完全準同型暗号を用いたゲノム秘匿情報検索の SSD 適用に向けた調査

辻　有紗†　　圓戸　辰郎††　　小口　正人†

† お茶の水女子大学　〒 112–8610 東京都文京区大塚 2–1–1
†† キオクシア株式会社　〒 108–0023 東京都港区芝浦 3–1–21 田町ステーションタワー S
E-mail: †{arisa-t,oguchi}@ogl.is.ocha.ac.jp, ††tatsuro1.endo@kioxia.com

あらまし　近年クラウド上でのビックデータ解析に向けて完全準同型暗号 (以下：FHE) を用いた秘密計算技術の実用化に向けた研究が盛んに行われている．FHE の課題として，bootstrap 処理などにより時間空間計算量が膨れ上がることが挙げられる．空間計算量については, DRAM を多く使用するためクラウドが活用されるが，1 台のリソースを分け合うことにより使用効率は約 65 %に止まる. また, DRAM の特性上, 1bit あたりのコストが高く, CPU から DRAM にデータの load/store を行う際，直列にしか命令を実行できない．そこで, 本研究では, ゲノム秘匿検索アプリケーションを対象として, DRAM と異なる特性を持つ SSD を併用することを提案する. 初めに, 処理の傾向を 調べ, cpu 使用率が高い箇所の ボトルネックは DRAM の load/store であることを確認した．また, SSD の並列 IO 処理性能を引き出すため, 適切な並列数の検討や並列処理時の CPU コストの変化を考察する．最後に, DRAM 使用時と SSD 使用時の CPU コストの変化を調べ, SSD 活用の可能性について考察した.

キーワード　完全準同型暗号, bootstrap, ゲノム秘匿情報検索, SSD, 並列処理

1　はじめに

近年多くの分野の産業が IT 化しデータを収集するようになったことで，それらの機密性の高いデータを有効活用し新たな知見やシステムを生み出すことが期待されている．しかし，これらのデータはサイズが大きく，個人で保有し分析することが難しいことが多いため, データの情報の安全性を保証したクラウド上でのビックデータ解析の実現が必要である．ここで, 従来の暗号技術では, 機密性の高いデータを暗号化してクラウド上へ保存することはできるが, 演算処理を行う際にはデータを復号化しなければならず, サーバからの漏洩や不正利用のリスクを避けることはできない．そこで, 現在は完全準同型暗号 (FHE) を用いて暗号化したままの状態で計算を実行する秘密計算が提案されている.

FHE の課題として, 時間空間計算量が膨れ上がることが挙げられるが, その中でもボトルネックは bootstrap [1] と呼ばれる処理である．bootstrap とは暗号文同士を演算する際に蓄積するノイズを取り除くため複数の鍵を用いて暗号化・復号化を行う処理を指し, 実行には膨大なメモリ容量を要する．FHE では平文にノイズを加えることで暗号化を行っているが, ノイズがパラメータ level で設定される閾値を超えると復号できなくなる．そのため, 任意回暗号文を演算できる FHE を実現するには bootstrap 処理が必須となる．先行研究 [2] では, ゲノム秘匿検索 [3] において, bootstrap を用いず, 検索クエリを 1 文字ずつ暗号化した上で送信し, 返ってきた結果を利用して次の文字に対する送受信を行う方法も模索されている．bootstrap は準同型加算・乗算に対し, 数百〜数千倍の時間を要するため, 一文字ずつ処理した方が高速である．一方, 検索クエリのサイズが大きい場合, server と client 間の通信量が大きくなるという問題がある.

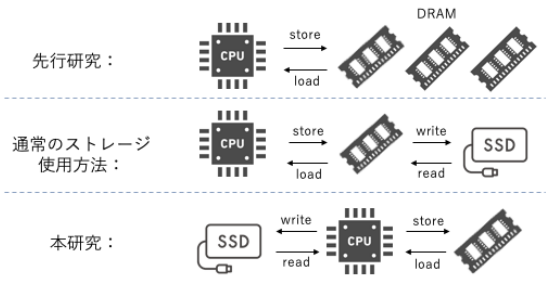

図 1　DRAM・ストレージの利用方法の比較

また, FHE は従来, 空間計算量が大きいためクラウドで活用される手法だが, VM や Container の利用によって 1 台のリソースを分け合うため, 適切な利用がされず使用効率が低い [4]. その上, DRAM1bit あたりの値段は SSD と大きく異なり, コストが高い．ゲノム秘匿情報検索について, サーバ側での演算に対してマスタ・ワーカ型の分散処理も検討されている [5]. HPC(High Performance Computing) のような, コストの優先度が低い環境で実行する場合は, 大幅な高速化が可能となる．しかしながら, FHE の活用が期待されるセキュリティ問題は多岐に渡り, その一部では, サーバの分散や大量の DRAM を消費することによるコスト増大の懸念が生じると想定される.

