メニーコア・共有メモリマシンにおける
プロセスとスレッドを併用したスケーラブルな並列処理

馬屋原 昂[†1]　山名 早人[†2]

†1 早稲田大学大学院基幹理工学研究科 〒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, yamana}@yama.info.waseda.ac.jp

あらまし 共有メモリマシン上での並列処理はスレッドベースとプロセスベースの 2 種類に分類できる．スレッドベースの並列処理はそのプログラミングの容易性からプロセスベースよりも一般的に利用されている．しかし，スレッドベースの並列処理はスレッド間でメモリアドレス空間を共有するため，多数のスレッドが同時にメモリを確保する際のロック競合が処理時間のボトルネックとなる．特にメニーコアマシン上ではこのボトルネックが顕著となる．本稿では，このメモリ確保時のボトルネック改善のため，プロセスとスレッドを併用した並列処理手法を提案する．具体的には，メモリアドレス空間の共有によるメモリ管理コストを削減し，複数の子プロセスを並列に生成することで，高速化を図る．72 コアの計算機による評価実験では，ベースラインであるスレッドベースの並列処理と比較して，C++ STL の std::map へのデータ挿入において 9.83 倍，完全準同型暗号のアプリケーションにおいて 14.5 倍の高速化を達成した．

キーワード 並列処理，プロセス，スレッド，メモリアドレス空間

1. はじめに

近年，ビッグデータの利活用が盛んに行われている．地球上で生成されるデータ全体のサイズは 2 年ごとに 2 倍になり，2020 年には 44ZB のデータになると予測されている[1]. その膨大なデータをインメモリで処理する際にはメニーコアマシンが利用される[2]. そして，メニーコア CPU として代表的な Intel®Xeon®プロセッサファミリー[3]のコア数は年々増加傾向にある. このように，メニーコアマシンにおけるビッグデータ処理の需要はますます高まると予想される．

メニーコア・共有メモリマシン上でビッグデータをインメモリで処理する際には，スケーラビリティが低下してしまう可能性がある．具体的には，多数のスレッドが同時にメモリを確保する場合であり，ロック競合が起きる．これはスレッドベースの並列処理がメモリアドレス空間を共有することに起因する．この問題に対して，マルチコア向けのメモリアロケータ [1][2][3][4][5]やメニーコア向けのメモリアロケータ [6][7][8][9]が提案されている．これらはミドルウェアとして扱われるため，メモリアロケータのレイヤ以外の処理に関してはスケーラブルな処理ができないため，メモリ管理に必要なシステムコールを頻繁に呼び出す場合には十分なスケーラビリティを得ることができない．さらに，これらの実装の中にはメモリリーク[5]や segmentation fault[9]を引き起こす実装があるという報告がある．そこで，本稿ではメモリアロケータの変更ではなく，プロセスベースの並列処理を利用することで，スケーラビリティの向上を目指す．この手法は使用するメモリアロケータとは独立しているため, OS に標準搭載されている十分に信頼性のあるメモリアロケータを利用しつつ，メモリアロケータのレイヤ以外の処理をスケーラブルにすることができる．

プロセスベースの並列処理に関する従来研究における問題は, 1) メニーコアマシンにおける実験がないこと, 2) 巨大なメモリマッピングサイズを持つプロセスからの子プロセス生成の処理時間を短縮する提案がないこと, 3) プロセスとスレッドを併用する並列処理を提案していないことである．そこで，メニーコアマシンにおいて，巨大なメモリマッピングサイズを持つアプリケーションの実験を行い，プロセスとスレッドを併用する並列処理を提案することで，スケーラビリティの向上を目指す．

本研究の貢献は，スレッドとプロセスを併用した並列処理を提案し，メニーコア・共有メモリマシン上で評価実験を行い，スケーラビリティの向上を達成したことにある．また，本稿では，単一の共有メモリマシン上での動作を対象とするため，クラスタを用いた並列処理については扱わない．

本稿は以下の構成をとる. 2.で関連研究を述べ, 3.でプロセスの生成に関する予備実験を行い，4.で提案手

[1] https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

[2] https://aws.amazon.com/ec2/instance-types/x1/?nc1=h_ls

[3] https://www.intel.com/content/www/us/en/products/processors/xeon.html

法について説明し，5.で評価実験を行う．最後に，6.でまとめる．

2. 関連研究

共有メモリマシンにおける並列処理はプロセスベースとスレッドベースの 2 種類に分類できる．以下，2.1.でプロセスベースの並列処理，2.2.でスレッドベースの並列処理を説明する．

