Vol.2017-OS-141 No.6
2017/7/26

FCMalloc: 完全準同型暗号の高速化に向けたメモリアロケータ

馬屋原昂† 佐藤宏樹† 石巻優† 今林広樹† 山名早人†

概要：マルチコアシステム上で多数のスレッドが同時実行される場合，メモリアロケーションがボトルネックになることがある．これは，複数のスレッドから同時にシステムコールが呼ばれることに起因する．TCMalloc，JEmalloc，SuperMalloc などの従来の汎用用途向けのメモリアロケータでは，各スレッドのローカルヒープメモリへロックフリーでアクセスすることで高速化を実現している．これに対して本稿では，完全準同型暗号計算を対象にした FCMalloc を提案する．完全準同型暗号計算ではメモリ使用量が既知の場合が多く，さらに，ある決まったパターンでメモリアロケーションが繰り返されるという特徴がある．こうした特徴を利用し，FCMalloc では pseudo free によってメモリマッピング情報を繰り返し利用することで，物理メモリレベルでメモリプールを用いる．さらに，ローカルヒープメモリ間の通信経路の構造を全結合とすることで，複数のスレッドによるアクセスのロック競合を減少させる．すなわち，システムコールの頻度を下げ，メモリ管理をできる限りユーザ領域で実現することにより高速化を実現する．完全準同型暗号上で構築した頻出パターンマイニングアルゴリズムである Apriori アルゴリズムを対象とした評価実験の結果，既存手法の中で最も高速である JEmalloc と比較して 2.4 倍の高速化を達成した．

キーワード：メモリアロケータ，malloc，free，仮想メモリ，物理メモリ，完全準同型暗号

1. はじめに

マルチコアシステムにおいて，並列で動作する多くのスレッドが同時に malloc(3)や free(3)などの関数を呼ぶとロック競合が引き起こされる可能性がある．その結果，マシンリソースを有効利用できずに，通常よりも長い処理時間を要する．この問題を解決するために，スケーラブルなメモリアロケータとして，JEmalloc[1], TCMalloc[2], SuperMalloc [3]が提案されている．これらのメモリアロケータは汎用用途向けであり，他のアプリケーションの挙動を妨げないようにメモリ使用量を抑えている．つまり，メモリをできる限り開放するという方針がとられている．しかし，完全準同型暗号（FHE: Fully Homomorphic Encryption）[4]を用いたアプリケーションのように，常にマシンリソースをほぼ占領して実行するアプリケーションで，かつ，メモリアロケーションが一定のパターンに限られる場合，メモリアロケータをさらにチューニングし高速化できる可能性がある．

本稿では，FHE を用いたアプリケーションを対象とした高速なメモリアロケータ FCMalloc を提案する．FHE を用いることにより，データを暗号化した状態で任意の演算を行うことが可能となる．近年のクラウドシステムの普及により，クライアントが保有するデータを外部のクラウドシステムに計算委託する機会が増えると共に，遺伝子データのような「高いセキュリティ保持が必要なデータ」に対する解析ニーズが増大しており，FHE はその一つの解決策として注目されている．

一方，FHE の問題点として，演算に長い時間を要する点や暗号文のサイズが膨大である点が挙げられる．また，演算中も暗号化された状態でデータが取り扱われるため，FHE では暗号化されたデータに対する比較演算ができない．すなわち，条件分岐の適用ができない．したがって，通常ならば計算処理の効率化のために行われる枝刈りなどの標準的な手法を利用できず，全てのデータに対して繰り返し計算を行うパターンが頻出するという特徴を持つ．こうした特徴を最大限に生かすことのできるメモリアロケータが FCMalloc である．

FHE を用いたアプリケーションが持つ「計算処理が静的に決定される」という特徴は，すなわち，「メモリ確保の挙動が予め静的に判定できる」，「特定のメモリサイズでのメモリ確保・開放が行われやすい」ことを意味する．例えば，FHE を用いたアプリケーションでは，暗号文の乗算のたびに，その演算用の一時メモリを必要とする．