そこで, DRAM とは異なるアクセス特性を持つ SSD を活用することを検討する. 先行研究 [6] により, swap 先に SSD を使用することで高速化されることが示されている．本研究では,

CPU の処理負荷を調べ, SSD を活用することで実行にかかるコストを削減し, 高速化する方法を検討する.

2 完全準同型暗号

2.1 特　　徴

準同型暗号とは, データを暗号化した状態で計算し, 完了後に復号することで, 暗号化せずに計算した場合と同じ結果が得られる暗号方式のことである. (1) を加法準同型暗号, (2) を乗法準同型暗号という.

$$Encrypt(m) \oplus Encrypt(n) = Encrypt(m+n) \ ... \ (1)$$
$$Encrypt(m) \otimes Encrypt(n) = Encrypt(m \times n) \ ... \ (2)$$

加法準同型暗号の代表的なものに, 合成数剰余判定仮定を用いた Paillier 暗号 [7], 乗法準同型暗号の代表的なものに, 大きい値の素因数分解の困難性を用いた RSA 暗号 [8] などがあり, これらは古くから知られている. 一方, (1),(2) 共に満たす暗号方式は, 実行可能な演算回数に応じて, SHE(Somewhat Homorphic Encryption), LHE(Leveled Homomorphic Encryption), FHE(完全準同型暗号, Fully Homomorphic Encryption) に分けられる [9] [10] [1]. SHE, FHE は 2009 年, Graig Gentry によって提案された.

SHE は乗算回数を比較的小さい定数回に制限された暗号方式である. Gentry はイデアル格子に基づき, 暗号化する際に小さいノイズを加えることで実現した. その後, RLWE 問題に基づくものや GCD 問題に基づくものなど, 様々な方法が提案されている.

FHE は任意回, 加算・乗算可能な暗号方式であり, SHE と bootstrap を元に構成される. SHE を用いて一定回数以上演算を行う場合, 暗号文のノイズ量が閾値を超え, 正しく復号できなくなる. ここで, bootstrap と呼ばれる, 暗号文に蓄積するノイズを取り除くため複数の鍵を用いて暗号化・復号化を行う処理を行うことで, 任意回の暗号文同士の演算が可能となる. ただし, bootstrap には膨大な時間空間計算量が必要であるため, 実用に至っていない.

SHE と FHE の中間にあたる, 事前に決めた任意回の乗算を可能とする LHE も, Brakskil, Gentry, Vercauteren により提案されている. モジュラススイッチ・鍵スイッチという仕組みにより構成され, 乗算によるノイズや暗号文サイズの爆発的な増加を防いでいる. 現在は暗号可能関数が比較的大きいことや, 時間空間計算量の面から, LHE が利用される場面が多い.

2.2 bootstrap

暗号文に蓄積したノイズを取り除くには, 複合可能な状態で一度暗号文を復号し, 復号結果を再び暗号化して, 暗号文演算処理を続ければ良い. しかし, この場合平文を生成することになり, データの安全性が保障されない. そこで, 完全準同型暗号の, データを暗号化した状態で処理できる性質 (2.1 節 (1), (2)) を用いる.

平文を m, 鍵ペア (pk, sk) に対応する暗号文を c, 鍵ペア (pk', sk') に対応する暗号文を c' とすると, (3) が成り立つ.

$$Encrypt(m, pk') = Decrypt(Encrypt(c, pk'), Encrypt(sk, pk')) \ ... \ (3)$$

暗号文 c, 秘密鍵 sk それぞれについて, pk′で暗号化した状態で, 復号処理をすることにより, 暗号文に蓄積したノイズの除去が行われる.ただし, 暗号文c, 秘密鍵skは pk'で暗号化されているため, 平文も pk'で暗号化されている.また, KDM 安全性を満たす場合,sk = sk′, $pk = pk'$ としてよく, 複数の鍵を用意する必要はない. 上の処理を, 複数回の準同型演算により暗号文のノイズが上限に近づくたびに行い, FHE を実現する.

2.3 ライブラリ

完全準同型暗号を実装するライブラリは多数存在し, それぞれ暗号方式や実行可能な処理が異なる.

Helib [11] [12] は RLWE 問題に基づく SVG 方式 [13] で, 平文を整数環とする. bootstrapping などの基本的な暗号処理に加えてパッキング, rotation なども備えており, よく用いられるライブラリの 1 つである. 一方, Palisade [14] は RLWE 問題に基づく CKKS 方式 [15] も備えており, 近似的に固定小数点数を扱うことができるため機械学習アプリケーションなどで用いられる. bootstrap はサポートされていない. 最近では, TFHE [16] [17] という LWE 問題に基づくライブラリも公開されている. 1bit ごとに暗号化・準同型演算を行い, bootstrap を高速に実行できる. 論理回路を評価することに向いている一方で, パッキングなどが利用できないため, 多数の入力に対する算術演算には向いていない.

そういった特徴を踏まえ, 本研究では Helib を用いた.