2.1. プロセスベースの並列処理

並列処理にプロセスを用いる研究の目的はパフォーマンスの向上またはデバッグの容易性の向上の 2 つに分類できる．

2.1.1. パフォーマンスの向上に関する研究

Global Interpreter Lock

Python[4]や Ruby[5]などのスクリプト言語では，インタプリタ本体のプログラム内のデータ競合を防止するために，GIL(Global Interpreter Lock)や GVL(Giant VM Lock)と呼ばれる機構を用いる．これらの機構は同時に動作可能なスレッド数を制限するため，スケーラビリティを低下させる．そこで，プロセスベースの並列処理を適用することで，メモリアドレス空間が分離し，ロック競合の回数が減り，スケーラビリティが向上する．

投機的実行

プログラム上で明示的に並列化されていない処理を対象として，子プロセスを並列に投機的実行することで高速化を行う研究がある[10]．これは処理の一部のデータ依存のために，その処理全体を並列化することが困難である場合に有用である．具体的には，共有データを操作する際に，基本的にはデータの読み書きが競合する頻度は少なく，確率的にはロック操作が不要な場合である．このような場合に，子プロセスを並列に投機的実行し，データの読み書きの競合の有無に応じて，投機的実行の可否を判断する．このとき，子プロセスの生成時間と生成元プロセスへ処理結果をフィードバックする時間を要することや無駄な投機的実行に要する計算資源に注意する必要がある．

メモリ管理コストの削減

プロセスベースとスレッドベースでの並列処理適用事例について比較する．表 2-1 に調査した事例を示す．なお，各々の研究の実験結果はアーキテクチャやアプリケーションに依存するため，直接の比較は困難であることに注意する必要がある．

まず，プロセスベースの並列処理の方がスケーラブルな結果を示す事例の理由として，malloc(3)や free(3)の内部で発生するキャッシュミス回数の削減が挙げられる[14][15]．これはメモリアドレス空間を共有せず，余計なメモリ管理コストが発生しないことによるロック競合の回数の削減の結果である．このとき，並列処理対象処理の粒度が細かい場合には，子プロセスの生成および終了処理のオーバーヘッドが並列処理対象に要する時間よりも大きくなってしまう例がある[15]．また，スケーラブルな並列処理の順番が 1) glibc malloc を用いたプロセス，2) TCMalloc[1]を用いたスレッド，3) glibc malloc を用いたスレッドであるという結果[14]から，メモリアロケータのレイヤのみではスケーラビリティの向上に限界があり，プロセスベースの並列処理が最も高いスケーラビリティを示す場合があることがわかる．

次に，スレッドベースの並列処理の方がスケーラブルな結果を示す事例の理由としては，実験対象マシンのコア数が少ないことが挙げられる[11][12]．つまり，同時並列実行数が少ないため，ロック競合が起きにくく，スケーラビリティが低下しない．また，コア数が 32 コアと多い場合においても，スレッドベースの並列処理の方が高速である例[13]がある．これはアプリケーションが malloc-intensive でないため，そもそもロック競合が起きにくいと考えられる．このとき，スレッドとプロセスによる処理時間の差が生じず，プロセスの生成時間とプロセス間のデータ通信に要する処理時間に加えて，プロセスのスケジューリングやコンテキストスイッチに要する時間の割合が高くなる可能性がある．

表 2-1 プロセスまたはスレッドの並列処理のスケーラビリティにおける優位性

	スケーラブルな並列処理方法 (プロセスまたはスレッド)	CPU の コア数	対象アプリケーション
Lopes ら[11]	スレッド	8	ベンチマーク （フィボナッチ数計算，N クイーン問題， 行列計算，円周率計算）
Inoue ら[12]	スレッド	4, 8	ベンチマーク（Java, PHP）
Barnes ら[13]	スレッド	2, 4, 32	AES
Chen ら[14]	プロセス	8	機械翻訳のデコード処理
Lu ら[15]	プロセス	16	8 種のアプリケーション