FCMalloc では，上述した FHE を用いたアプリケーションのように，1) マシンリソースをほぼ占有して実行するアプリケーションであり，他のアプリケーションへの影響を考慮する必要がない，2) メモリ確保の挙動が既知であり，マシンのリソースを超過しないことが静的に把握できるという前提条件の上で，メモリを操作する際に生じるロック競合によるコストを減らすことで高速化を実現する．具体的には，スレッド毎にローカルヒープを置き，できる限りロックフリーにメモリ管理を行う．さらに，ローカルヒープ同士の通信経路を全結合させることで，のべのロック回数は増えるが，ロック処理を分散化し，総合的にロックのコストを減らす．

本研究の貢献は，FHE に代表されるアプリケーションの特徴に基づいた前提条件を設けた上で，マルチコア環境で発生するロック競合のコストを削減し，高速化を達成したことにある．

本稿の構成は，次に示す通りである．まず，2 節で関連研究を述べ，3 節で提案手法である FCMalloc について説明

† 早稲田大学
Waseda University

ⓒ 2017 Information Processing Society of Japan

する．4 節では，提案手法の有効性を示すために FHE のアプリケーションを用いて評価実験を行う．最後に 5 節で，まとめと今後の課題を述べる．

2. 関連研究

本節では，メモリアロケータの関連研究を紹介する．

2.1 malloc(3)レベルのメモリアロケータ

malloc(3)レベルのメモリアロケータの多くは汎用用途向けである．つまり，同時に動いている他のアプリケーションへの影響を考慮しつつ，メモリ確保に要する時間を短くすることを目標としている．現在普及している CentOS や Ubuntu で一般的に用いられている glibc malloc は，レガシーな実装であるため，スケーラブルなメモリアロケータとして，JEmalloc[1]，TCMalloc[2]，SuperMalloc[3]が提案されている．これらの関連研究と提案手法のメモリアロケータの簡略化した構造を表 1，図 1 に示す．図表中，ローカルヒープは各スレッドに紐付いているヒープであり，グローバルヒープはメモリアロケータ毎にただ一つ存在するヒープである．

glibc malloc はスレッドに紐づくローカルヒープが存在せず，基本的に直接 OS にメモリを要求するレガシーな実装である．さらに，free(3)によって，仮想および物理メモリ空間を開放するため，メモリ使用量については最小限に抑えることができる．しかし，システムコールの発行が増え，新規に確保したメモリ領域へアクセスする際には必ず first touch[a]が生じるため，処理時間がかかる．

一方，表 2 より glibc malloc 以外のメモリアロケータは仮想メモリを開放しないため，既に仮想メモリが確保されている場合には新規にシステムコールを発行する必要がない．64-bit システムにおいて，仮想メモリ空間は物理メモリ空間と比較して巨大であり，仮想メモリ空間の専有は大きなコストではない．仮想および物理メモリ空間の両方を開放しないことは，余計なシステムコールを発行せず，first touch も発生させないことにつながる．一方で，これは処理時間とメモリ使用量のトレードオフとなる．

一般的に，メモリアロケータでは，スレッド毎にローカルヒープを保有し，ローカルヒープの中でメモリプールをロックフリーでアクセスする．グローバルヒープはローカルヒープの情報を一元管理する役割を担う．ヒープ内で保有するメモリプールに関して，使用するメモリ領域を管理するために，ヒープ間の通信が発生する．この通信をグローバルヒープや OS を経由して一元管理すると使用するメモリ容量を小さく抑えることができる．しかし，この通信はロック競合を引き起こす．特にグローバルヒープや OS と通信可能なローカルヒープは制限されるため，ボトルネックとなりやすい．

2.2 メモリライブラリレベルのアロケータ

Web サーバ[5]やデータベース[6]などの特定のアプリケーションに特化したメモリアロケータを用いる場合には，ライブラリレベルで実装することが多い．malloc(3)レベルとの比較を表 2 に示す．表に示す通り，malloc(3)レベルの方が汎用性が高いが，最適化度合いが低くなる．一方で，ライブラリレベルの実装では，関係するライブラリの変更に伴う更新作業が必要となり，保守コストが高くなる．つまり，「汎用性」，「最適化度合い」，「保守コスト」のトレードオフとなる．