3 ストレージデバイス

3.1 SCM(Storage Class Memory)

近年, 半導体の小型化・低価格化・高性能化により CPU・DRAM の高性能化が進められているが, 現在は物理的な問題により半導体の更なる小型化が難航し, 成長率が下がっている. 一方, SSD は, 研究が進めれられている段階で, 今後の成長が期待される. 3DNAND フラッシュメモリでは, 複数 bit 保存できるセルを水平に並べた層を垂直に重ねる. 各層のセル数・各セルの bit 数の増加が難しい場合も, 集層数を上げることで, 大容量化が可能である. 一方, Intel と Micron が共同開発した 3D XPoint では, クロスポイント構造の平面上のメモリセルアレイを垂直に並べる. クロスポイント構造は, アクセスラインと呼ばれる信号線が直行するように配置されており, アクセスラインがクロスする点 (メモリセル) に 1bit のデータを記憶する. そのため, NAND フラッシュメモリのようなブロック消去やウェアレベリングは必要ない. NAND の 1000 倍の速さで読み書きが可能とされており, SCM(Storage Class Memory) としての活用も期待される. SCM は, メモリとストレージの性能差を埋める役割を担い, 読み書きが高速, 低電力消費, 不揮発性などの特徴を持つ.

一方, SSD の高速な読み書き性能を発揮するためにはインタフェースの性能向上も必要となる. 従来はコントローラと SSD 間を HDD 用に設計された SATA などで接続していたが, SSD 用の PCI express で接続し, NVMe プロトコルでアクセスすることで高速化される. また, NVMe を使用すると並列処理が可能となるため大幅なスループットの上昇が見込める. 最近では NVMe の発展として, サーバからストレージまでのアクセスを NVMe で接続する NVMe-oF というプロトコルも使用されている. 以上のプロトコルやインターフェースを用いると, SSD と CPU の間で並列 IO 処理が可能である.

3.2 DRAM と SSD の比較

表 1 DRAM, SSD の特性の違い

特性	DRAM	SSD
アクセス速度	50ns	1ms
並列 IO	×	○
1GB あたりの費用 [18]	35 円	0.72 円

表 1 に SSD と DRAM の特性の違いを示す. CPU から DRAM にデータの load/store を行う際，直列にしか命令を実行できない. 一方, CPU から SSD にデータの read/write を行う際，並列 IO が可能で, 理想的には SSD コントローラに設定されている queue depth という数値の分だけ命令を同時に実行することが可能である.

また, DRAM ではプログラムの並列数が 128 程度で最大のスループットになるのに対して, SSD は並列数 32 程度で最大に達する. こちらは, intel optane SSD の特徴の 1 つで, 一般的な CPU コア数が多くない環境で最適とされる, 小さな並列数で最大のスループットが出る [19].

4 先行研究

4.1 ゲノム秘匿情報検索

アプリケーションについて述べる. クライアントサーバ 型通信で, クライアントがサーバのゲノムデータベース (PBWT) [20] に対して特定の塩基配列が最長マッチを持つか問い合わせを行う. 具体的な手順を以下に示す.

（1） クライアントはクエリを全文暗号化し，その他のパラメータと一緒にサーバに送信する.

（2） サーバはクエリを元に FHE 演算を行い，結果をクライアントに送信する.

（3） クライアントはサーバから送られてきたデータを復号し, 結果を得る.

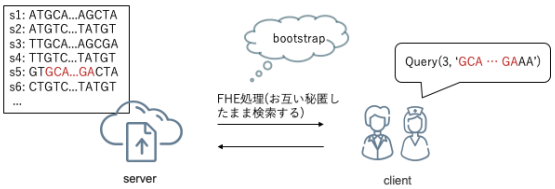

図 2 ゲノム秘匿情報検索の流れ

この時, お互いに持っているゲノム塩基配列や調べたい内容を秘匿しながら行うことが FHE により実現され, 個人情報の漏洩が防がれている. ここで, クエリ長や検索ポジション数に関わらず実装を可能にするには, FHE の演算で bootstrap の導入もしくは LHE でより多くの演算を可能にするための大きなパラメータの指定が必要となる. また, これらの暗号化処理には膨大なメモリ容量が必要な一方で, クラウドでは適当に DRAM が割り当てられないことが多く, スワップ処理が発生する場合がある. [4] スワップ発生時の実行状態の変化について, 6.6 章で検討する.

4.2 LHE と FHE の比較

先行研究では, ゲノム秘匿検索において, ノイズ量の閾値を表すパラメータ level を大きい値にし bootstrap が起こらないように設定した場合と, level を一定の小さな値にし, bootstrap を行う場合で実行時間の比較が行われている. 検索クエリ長に比例して bootstrap の回数が増加し, また, bootstrap は準同型加算・乗算の数百〜数千倍時間がかかる. そのため, クエリ長が伸びるほど bootstrap 処理のない LHE として実行した方が高速になることが示された.