[4] https://www.python.org/

[5] https://www.ruby-lang.org/en/

2.1.2. デバッグの容易性の向上に関する研究

マルチスレッドなアプリケーションに頻出する代表的なバグとして，*deal lock*，*race conditions*，*atomicity violations*，*order violations* が挙げられる．これらのバグに対するデバッグを容易にするツール[16]の提案がある．これはスレッドベースの並列処理ではメモリアドレス空間を共有するために，メモリアクセスを行う前にバグ検知を行うことが困難であるが，プロセスベースの並列処理とすることで，メモリアドレス空間が独立するため，事前のバグ検知が可能となり，デバッグの容易性が向上する．また，このツールの実験結果として，通常のスレッドベースの並列処理に近い処理時間で，対象処理の実行とデバッグを同時にできることを示した．

2.2. スレッドベースの並列処理

メニーコア・共有メモリマシン上の malloc-intensive なアプリケーションにおけるスケーラビリティの低下は，主にメモリ管理に関するコストによって引き起こされる．これはスレッドベースの並列処理がメモリアドレス空間を共有することに起因する．例えば，malloc(3)処理の内部ではメモリチャンクの使用状況の管理にロック操作が必要である．このロック操作の競合問題を解決するための研究を以下で紹介する．

マルチコア向けのスケーラブルなメモリアロケータとして，TCMalloc[1]，JEmalloc[2]，Streamflow[3]，TBBmalloc[4]，SuperMalloc[5]，メニーコア向けとして SFMalloc[6]，SSMalloc[7]，scalloc[8]，MCMalloc[9]などの研究がある．これらのメモリアロケータはスレッドごとにローカルヒープを用意することで，ロック操作なしにメモリチャンクに関する情報へのアクセスを実現している．並列プログラミングで有名な producer-consumer pattern においては，メモリの確保と解放を行うスレッドが異なるため，スレッドごとのローカルヒープ内のメモリチャンク量は偏ってしまう．したがって，そのメモリチャンク量を制御するための機構が必要となる．そして，基本的にこの機構ではロック操作を必要とする．

これらのメモリアロケータはそれ自身のレイヤにおいて,スケーラブルな処理を行うための提案であり，異なるレイヤではスケーラブルな処理とはならない．例えば，mmap(2), munmap(2), madvise(2)などのシステムコール内部処理でロック競合が引き起こされ，このシステムコールレイヤでのロック競合の防止は困難である．さらに，同一プロセスが異なるコアでメモリマッピングを変更した際に生じる TLB shootdown は避けることができない．一方，プロセスベースの並列処理では，プロセス単位でメモリマッピングが独立しているため,システムコールによるロック競合を抑え,TLB shootdown を避けることができる．

まとめとして，メモリアロケータはミドルウェアとして使用するため，アプリケーションレイヤのメモリアクセスの独立性を活かすことができず，余計なロック競合が生じてしまう．しかし，子プロセスを利用することで，明示的に独立したメモリアドレス空間での処理を可能とし，ロック競合のコストを抑えることができる．

3. プロセスの生成に関する予備実験

本節では，提案手法の根拠の補強のために，プロセスの生成を行う fork(2)処理に関する予備実験を行う．具体的には，3.1.でプロセスのメモリ使用量に対する fork(2)の処理時間，3.2.でマルチスレッドによる fork(2)の呼び出しの問題点を明らかにする．それぞれの実験環境を表 3-1 に示す．

表 3-1 実験環境

名称	値
CPU モデル	Intel® Xeon® CPU E7-8880 v3 @ 2.30GHz
CPU 数	4
コア数	72 (18x4)
メモリサイズ	1TB
L1, L2, L3 キャッシュサイズ	32KB, 256KB, 45MB
OS	CentOS 6.9 (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

3.1. メモリ使用量と子プロセス生成時間

子プロセス生成元のプロセスのメモリ使用量と fork(2)の処理時間の予備実験を行うが，実際には，fork(2)はメモリマッピングをコピーする．このメモリマッピングサイズは基本的にはメモリ使用量と相関があるが, malloc(3)の呼び出し方法に依存して変化する．したがって，同一のメモリ使用量でも異なるメモリマッピングサイズとなり得るため，malloc(3)の呼び出し方法について，1）4KB ごと，2）2MB ごと，3）一括で行うの 3 種類のメモリ確保の方法を試す．なお，比較のためスレッドでの実装についても同じ評価を行う．