表 1 メモリアロケータの比較

名称 (バージョン)	free(3)の処理内容		ヒープ		ヒープ間の通信	
	仮想メモリを 開放しない	物理メモリを 開放しない	グローバル ヒープが存在	ローカル ヒープが存在	グローバルヒープ または OS 経由の通信	直接ローカルヒープ 同士の通信
glibc malloc						
JEmalloc[1]	✔	✔	✔	✔		
TCMalloc[2](2.2rc 未満)	✔		✔	✔	✔	
TCMalloc[2](2.2rc 以上)	✔			✔	✔	
SuperMalloc[3]	✔			✔	✔	✔
FCMalloc(提案手法)	✔	✔		✔		✔

表 2 実装レイヤ別のメモリアロケータの比較

実装レイヤ	malloc(3)	ライブラリ
汎用性	言語・ライブラリを問わない	言語・ライブラリに依存する
アプリケーションの アップデートにともなう保守コスト	不要である	変更を加える必要性があり， バグを発生させる可能性もある
対象アプリケーションの プログラミング言語	基本的には依存しないが，GC を備える言語では動作が隠蔽される可能性がある	実装に用いた言語に限定される
最適化	複数のライブラリを組み合わせた場合でも， ある程度最適化された処理が可能である	個々のライブラリのみに対して 最適化可能

a 仮想メモリへの初回アクセス時，対応する物理メモリのマッピング情報がない場合に割込例外処理として随時 OS がメモリマッピングを行う処理．

ⓒ 2017 Information Processing Society of Japan

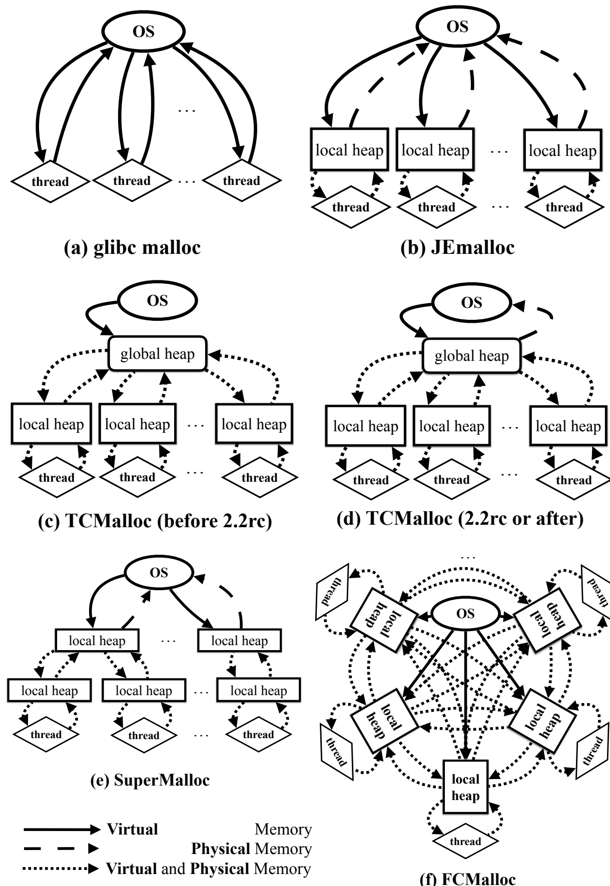


図 1 メモリアロケータの比較

3. 提案手法

本節では，pseudo free と全結合ローカルヒープを利用することにより高速化を図った FCMalloc を提案する．

3.1 対象アプリケーション

FCMalloc が高速化の対象とするアプリケーションは，次の２条件を満たすアプリケーションとする．

1) マシンリソースの占有

マシンリソースを占有して実行できるアプリケーション（例 : FHE）であり，他のアプリケーションへの影響を考慮する必要がない．本条件により，CPU コアやメモリ資源を当該アプリケーションから柔軟に利用することが可能となる．

2) メモリ確保の挙動が既知

メモリ確保の挙動が静的に把握できる（例 : FHE）ことにより，ローカルヒープ間でのメモリプールの制御が可能となる．また，マシンが持つメモリリソースを超過しないことを前提とする．