しかし, LHE を用いて level を大きく設定した場合, 暗号文サイズが大きくなり, メモリ使用量や通信量など, client に負荷がかかる. また, 暗号処理の重さが事前にわからず適切な level を設定できない場合もある. 以上から, bootstrap の高速化が求められる.

5 評価方法

FHE の膨大なメモリ使用量に対して, クラウド上の DRAM で対応した場合, クラウドのリソースの分散による使用効率の低下や DRAM の 1bit あたりのコストが高いことが問題となる. 一方, SCM は DRAM に比べコストが低く, クラウド上でのリソースの分散も発生しない. そこで, アプリケーション実行時の CPU の状態を計測し, DRAM とは異なるアクセス特性を持つ SCM を活用する方法を検討する.

5.1 評価環境

表 2 に評価環境を示す. ゲノムデータベースについては 1 サンプルあたり 10000 文字のデータを 2184 サンプル用意した.

表 2 評価環境

CPU	Xeon E5-2643 v3
	(3.40 GHz × 6 Cores) × 2
L1 (i,d)cache	32KB, 64B/line
L2 (i,d)cache	256KB, 64B/line
L3 cache	20480KB, 64B/line
DRAM	DDR4, 512GB
SSD	Intel Optane SSD 800P
	118GB, 3D XPoint
Docker1	memory 512GB
Docker2	memory 1GB, swap memory 9GB

また, プログラム中で bootstrap 処理の割合を大きくすることを目的に, 検索クエリの長さは 1, データベース上の検索開始ポジション数は 1 に設定した. 6.1〜6.4 章の評価は Docker1, 6.5 章の評価は Docker2 のコンテナ上で行った.

6 評　　価

6.1 処理の流れ

大きく分けて 3 つの処理, initialization, lookup, bootstrap に分けられる.

initialization は (コネクション確立やファイル転送にかかる時間を除いた) 検索するための準備にかかる時間を指す. クライアントからサーバへ, 検索クエリの長さ, ダミーも含めた検索開始ポジション, 公開鍵の情報, FHE コンテキストなどを送信する. サーバは, クライアントから受け取った公開鍵の情報や FHE コンテキストを元に公開鍵の作成する. また, PBWT をマッチ数の検索がしやすいベクトル LUT(Look-up table) へ変形し, 検索環境を整える.

lookup は bootstrap を除く検索の主要な処理にかかる時間を指す. サーバでは, ゲノム配列の一文字ずつ検索するたびに変わるテーブルの更新や, 暗号化された検索配列とデータベースの照合を行う. クライアントは待ち状態で作業は行わない.

bootstrap は Helib の recrypt 関数を用いて行われる bootstrap 処理を指す. NTL ライブラリ [21] による並列化が行われており, 並列度が高い.

6.2 並列化可能性の調査

bootstrap 処理の並列度が高いことから, 処理全体について並列度の調査を行った. lookup の照合処理も並列化されていたため, 残りの initialization と lookup の一部について独立性の高い計算部分を OpenMP を用いてマルチスレッド化し, 終始一定数のスレッドが存在する状態で, 計測を行った. 図 3 に実行時間の推移と速度向上率を示す. ゲノム秘匿検索については論理 CPU 数 12 まで並列数に伴い減少し, その後は 増加に転じる. 並列化による CPU コストが変化し, オーバーヘッドが大きいことが原因である. また, 処理の一部は依存性が高く, 逐次処理となるため, 速度向上効率は一部の非効率部分により律速

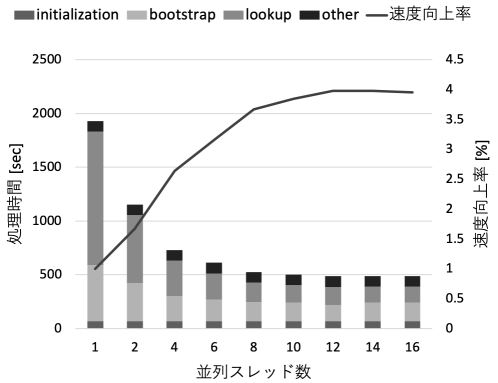

図 3　並列化による速度向上率

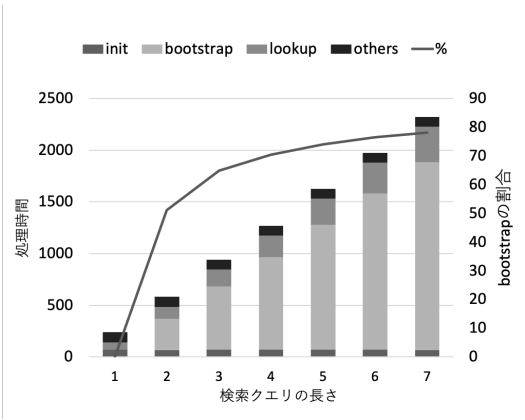

図 4　クエリ長の変化に対する処理内訳の推移

される. 今回は initialization の並列度が低く, 設定スレッド数の増加により余分なタスクの切り替えが発生し, 実行時間が増加する. 一方, bootstrap と lookup の並列度は非常に高く, 設定スレッド数の増加により高速化される. ここで, 実際は検索クエリ長やポジション数は大きい値の場合が多く, FHE 処理特に bootstrap 処理の割合が大きくなる. 図 4 に, 検索クエリ長の変化における bootstrap 処理の割合の増加を示す. したがって, プログラム内で並列処理が可能な割合は大きく, CPU のマルチコア性能の活用が可能である.