本予備実験の結果を図 3-1 に示す． なお，生成元のプロセスのメモリ使用量が 500GB 以上の場合, 子プロセスの生成に失敗し，errno に ENOMEM の値が設定されたため，評価は 400GB までの範囲で実施した．図 3-1 より，プロセスのメモリ使用量が 1MB 以上のサイズのとき fork(2)の処理時間と比例関係にある．実アプリケーションでは，メモリを一括で確保せず，一定のサイズずつ確保する場合が多い．このとき，メモリマッピングサイズが小さい場合には，プロセス生成のオーバーヘッドは無視できる時間となるが，例えば，メモ

リ使用量が 100GB 以上では秒単位の処理時間を要し，無視できなくなる．したがって，巨大なメモリマッピングサイズを保持するアプリケーションでは fork(2)の処理時間が問題となる．一方，スレッドでの実装の場合，生成時間はほぼ一定であり，メモリ使用量が高くなるにつれて，多少の変化はあるものの，プロセスの生成時間と比較するとはるかに高速である．

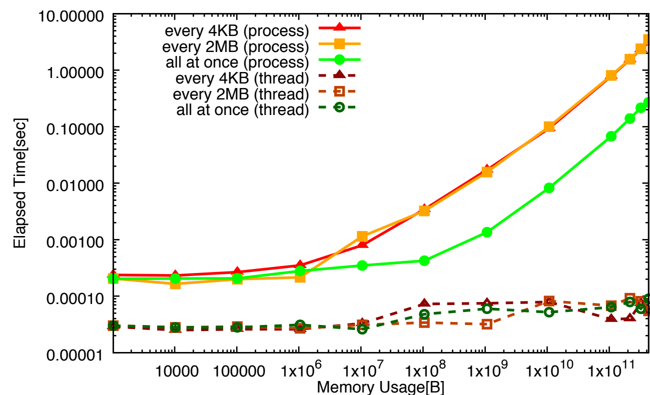

図 3-1 プロセスのメモリ使用量と fork(2)の処理時間

3.2. マルチスレッドによる子プロセス生成

並列に子プロセスの生成する予備実験を行う．この実験の目的はナイーブな手法として考えられるマルチスレッドによる子プロセスの生成を試すことである．シングルスレッドによる呼び出し（直列子プロセス生成）とマルチスレッドによる並列呼び出しを行い，具体的な手順として，100GB のメモリを 4KB ごとの確保した後に，対象マシンのメモリ搭載量を超えないように，計 8 個の子プロセスを生成する．実験結果を図 3-2 に示す．

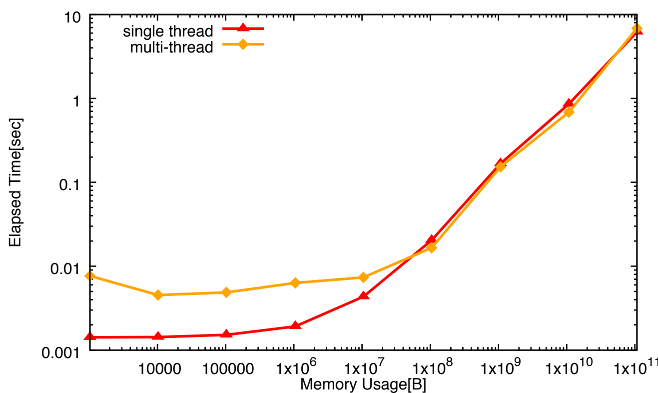

図 3-2 シングルスレッド・マルチスレッドによる fork(2)の処理時間

メモリ使用量が 10MB 以下の場合はスレッドの生成に一定時間を要するため，マルチスレッドによる fork(2)処理の方がより多くの時間を要する．一方，メモリ使用量が 100MB 以上の場合は fork(2)の処理時間に大きな差異はない．これは，同一のメモリアドレス空間において，fork(2)は同時に処理できないためであると考えられる．

また，マルチスレッドによる呼び出しの致命的な問題として，デッドロックが発生する可能性がある．つまり，他のサブスレッドがロックを獲得した状態のメモリをメインスレッドがコピーしてしまう可能性がある．具体的には C++において，サブスレッドで std::cout への書き込み中にメインスレッドが fork(2)した際に，子プロセスが std::cout への書き込みを行うときにデッドロックが発生する．

まとめとして，シングルスレッドによる呼び出しと比較して，マルチスレッドによる fork(2)の並列呼び出しは処理速度を向上させず，デッドロックを発生させる可能性があるため，子プロセスの生成を並列に行う手法として不適切である．