提案手法は，マルチコアシステムのコア数が多いほど，同一サイズのメモリが頻繁に確保・開放されるほど，ロック数を減らすことができ効果を発揮する．

3.2 提案手法概要

提案のコアとなる技術について述べる．

1) Pseudo free

実際には仮想メモリも物理メモリも開放しない free 処理である．つまり，アプリケーションがカーネル空間へ移動することなく，ユーザ空間で処理が完結する. 図 1(f)において, 各 local heap は OS に仮想・物理メモリを返却していない.

2) 全結合ローカルヒープ

ローカルヒープ同士の通信路を全結合させ，グローバルヒープや OS を必要とせずに，メモリ管理を行う．図 1(f)において，各 local heap は互いに仮想メモリの管理を行うことが可能である．

提案手法は次に示す通り，初期処理後にメイン処理を開始し，上述のコアとなる技術を適用し高速化を図る．

1) 使用メモリ領域分のメモリプールの確保

メモリページ毎にメモリへアクセスを行い，first touch を済ませることで, メイン処理実行に伴う初回メモリアクセス時に，アプリケーションがカーネル空間への移動することを防ぐ．

2) メモリプールのローカルヒープへの分配

既定のコア数を基にメインスレッドとサブスレッド用のローカルヒープにメモリプールを割り当てる．一般的に，メインスレッドではなくサブスレッドにおいて処理を並列に実行することから，総使用メモリ容量 M_{total} = (メインスレッドメモリ容量 M_{main} + サブスレッドメモリ容量 M_{sub} × コア数 N_{core})となる．

3.3 Pseudo Free

本項では，pseudo free について説明する．具体的には，free(3)をオーバライドすることで，free(3)が呼ばれた際に，メモリを開放しない選択肢を実現する．このとき，メモリを開放しないことで，余計なシステムコールの発行によるカーネル空間への移動と first touch の抑制をする．したがって, ユーザ空間のみでプログラムを実行することができ，空間移動に伴うキャッシュフラッシュの頻度の低下にも貢献する．また，スレッド毎のローカルヒープのメモリプールと組み合わせることで，他のスレッドとのロック競合を避けつつ，並列に処理を行うことができるため，効率的に free(3)処理を行うことができる．

一方で，メモリのマッピングを継続して保持するため，常にメモリ使用量を一定の値を維持する．すなわち，メモリ使用効率は悪くなる．

3.4 全結合ローカルヒープ

全結合ローカルヒープは，メモリ管理を行うための通信によるロック競合を減らすために用いる．ローカルヒープとはスレッド毎に管理するメモリ領域であり，スレッド固有領域（Thread Local Storage, TLS）を用いて実装する．また，全結合ローカルヒープは図 1 (f)に示したように，各ロ

ーカルヒープ間で互いに仮想メモリに関する情報を通信する．これによって，ボトルネックとなり得る通信経路が存在しなくなるため，ローカルヒープ間の通信回数は増加するが，全体としてのロック競合のコスト減少につながる．一方で，グローバルヒープや OS を経由しないことにより，情報を一元管理できないという問題が新たに発生する．このため，通信回数の通信内容の最適化を行うことが困難であり，通信回数の増加やメモリ使用量の増加を招く．このような性質から，FCMalloc は特にメモリの確保・開放をマルチスレッドで同時にかつ頻繁に繰り返すアプリケーションに適する．

次に，各ローカルヒープが通信する内容について説明する．3.5 項において詳述するが，各ローカルヒープはメインヒープとサブヒープを持ち，メインヒープは自由に使用可能なヒープ，サブヒープは一時的に保持するだけのヒープであり，自由に使用可能ではない．元々メモリが確保されたローカルヒープとそのメモリを現在保持しているローカルヒープとが同じである場合には，そのメモリはメインヒープ上に存在し，そうでない場合にはサブヒープ上に存在する．このとき，サブヒープは，他のあるローカルヒープにとってはメインヒープとして扱うことが可能である．したがって，サブヒープに関する情報を他のローカルヒープへ通信し，メインヒープとして再利用する必要がある．