6.3 アプリケーションの傾向

図 5 に実行時間に対する CPU 処理の内訳を示す. initialization では, server は client による FHE コンテキストや公開鍵の作成を待機する時間が長いため, idle 値が高く推移する. その後, disk から公開鍵や暗号文を読み込むため, iowait も発生している. 続いて bootstrap は前半では処理の並列度が低く, CPU の最大の性能を使用していない. 後半は暗号化・復号化処理により引き起こされる CPU の計算処理が重く, 並列度が高いため, usr 値が高い. bootstrap の内部の処理内容について今後調査を行う必要ある. lookup もゲノム配列データベースに対応するテーブルの更新や暗号化されたクエリとの照合による CPU の計算処理が重く, 並列度が高いため, 終始 usr 値が高く推移している.

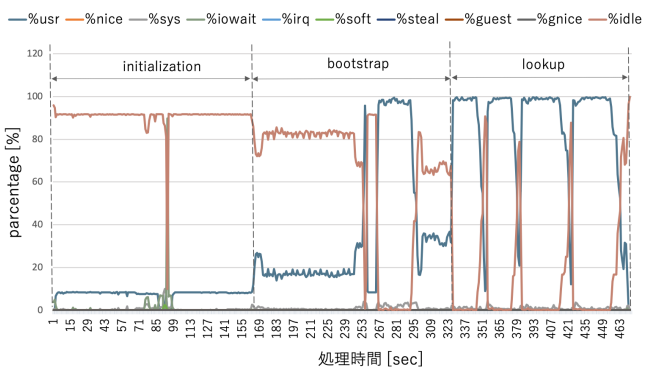

図 5　CPU 処理内訳

表 3　hyperthead 有効時・無効時の比較

	無効時 (12)	有効時 (12)	有効時 (24)
総実行時間 (s)	477.33	605.67	623.67
initialization(s)	69.81	95.20	94.40
lookup(s)	144.17	155.43	166.41
bootstrap(s)	164.96	223.12	230.30
IPC	2.56	2.49	2.19
context switch	163,206	294,530	540,457
cpu migration	10,655	11,760	33,077
iTLB load 数	35,309,006	66,520,758	104,796,253
iTLB load misses(%)	207.75	153.90	115.66
L3 cache load 数	5.77×10^{10}	6.04×10^{10}	6.29×10^{10}
L3 cache load misses(%)	8.45	8.70	9.22

ここで, 処理全体を通した処理内訳の各コアの平均値では, ストレージの I/O コストを示す iowait は処理全体の 0.2 %と低く, メモリサイズが大きく swap が発生しない環境ではストレージ IO は問題とならない. また, idle 値が処理全体の 57.51 %を占めており CPU 使用率が低いことから, bootstrap や lookup 処理の割合が低い場合はオーバースペックとなることがわかる. 実際は, PBWT や検索クエリのサイズが大きく, bootstrap や lookup の負荷が大きいため, usr 値の割合が大きい.

6.4　予 備 実 験

予備実験として，hyperthread を有効にし, usr 値が高い箇所について, CPU の演算処理とメモリへの load/store 処理の処理の重さを比較した. hyperthread のような仮想的な並列化では計算性能は約 2 倍となるが, メモリへの load/store は逐次処理となる. 計算処理性能を最大限に引き出すと予想される, 物理コア数の倍に当たる並列数 24 まで計測を行った. 無効時並列数 12, 有効時並列数 12, 有効時並列数 24 の場合の実行時間と CPU コストの比較を表 3 に示す. 実行時間は lookup, bootstrap 共に, 無効時 (並列数 12)<有効時 (並列数 12)<有効時 (並列数 24) となる. hyperthread 有効時も, 実行時間は並列数の増加に伴い物理コア数と同じ並列数 12 まで減少し, その後は増加する. 12 以上の並列化に伴う CPU コストの大きな変化が見られないことや, IPC の低下から, メモリへの load/store がボトルネックとなり, 2 倍になった CPU の計算処理性能を活用できていないことがわかる. 有効時と無効時で同じ並列数の場合を比較すると, 有効時の方が低速である. この差は context swtich の多発など, CPU コスト増加によるオーバーヘッドの時間を表す.

