均到着時間 D [ms]に対する時間[ms]					
	$D = 60$ [ms]	$D = 70$ [ms]	$D = 80$ [ms]	$D = 90$ [ms]	$D = 100$ [ms]	$D = 120$ [ms]
ナイーブ手法 (1 スレッド)	143,354	146,066	134,036	146,159	161,897	196,799
ナイーブ手法 ($[n_{q_i}/2]$ スレッド)	126,634	126,979	133,520	145,248	160,605	195,600
ナイーブ手法 (n_{q_i} スレッド)	153,122	153,039	153,146	146,915	160,532	195,467
動的手法	145,761	147,913	146,059	145,560	160,513	195,519

に示す。いずれも横軸がジョブ番号であり、縦軸が各ジョブのレイテンシである。301 番目のジョブを開始してから 1,900 番目のジョブを完了するまでの時間を表 6 に示す。

平均到着間隔 D が 70[ms] 以下の場合には、単位時間あたりにおける開始したジョブ数が実行完了したジョブ数を上回るため、レイテンシが増加傾向にある。なお、ナイーブ手法 ($[n_{q_i}/2]$ スレッド) がナイーブ手法 (1 スレッド) よりもレイテンシの増加率が低い理由として、ナイーブ手法 ($[n_{q_i}/2]$ スレッド) の方が 1 つのジョブの完了時間が短く、実行中のジョブ数の増加が緩やかであるためと考えられる。

$D \geq 100$ [ms] では、実行完了したジョブ数の方が開始したジョブ数よりも多いため、ナイーブ手法 (1 スレッド) では 2,000[ms] 程度、その他の手法では 1,000[ms] 程度でレイテンシは安定している。 $D \in \{80,90\}$ においては、ナイーブ手法 (1 スレッド) では、レイテンシが 2,000[ms] 程度であるジョブが多い。ナイーブ手法 ($[n_{q_i}/2]$ スレッド) も同様に、レイテンシが長くなることもあるものの、1,000~2,000[ms] の範囲内であるジョブが多い。しかし、ナイーブ手法 (n_{q_i} スレッド) では、 $D = 80$ [ms] ではレイテンシが増加傾向にあり、 $D = 90$ [ms] ではレイテンシが 1,000~7,000[ms] と安定していない。動的手法は、 $\bar{T} = 80$ [ms] ではレイテンシが増加傾向にある一方で、 $\bar{T} = 90$ [ms] ではレイテンシが 1,000~4,000[ms] の範囲内に収まっている。

表 6 について述べる。 $D \geq 90$ [ms] の場合における差が生じた理由は、単位時間あたりに開始されたジョブ数が実行完了したジョブ数より少なく、1 つの準同型演算に多数のスレッドを割り当てる方が末尾のジョブの完了時間が短いためである。一方、 $D \leq 80$ [ms] の場合は、ナイーブ手法 ($[n_{q_i}/2]$ スレッド) が最も短く、他の手法と比べて最大 26 秒の差が存在する。したがって、ナイーブ手法 ($[n_{q_i}/2]$ スレッド) が最良であるという結果が得られた。

動的手法が最良ではなかった原因について考察する。まず、ナイーブ手法 (n_{q_i} スレッド) は、 \bar{T} が小さいほどレイテンシが長くなるため、 D が小さい場合は、少ないスレッドで実行するのが良いことが分かる。一方で、ナイーブ手法 (1 スレッド) とナイーブ手法 ($[n_{q_i}/2]$ スレッド) との比較の通り、実行中のジョブ数が増えるとレイテンシが増えてしまうため、計算資源に余裕がある内にはできるだけ多くのジョブを完了する必要がある。したがって、動的手法が最良ではなかった原因は、使用スレッド数を決定する基準

が適切ではなかったためと考えられる。具体的には、実行可能タスクに割り当てるために確保する計算資源の見積もりが実際よりも過小もしくは過多であったと考えられる。確保すべき計算資源を適切に見積もるために必要な要因の一つとして、タスク開始におけるオーバーヘッドを見積もりの計算に含めることが挙げられる。タスク開始のオーバーヘッドによって、タスクを開始する時刻が予想よりも遅れることで、計算資源に余裕が生じる場合があるためである。特に、メニーコア CPU は動作周波数が低いため、動作周波数の高い CPU と比べてオーバーヘッドが大きくなりやすいため、見積もりへの影響が十分に存在すると考えられる。また、準同型暗号演算の種類を元に見積もったが、暗号文のレベルを考慮していない。準同型暗号演算の実行時間はレベルに依存するため、レベルを考慮することで動的手法を改善することができると考えられる。