また，この全結合ローカルヒープが効果的に機能する例として，あるメモリを malloc(3)したスレッドと free(3)したスレッドが異なる場合が挙げられる．このような状況は例えば，並列プログラミングの producer-consumer パターンにおいて起こり得る．

3.5 ローカルヒープ内のメモリ確保・開放処理

ローカルヒープが保持するメインヒープとサブヒープのそれぞれのヒープは，サイズ別のメモリプールを持つ．これらは片方向リストで表され，使用可能なメモリ領域の先頭アドレスを格納する．また，FCMalloc は要求メモリサイズを 8B 以上の最も近い 2 冪のサイズに丸め込む．

メインヒープは実行中のスレッドにおいて確保されたメモリのみを保持する．一方，サブヒープは他のスレッドにおいて確保されたメモリを一時的に保持する．したがって，サブヒープ内のメモリを元々確保されたスレッドのローカルヒープに返却するために，ローカルヒープ同士でロックしつつ通信を行う．このように，各ローカルヒープのサイズ別の片方向リストが保有するメモリ量は均衡を保つ．

以上をまとめると，malloc(3)は現在実行中スレッドのメインヒープの中の適切なメモリサイズを保有する片方向リストから pop したメモリの先頭ポインタを返り値とする．一方，free(3)は対象となるメモリが元々確保された場所に応じて，メインヒープまたはサブヒープへ push する．

4. 評価実験

本節では，FHE のアプリケーションを用いて FCMalloc の有効性を実験により評価する．アプリケーションとして，FHE スキーム上で Apriori アルゴリズムを構築し，その中で最も時間を要する処理であるアイテムのサポート値計算を実験の対象とする．

4.1 実験対象

FCMalloc の実験対象として, FHE のアプリケーションを用いる. FHE スキーム上の Apriori アルゴリズム[7] を HElib [8]を用いて C++で実装した．このアプリケーションは初期処理，メイン処理，終了処理で構成される．まず，初期処理で暗号文データベースを生成し，終了処理でその暗号文データベースを破棄する．メイン処理では対象データとなるアイテムのサポート値計算を行う．一般的に，Apriori アルゴリズム[9] [10]では様々なミニマムサポート値を設定してクエリを発行するため，メイン処理は繰り返し実行される．したがって，メイン処理に要する時間の改善が重要である．

サポート値計算では，FHE の乗算処理が大量に行われるため, その乗算処理について説明する. HElib では暗号文を Ctxt クラスで表現する．暗号文同士の乗算処理を行うためには，予め演算対象の暗号文のレベルを低い方に一致させる必要があり, modDownToLevel(), tensorProduct()の順番でメソッドを呼び出す必要がある．

次に，本実験におけるアイテムのサポート値計算について説明する．まず，データベースのアイテム数をN_{item}，トランザクション数をN_{trans}とする. また, クエリの頻出パターン候補のアイテム長をL_{item} とすると，そのアイテム集合は$I = \{i_0, \ldots, i_{L_{item}}\}$となる．このとき，暗号文$C_{t_i}$のトランザクション ID をt，アイテム ID をiとすると，サポート値計算は次の式で表される．

$$\sum_{t=1}^{N_{trans}} \left(C_{t_i_0} \cdot C_{t_i_1} \cdot \ \ldots \ \cdot C_{t_i_{L_{item}}} \right) \qquad (1)$$

同一トランザクション毎に暗号文同士の乗算を行った後に，それらの結果を全て加算する．また，クエリは頻出アイテムの数と頻出パターン候補のアイテム長の組み合わせの回数分作成されるため，合計で${}_{N_{item}}C_{L_{item}} \cdot (L_{item} - 1) \cdot N_{trans}$回の乗算処理を行うことになる．さらに，FHE ではN_{slot}個の暗号文を 1 つのベクトルとして扱うパッキング手法[11][12]により, 最終的に行う乗算処理の回数はN_{slot}で除算した$\lceil {}_{N_{item}}C_{L_{item}} \cdot (L_{item} - 1) \cdot N_{trans} / N_{slot} \rceil$回となる．