4. プロセスとスレッドを併用したスケーラブルな並列処理の提案

本節では，子プロセスとスレッドを併用したスケーラブルな並列処理を提案する．4.1.にて提案手法の概要を示し，4.2.にて対象とする並列処理の条件を述べ，4.3.にて子プロセスの並列生成手法について述べる．

4.1. 概要

プロセスとスレッドを併用した並列処理手法を提案する．その全体構成を図 4-1 に示す．プロセスベースの並列処理はメモリアドレス空間の独立性により，スレッドベースの並列処理よりもスケーラブルな処理が可能である．しかし，この独立性によって，プロセス同士のデータの共有のために通信が必要となる．そこで，子プロセスの Copy-on-Write を利用して，異なるメモリアドレス空間で読み取り専用の共有データへの効率的なアクセスを実現することにより，プロセス間でデータを共有する．このとき，各子プロセスに対して，スケーラビリティを低下させない程度のスレッド数で並列処理を行うことで，不必要に子プロセスを生成することを防ぐ．ただし，子プロセス数とスレッド数はユーザが決定し，実際には，子プロセス生成元のメモリマッピングサイズや並列化の対象となる処理の malloc(3)や free(3)などのメモリ管理に関する処理の呼び出し頻度に依存して決定する．その基本方針として，malloc-intensive なアプリケーションほどスレッドの同時実行数を減らす．

また，3.の予備実験から明らかに子プロセスの生成において，メモリマッピングのコピーにかかる処理時間が問題となることがわかった．そこで，ナイーブに子プロセスを生成するのではなく，並列に生成することでこの処理時間を短縮する．

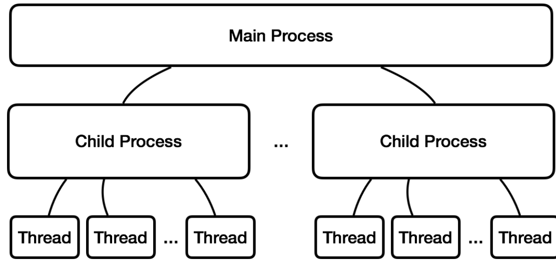

図 4-1 プロセスとスレッドを併用する
提案手法の全体構成

4.2. 対象とする並列処理の条件

提案手法を適用する上で前提とする並列処理対象の条件を次に示す．

条件 1) 必要なデータが事前に揃う

提案手法では，Copy-on-Write によって，親プロセスと子プロセス間でデータのメモリマッピングを共有する必要があるため，並列処理の直前に子プロセスを生成しなくてはならない．したがって，いわゆるプロセスプールを利用することができない．これは，異なるプロセス間で共有してアクセスできるメモリ領域を mmap(2)によって新たに生成することで回避できる．しかし，既存のプログラムの大部分を変更する必要があり，コストが高く現実的ではない．さらに，非プリミティブな値の多くはポインタを利用するため，それらのポインタが mmap(2)で確保した領域のみを指し示す必要がある．

条件 2) 並列処理の粒度が荒い

並列処理の粒度が細かい場合には他の子プロセスのデータが必要となる度に，プロセス間でのデータ通信が発生し，スケーラビリティの低下を引き起こす．したがって，並列処理が互いに独立している状態が理想的である．

条件 3) フィードバックするデータサイズが小さい

メモリアドレス空間が独立しているため，親プロセスに対して，子プロセスの並列処理結果のフィードバックをする必要がある．このとき，途中の計算結果は不要かつ最終的な計算結果のデータサイズが小さくなるほど効率が良い．

4.3. 子プロセスの並列生成手法

子プロセスの生成時間は，生成元のプロセスのメモリマッピングサイズに依存する．したがって，大容量メモリマシン上で実行するアプリケーションでは子プロセスの生成時間がボトルネックとなり得るため，子プロセスを並列で生成する手法を以下に示す．また，図 4-2 に 16 個の子プロセスを並列生成するときのプロセスの生成順とそれらの親子関係の例を示す．