従って，CPU 使用率が高い箇所について，メモリへの load/store を並列に行うため, 処理を並列化し, DRAM を並列 IO が可能な SSD に取り替えることが有効な可能性がある.

ここで, context switch の増加に伴い, 一般的に想定される L3 cache load 数・load ミス率,iTLB load 数の増加が見られる. 一方, ゲノム秘匿検索では, iTLB load ミス率や dTLB store ミス率は減少する. 図 6 に dTLB store ミス数・ミス率の変化を示す.

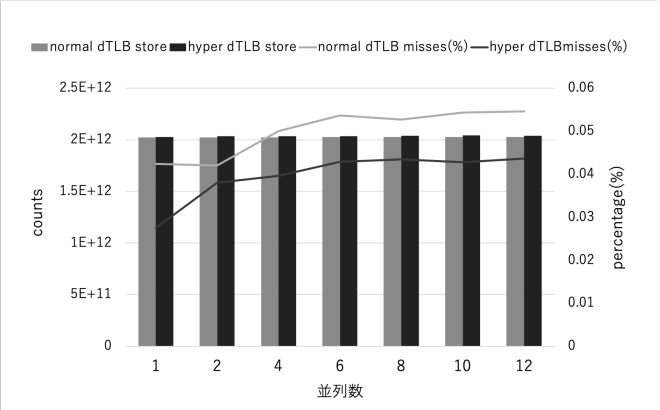

図 6　hyeprthread 有効時と無効時の dTLB store 数・store ミス率の比較

6.5　並列化による処理コストの変化

SSD で並列処理するにあたり, 最適な並列処理の範囲の検討やプログラムの改良が重要である.　そこで, 並列化に伴う CPU コストの変化を計測した. 変化が顕著なものの 1 つに context switch があり, 並列数が 1 増加するごとに処理全体を通して約 1 万回線形増加する. 並列数 12 でも 1 分あたり 120 回程度までしか上らないことから, context switch によるオーバーヘッドは問題にならない. また, 並列数の増加に伴い, L3 キャッシュロードミス率は 4.88 %〜9.00 %, L3 キャッシュストアミス率は 5.00 %〜26.44 %まで上昇する (図 7). 並列に行われる方が短期間にアクセスされるメモリ領域が広いためメモリにないデータにアクセスする頻度が上がることが要因である. ここで, L3 load ミス率に比べ L3 store ミス率の上昇具合が大きい. 読み込みの局所性が高いこと, または DDIO(data direct IO) [22] の機能により, HDD にあるデータにアクセスする際に, HDD から L3 キャッシュに直接データが置かれることが原因である. 同様の理由で page fault も多発する. 並列数 1〜16 で, 6638 回/s〜8449 回/s と緩やかに上昇しており, 並列数 1 でも一般的な値と比較すると高頻度なため, 改善が必要である.

CPU のキャッシュは DRAM に比べて高速にアクセスできるため, キャッシュのサイズを適切にすることや, キャッシュを有効活用できるようなメモリアクセスの構成にすることも重要である. そこで, 暗号化処理でアクセスするメモリ容量とキャッシュサイズと比較するため, 公開鍵やクエリと PBWT の照合

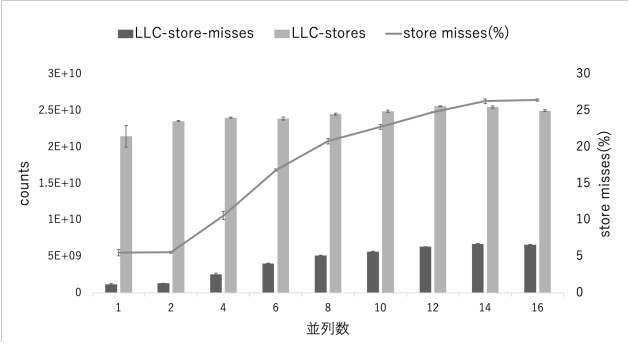

図 7　並列数の変化に伴う LLC store 数・store ミス率の変化

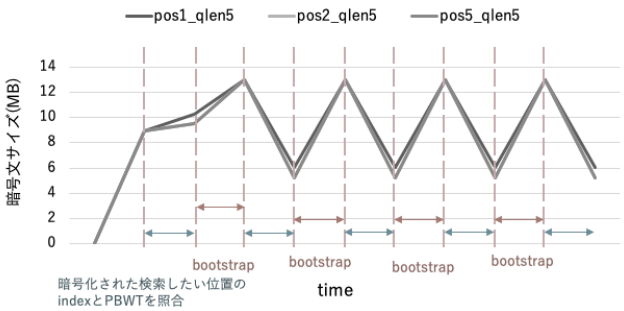

図 8　照合結果を保持する暗号文サイズの変化

結果を表す暗号文のサイズについて計測した結果を図 7 に示す. 初めの検索クエリの暗号化に伴いサイズが約 2600 倍ほど増加し, 暗号化文同士の演算が開始する. ここで検索クエリについて一文字ずつ, PBWT と照合し, 続けて bootstrap による暗号文のノイズの削除を行なっている. グラフから, bootstrap 処理後には毎回 12MB 程度まで大きくなり, 照合により 5MB 程度に減少することがわかる. 原因については今後調査する必要がある. 暗号化した状態で演算を行う最中は, 一度に複数の暗号文を用いるため, メモリ使用量が L3 キャッシュサイズ 20MB を超え, キャッシュミスが発生することがわかる. 暗号処理時のキャッシュミス削減に向けて, プログラムの改良により使用する暗号文の数を減らすことや, 暗号文サイズが小さいライブラリの使用が有効である.