4.2 実験環境

本実験で用いるサーバのスペックとその環境を表 3 に示す．FHE スキーム上で構築した Apriori アルゴリズムではクライアントとサーバ間で暗号化したデータの送受信を行

うが，その通信は本稿においては本質ではないため，サーバのみを用いた環境で実験を行う．

表 3 実験環境

名称	値
CPU model	Intel(R) CPU E7-8880 v3 @ 2.30GHz
コア数	72
メモリサイズ	1TB
L1, L2, L3 キャッシュサイズ	32KB, 256KB, 45MB
OS	CentOS 6.7 (64-bit)
Linux version	2.6.32-504.30.3
g++ version	4.9.2
g++最適化オプション	-O2

4.3 パラメータ

HElib および Apriori アルゴリズムに関するパラメータをそれぞれ表 4, 表 5 に示す. なお, 表 4 の括弧内の値は HElib におけるデフォルトの値である．

表 4 HElib に関するパラメータ

パラメータ	値	説明
p^r	11^7	平文空間
k	(80)	セキュリティパラメータ
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距離
N_{slot}	915	スロット数

表 5 Apriori アルゴリズムに関するパラメータ

パラメータ	値	説明
N_{item}	18	アイテム数
N_{trans}	1,943,424	トランザクション数
L_{item}	2	頻出パターン候補のアイテム長
N_{mul}	324,972	乗算処理の回数

4.4 実験結果・考察

実験結果を表 6 に示す．実験環境の都合により SuperMalloc[3]は実験対象から除外した[b]．なお，最も良い性能を示す値を太字とした．

結果として，FCMalloc は JEmalloc と比較してメイン処理を2.4倍高速化できることを確認した. メイン処理中で, 特に高速化に貢献した tensorProduct()の処理中の CPU 使用率を図 2 に示す. これらの図は htop[c]コマンドのスクリーンキャプチャである．図 2 (e)より，FCMalloc は pseudo free および全結合ローカルヒープによって，多くの時間においてユーザ空間内（図中緑色で示される部分）で動作していることが確認できる．これはカーネル空間への移行やロック競合コストが解消された結果であると考えられる．したがって，FCMalloc はロック競合を抑えることで，CPU 使用率が最も高くなり, マシンリソースを効率良く利用している．

一方, メイン処理中の modDownToLevel()の処理時間は明らかに JEmalloc より長い．また，このときに malloc(3)および free(3)は頻繁に呼ばれないことを確認した. したがって，メモリアロケータに直接的に依存する結果ではなく，間接的に依存する結果に基づく違いが原因であると考えられる. 仮説として，malloc(3)の返り値のメモリアドレスに起因する可能性がある[13]．その理由はメモリの下位アドレスはキャッシュラインの格納場所に影響するためである．

次に暗号文の free 処理時間について考える. TCMalloc や JEmalloc は非常に長い時間を要し, これはいわゆる 3s pause problem [3]と呼ばれる問題である．この問題は，ガベージコレクションにおいて起こり得る一時停止問題[14]と同様な現象であるが, FCMalloc は pseudo free によってこの問題を解決している．

しかし, FCMalloc は他のメモリアロケータと比較してメモリ使用量が高いことが問題である．これは，FCMalloc がサイズ別のメモリプールを所持し，それを保ち続けていることが原因である．つまり，過去にすでに使用したメモリサイズのメモリプールを将来的に使われる可能性を考慮して，余計に確保したままとしている．未使用領域を完全にサイズ別で管理するのではなく，必要となるサイズに応じて適切に分配して管理することで，この問題の解決に近づくと考えられる．

5. おわりに

本稿では，マルチコアシステム上で多数のスレッドを用いて並列分散処理する場合に問題となるメモリ管理時のロック競合を解消するメモリアロケータ FCMalloc を提案した．FCMalloc は，特に静的にメモリ管理が可能なアプリケーションに対して有効となる. FCMalloc ではメモリ管理を各スレッドのローカルヒープを全結合した通信路で行い，pseudo free によって物理メモリレベルでのメモリプールを用い，汎用的なメモリアロケータよりも高速化した．評価実験として，FHE のアプリケーションの Apriori アルゴリズムの計算処理において，既存のメモリアロケータである JEmalloc と比較して 2.4 倍の高速化を達成した．