9. まとめ

本稿では、メニーコア CPU における準同型暗号演算を高速化するためのタスクスケジューリングを対象に、使用するスレッド数を動的に変更する手法に対して実験を行った。実験で用いた動的手法は最良ではないという結果が得られ、最良ではなかった原因について考察した。

本研究において最終的に解決したい課題に関して、実験結果から得られた今後の課題を示す。まず、使用するスレッド数の決定基準を改良することが挙げられる。例えば、実行可能タスクのために確保する計算資源の見積もりを改善することである。ただし、使用スレッド数の決定基準を複雑にすることで、動的スケジューリングによるオーバーヘッドが大きくなってしまふ。そのため、できるだけ動的スケジューリングの負荷を軽くすることも今後の課題である。

謝辞

本研究は、JST、CREST、JPMJCR1503 の支援を受けたものである。

参考文献

- [1] Paillier, Pascal: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes, Proc. EUROCRYPT 1999, LNCS, vol 1592, pp.223-238, Springer (1999).
- [2] Gentry, C.: Fully Homomorphic Encryption using Ideal Lattices, Proc. STOC 2009, pp.169-178, ACM (2009).
- [3] Lyubashevsky, V., Peikert, C. and Regev, O.: On Ideal Lattices and

- Learning with Errors over Rings, Proc. EUROCRYPT 2010, Vol.6110, pp.1-23, Springer (2010).
- [4] Cheon, H. J., Han, K., Kim, A., Kim, M. and Song, Y.: Bootstrapping for Approximate Homomorphic Encryption, Proc. EUROCRYPT 2018, LNCS, Vol.10820, pp.360-384, Springer (2018).
- [5] Brakerski, Z., Gentry, C. and Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping, Proc. ITCS 2012, pp.309-325, ACM (2012).
- [6] Gentry, C., Halevi, S. and Smart, P. N.: Homomorphic Evaluation of the AES Circuit, Proc. CRYPTO 2012, LNCS, Vol.7417, pp.850-867, Springer (2012)
- [7] Brakerski, Z.: Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP, Proc. CRYPTO 2012, LNCS, Vol.7417, pp.868-886, Springer (2012).
- [8] Fan, J. and Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption, IACR Cryptology ePrint Archive, Report 2012/144, 入手先<<https://eprint.iacr.org/2012/144>> (2012).
- [9] Bajard, C. J., Eynard, J. and Hasan, A.M.: A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes, Proc. ASC 2016, LNCS, Vol.10532, pp.423-442, Springer (2016).
- [10] Cheon, H. J., Kim, A., Kim, M. and Song, Y.: Homomorphic Encryption for Arithmetic of Approximate Numbers, Proc. ASIACRYPT 2017, LNCS, Vol.10624, pp.409-437, Springer (2017).
- [11] Cheon, H. J., Han, K., Kim, A., Kim, M., and Song, Y.: A Full RNS Variant of Approximate Homomorphic Encryption, Proc. SAC 2018, LNCS, vol.11349, pp.347-368, Springer (2018).
- [12] Smart, P. N. and Vercauteren, F.: Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes, Proc. PKC 2010, LNCS, Vol.6056, pp.420-430, Springer (2010).
- [13] Smart, P. N and Vercauteren, F.: Fully homomorphic SIMD operations, Designs, codes and cryptography, Vol.71, No.1, pp.57-81 (2014).
- [14] Microsoft: Microsoft SEAL (release 3.2), GitHub (online), 入手先<<https://github.com/Microsoft/SEAL>>, (参照 2019-03-12) .
- [15] Lepoint, T. and Paillier, P.: On the Minimal Number of Bootstrappings in Homomorphic Circuits, Proc. FC 2013. LNCS, vol.7862. pp.189-200, Springer (2013).
- [16] 佐藤宏樹, 馬屋原昂, 石巻優, 山名早人: 完全準同型暗号による秘密計算回路のループ最適化と最近傍法への適用, DEIM 2017, H6-3 (2017).
- [17] Chen, H.: Optimizing relinearization in circuits for homomorphic encryption, arXiv, 1711.06319, pp.1-10, 入手先 <<https://arxiv.org/abs/1711.06319>> (2017).
- [18] 佐藤宏樹, 山名早人: 完全準同型暗号における bootstrap problem 及び relinearize problem の厳密解法の高速度化, DEIM 2019, I5-4 (2019).
- [19] Carpv, S., Dubrulle, P. and Sirdey, R.: Armadillo: A Compilation Chain for Privacy Preserving Applications. Proc. SCC 2015, pp.13-19, ACM (2015).
- [20] Crockett, E., Peikert, C. and Sharp, C.: ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY. Proc. CCS 2018. pp.1020-1037, ACM (2018).