6.6　DRAM と SSD 使用時の処理コストの比較

CPU からのアクセス先を DRAM から SSD に変更した場合, アクセス特性の違いから実行時間や CPU コストの変化にどのような変化が見られるか計測した. ゲノム秘匿検索はファイルの読み書きが少なく, 十分なメモリサイズが用意されている場合はストレージ IO が少ない. そのため, SSD へストレージとしてアクセスするには, メモリサイズを制限し, 意図的に swap 処理を発生させる必要がある.

表 4 に (1) メモリが十分な状態, (2) メモリを 1GB に制限し swap 先を RAM ディスクに割り当てた場合, (3) メモリを 1GB に制限し swap 先を SSD に割り当てた場合の比較を示す. (1) と (2) の比較は swap 処理による変化を表す. (2) と (3) の比較は アクセス先を DRAM から SSD へ変更した場合の変化を表す.

表 4　(1)-(2) swap 発生による実行状態の変化
(2)-(3) アクセス先が DRAM または SSD の場合の実行状態の比較

	(1) 通常状態	(2) DRAM	(3) SSD
総実行時間 (s)	477.33	532.67	555.33
initialization(s)	69.81	76.94	75.35
lookup(s)	144.17	154.71	162.78
bootstrap(s)	164.96	195.58	206.77
IPC	2.56	2.44	2.31
context switch	163,207	646,707	805,489
page fault	15,936,789	19,326,180	20,665,279
L1d load misses(%)	8.67	8.65	8.67
LLC load 数	5.77×10^{10}	6.04×10^{10}	6.04×10^{10}
LLC load misses(%)	8.45	8.81	9.44
LLC store 数	2.56×10^{10}	2.56×10^{10}	2.59×10^{10}
LLC store misses(%)	24.78	25.16	25.37

swap 発生や IO 待ち時間の増加により, 広くメモリアクセスするため, 一般的に想定される page fault の増加, dTLB load ミス率の上昇, LLC load ミス率の上昇が見られる. 一方, ゲノム秘匿検索では, L1 dcache load ミス率は増加しない. 長時間, 高頻度でアクセスされるメモリ領域があることが予想される. LLC store ミス率の増加も見られない. 並列数 12 で実験しており, 通常状態から store の局所性が低いことが原因と考えられる.

また, スワップ処理により, context switch の回数が約 4 倍ほど増加している. 一般的には, ページフォルトによるカーネル空間とユーザ空間の切り替えや, swap 処理の待ち時間にタスクが頻繁に切り替わることが要因である. RAM ディスクに swap する際は 267 回/s, SSD に swap する際も 314 回/s と低く推移しており, context switch は問題とならない.

図 9 に dTLB store 数・store ミス率の変化を示す. swap が発生することにより dTLB store ミス率が大きく減少することが分かる. 減少する原因は今後調べる必要がある. 一方, iTLB は, swap の発生や SSD へのアクセスにより, load 数, load ミス数共に上昇する. 図 10 に iTLB load 数・load ミス率の変化を示す. load 数の増加は context switch の増加による命令の先読みが原因と考えられる. load ミス率は変化が見られない. 命

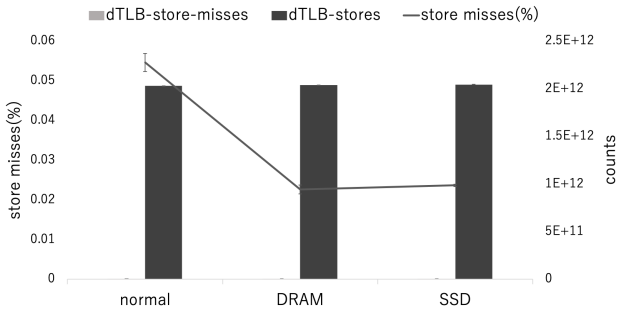

図 9　dTLB store 数・store ミス率の変化

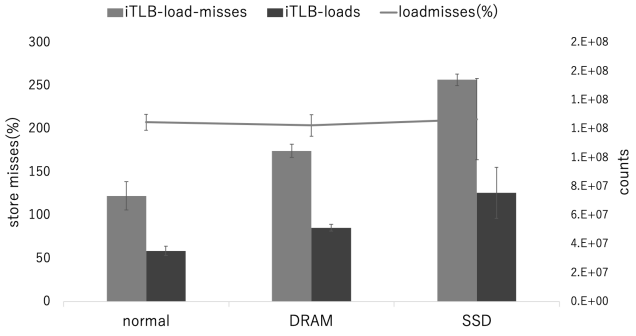

図 10　iTLB load 数・load ミス率の変化

令に関しては, 中程度に局所性があると予想される. iTLB load ミス数に関しては値のばらつきが激しく, スラッシングが起きにくいプログラムの構成や, 適切な iTLB サイズを検討する必要がある.