今後の課題として，ローカルヒープのサイズ別メモリプールを改善することでメモリ使用量の削減すること，malloc(3)の返却メモリアドレスによるキャッシュラインの配置に考慮することでの処理時間の高速化が挙げられる．さらに，FHE アプリケーション以外に対して検証することで，応用先を広げることを目指す．

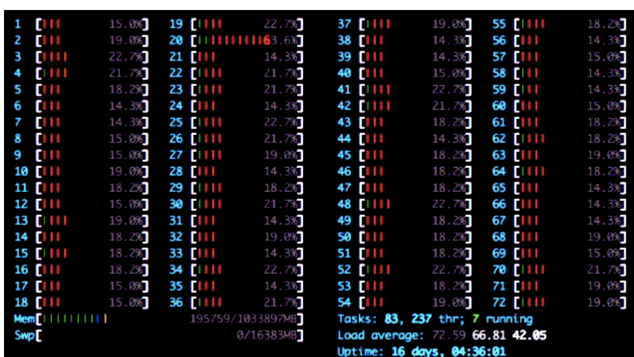
b SuperMalloc は Linux 2.6.38 以降に実装された MADV_HUGEPAGE を指定して madvise(2)の呼び出しを行うが，本実験の環境では未実装である．

c http://hisham.hm/htop/

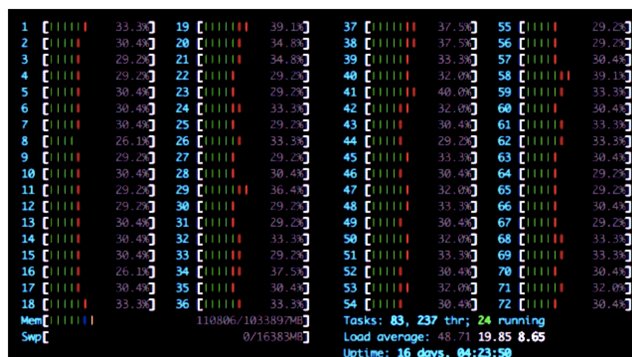
Vol.2017-OS-141 No.6
2017/7/26

表 6　各種メモリアロケータの実行時間比較（Apriori でのサポート値計算部分の実行時間）

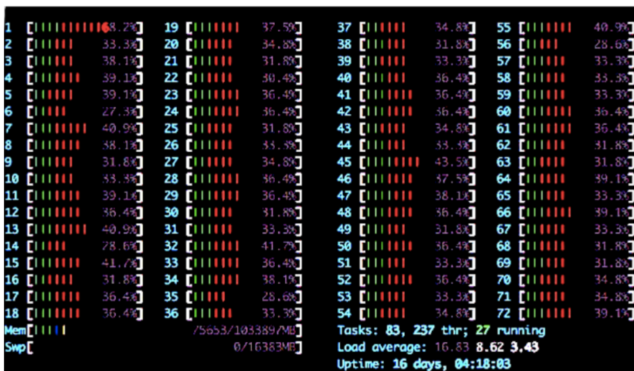
	名称	glibc malloc	JEmalloc[1]	TCMalloc[2]	TCMalloc[2]	FCMalloc(提案手法)
	バージョン	2.12-1	3.6.0	2.0	2.5	-
	初期処理時間[sec.]	301.6	**59.7**	88.6	107.0	83.7
メイン処理	modDownToLevel()の処理時間[sec.]	6.6	**6.0**	17.3	35.0	9.6
	tensorProduct()の処理時間[sec.]	350.0	22.4	94.6	118.9	**5.8**
	終了処理時間[sec.]	0.25	9.3	79.2	4.2	**0.08**
	合計処理時間[sec.]	356.9	37.7	158.1	191.1	**15.48**
	終了処理時間[sec.]	0.9	0.5	0.4	6.3	**0.02**
	最大メモリ使用量[GB]	**197.4**	199.9	208.5	208.8	306.1