手順 1) 子プロセスの生成元プロセスが fork(2)を呼び出す．

手順 2) 新しく生成したプロセスを含めた各プロセスが fork(2)を呼び出す．

手順 3) 子プロセス数が指定数に達するまで手順 2)を繰り返す．

この手法により，1 プロセスあたりの子プロセスの最大生成数は，直列に生成する場合はn個であるが，並列に生成する場合は$\lceil \log_2 n \rceil$となる．また，3.2.の予備実験より，マルチスレッドを用いたプロセス生成は不適切であり，生成した子プロセスを再利用して木構造のように子プロセスを生成する必要がある．なお，プロセスの join 処理は，プロセスツリーの葉に該当するプロセスから順次行う．

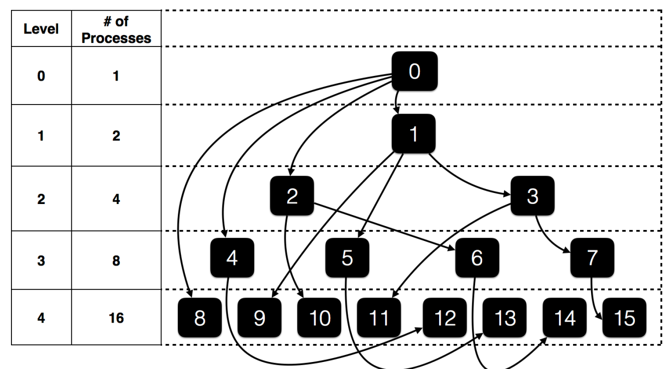

図 4-2 子プロセスの並列生成手法の例
（16 プロセス）

5. 評価実験

本節では，提案手法の評価実験とその結果，考察を述べる．

5.1. 実験対象

次の 3 通りの実験用のアプリケーションを C++で実装した．

実験 1) malloc(3)の呼び出し

malloc(3)で要求するサイズはスモールメモリチャンクサイズ（4KB 以下）とラージメモリチャンクサイズ（256KB 以上）とする．変数はラージメモリチャンクサイズの呼び出し率である．また，オンデマンドページングによって，実際にアクセスがあるまで，ページ割当が発生しないため，ページサイズ（4KB）ごとに確保したメモリへアクセスすることで，メモリマッピングを行う．この実験は最も malloc-intensive なアプリケーションに対する提案手法の影響を確認することを目的とする．

実験 2) C++ STL の std::map へのデータ挿入

変数は std::map へのデータの新規挿入率である．新規挿入率が低い場合には，文書のワードカウント処理と類似する．一方，新規挿入率が高い場合には，ID の重複確認処理と類似する．それぞれ，

新規挿入率を 10%，100%とした．これらは，一般的なアプリケーションの処理の一例と考えられるため，そのようなアプリケーションに対する提案手法の影響を確認することを目的とする．

実験 3) 完全準同型暗号のアプリケーション

完全準同型暗号（FHE: Fully Homomorphic Encryption）[17]を用いて構築した頻出パターンマイニング(Apriori)[18]のアイテムのサポートカウント処理を行う．これは HElib[6]を用いて実装した．主な計算処理は多倍長整数の乗算と加算であり，C++ STL の std::vector を多数利用している．この実験は，特定のアプリケーションに対する提案手法の影響を確認することを目的とする．

実験 1)と実験 2)において，i) スレッドベースの並列処理（glibc malloc, JEmalloc[2], MCMalloc[9]），ii) スレッド・プロセスベースの並列処理（並列子プロセス生成）を試す．実験 3)において，i) スレッドベースの並列処理，ii) スレッド・プロセスベースの並列処理（直列子プロセス生成），iii) スレッド・プロセスベースの並列処理（並列子プロセス生成）を試す．このとき，並列数に対するプロセス数と各プロセスのスレッド数を表 5-1 に示す．これはプロセスの生成時間を考慮した上で，プロセス数を 2 のべき乗に設定し，各プロセスのスレッドがスケール可能な数となるように調整した結果である．

表 5-1 並列数に対するプロセス数と各プロセスのスレッド数

並列数	プロセス数	各プロセスのスレッド数
1	1	1
8	1	8
16	2	8
24	4	6
32	4	8
40	8	5
48	8	6
56	8	7
64	8	8
72	8	9

5.2. 実験環境・パラメータ

実験環境は 3.にて示した表 3-1 と同一である．実験 3)において，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 5-2，表 5-3 に示す．なお，表 5-2 の括弧内の値は HElib におけるデフォルトの値である．$N_{trans} < p^r$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となる値を設定した．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 または 1 の値を用いた．

表 5-2 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 5-3 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