7 まとめと今後の課題

本稿では, ゲノム秘匿情報検索について, DRAM と SSD の良い特性を引き出せるようなハードウェアの使い分けやプログラムの改良により, 高速実行できる可能性について調査を行った. CPU の処理内訳の考察や hyperthread 有効時との実行状態の比較から, メモリへの load/store 処理が重いことが分かった. 更に, 並列処理可能な範囲が広いことから, DRAM を SSD に取り替え, CPU と SSD の間で並列 IO 処理を行うことで効率化される可能性が高い.

また, 並列処理を行うことによる CPU コストの変化を計測した結果, page fault の多発など, アプリケーション特有の特徴があることが分かった. それぞれの CPU コストの変化について詳しく考察し, 最適な並列数や並列化の範囲を検討することが必要である. アクセス先を DRAM から SSD に変えることによりどのような変化が見られるかについても, 今回計測したデータをもとに考察し, CPU コストの削減方法や DRAM と SSD を階層的に活用するための検討を行いたい.

謝　　辞

本研究の一部は, キオクシア株式会社の支援を受けて実施したものである.

文　　献

[1] Craig Gentry. *A FULLY HOMOMORPHIC ENCRYPTION SCHEME*. PhD thesis, Stanford University, 2009.
[2] 山田 優輝, 小口 正人. クラウド環境におけるゲノム秘匿検索に向けた完全準同型暗号ライブラリの比較と分析. データ工学と情報マネジメントに関するフォーラム (DEIM). 2019.
[3] Yu Ishimaki, Hiroaki Imabayashi, Kana Shimizu, and Hayato Yanama. Privacy-preserving string search for genome sequences with fhe bootstrapping optimization. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3989–3991, 2016.
[4] Muhammad Tirmazi, Adam Barker, Nan Deng, Md.E Haque, Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Fifteenth European Conference on Computer Systems (EuroSys '20)*, pages 1–4, 2020.
[5] 山本 百合, 小口 正人. 完全準同型暗号によるゲノム秘匿検索の分散処理. データ工学と情報マネジメントに関するフォーラム (DEIM). 2017.
[6] 廣江彩乃, 圓戸辰郎, 小口正人. SSD のセキュアな活用に向けた暗号化アプリケーション実行時性能評価. データ工学と情報マネジメントに関するフォーラム (DEIM). 2021.
[7] P.Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of EURO CRYPT 1999, Lecture Notes in Computer Science , Vol.1592*, pages 223–238, 1999.
[8] Cynthia Dwork. Differential privacy. In *Languages and Programming, LNCS, vol. 4052*, pages 1–12, 2006.
[9] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. In *ACM Computing Surveys, Vol. 51, Issue 4*, 2018.
[10] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjosteen, Angela Jaschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. In *Cryptology ePrint Archive 2015/1192*, 2015.
[11] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. In *IBM Research*, 2012.
[12] HElib. `https://github.com/homenc/HElib`.
[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proc. of the 3rd ITCS*, 2012.
[14] PALISADE. `https://palisade-crypto.org/software-library/`.
[15] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, page 409–437, 2017.
[16] M. Georgieva I. Chillotti, N. Gama and M. Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. In *Journal of Cryptology, volume 33*, page 34–91, 2020.
[17] TFHE. `https://tfhe.github.io/tfhe/`.
[18] SBI 証券. DRAM と NAND フラッシュの価格推移 (2020 年 5 月). https://www.sbisec.co.jp/ETGate/?OutSide=on_ControlID=WPLETmgR001Control_PageID=WPLETmgR001Mdtl20_DataStor-eID=DSWPLETmgR001Control_ActionID=DefaultAIDgetFlg=onburl=search_marketcat1=marketcat2=reportdir=reportfile=market_report_fo_topic_200715_01.html.
[19] Intel. 重要なポイントでのパフォーマンス https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/optane-technology/performance-where-it-matters-tech brief.html.
[20] R. Durbin. Efficient haplotype matching and storage using the positional burrows-wheeler transform (pbwt). In *Bioinformatics, vol. 30, no. 9*, pages 1266–1272, 2014.
[21] NTL. `https://www.shoup.net/ntl/`.
[22] Intel Corporation. Intel data direct i/o technology overview. In *TECHNOLOGY OVERVIEW Intel Xeon Processor E5 Family*, 2012.