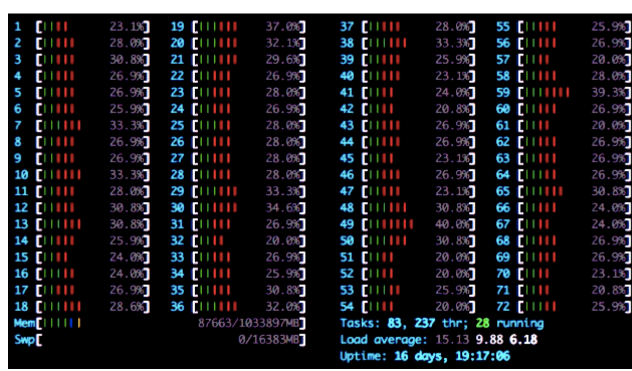
(a) glibc malloc



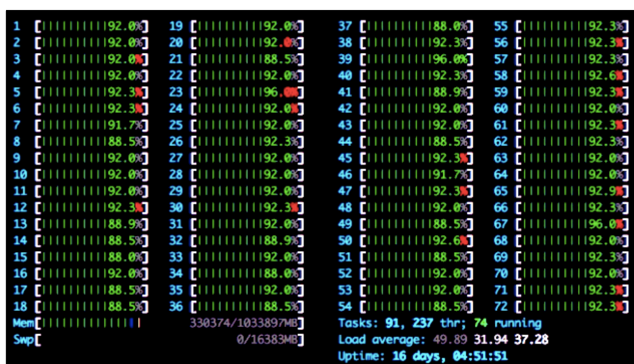
(b) JEmalloc



(c) TCMalloc(version 2.0)



(d) TCMalloc(version 2.5)



(e) FCmalloc(提案手法)

図 2　各メモリアロケータ利用時の CPU 使用率

ⓒ 2017 Information Processing Society of Japan

謝辞　本研究は，科学技術振興機構（JST）CREST（認可番号：JPMJCR1503）の支援を受けたものである．

参考文献

[1] Evans, J., “A Scalable Concurrent malloc(3) Implementation for FreeBSD”, Proc. of BSDCan Conf., 2006.
[2] Google Inc., “gperftools: Fast, multi-threaded malloc() and nifty performance analysis tools”, http://code.google.com/p/gperftools/ accessed on 2017-06-30.
[3] Kuszmaul, B. C., “Supermalloc: A Super Fast Multithreaded Malloc for 64-bit Machines”, Proc. of the 15th Int'l Symp. on Memory Management (ISMM), pp. 41-55, 2015.
[4] Gentry, C., “A Fully Homomorphic Encryption Scheme”, Ph.D. Thesis, Stanford University, 2009.
[5] nginx, https://www.nginx.com/ accessed on 2017-06-30.
[6] MySQL, https://www.mysql.com/ accessed on 2017-06-30.
[7] Imabayashi, H., 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.
[8] HElib, http://shaih.github.io/HElib/ accessed on 2017-03-15.
[9] Agrawal, R., et al., “Mining Association Rules between Sets of Items in Large Databases”, Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data, pp. 207-216, 1993.
[10] Agrawal, R. and Srikant, R., “Fast algorithms for mining association rules”, Proc. of 20th Int'l Conf. Very Large Data Bases (VLDB), pp. 487-499, 1994.
[11] 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.
[12] Smart, N.P. and Vercauteren, F., “Fully homomorphic SIMD operations”, Designs, Codes and Cryptography. vol. 71, no. 1, pp. 57-81, 2014.
[13] Afek, Y., et al., “Cache Index-Aware Memory Allocation”, Proc. of the 11th Int'l Symp. on Memory Management (ISMM), pp. 55-64, 2011.
[14] Gidra, L., et al., “A Study of the Scalability of Stop-the-World Garbage Collectors on Multicores”, Proc. of the 18th Int'l Conf. on Architectual Support for Programming Languages and Operating Systems (ASPOLS), pp. 229-240, 2013.

ⓒ 2017 Information Processing Society of Japan