5.3. 実験結果および考察

実験 1)，実験 2)の結果をそれぞれ図 5-1，図 5-2，実験 3)の結果を図 5-3，図 5-4，図 5-5 に示した．

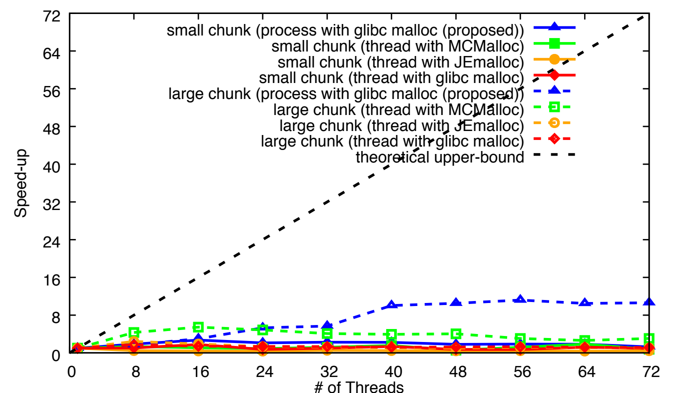

図 5-1 実験 1) malloc(3)呼び出しのスケーラビリティ

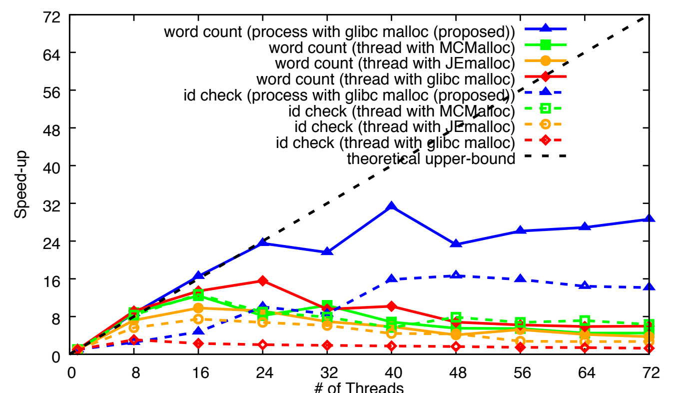

図 5-2 実験 2) std::map へのデータ挿入のスケーラビリティ

[6] http://shaih.github.io/HElib/index.html

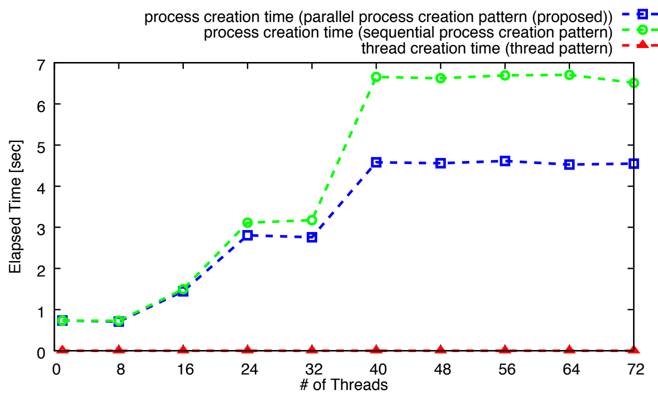

図 5-3 実験 3) 完全準同型暗号のアプリケーションのプロセスまたはスレッドの生成時間

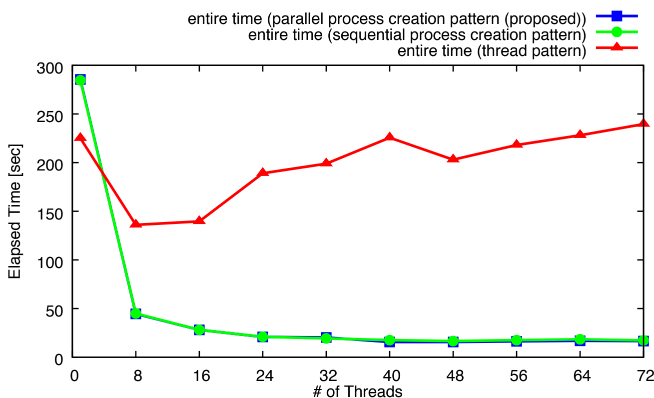

図 5-4 実験 3) 完全準同型暗号のアプリケーションの全体の処理時間

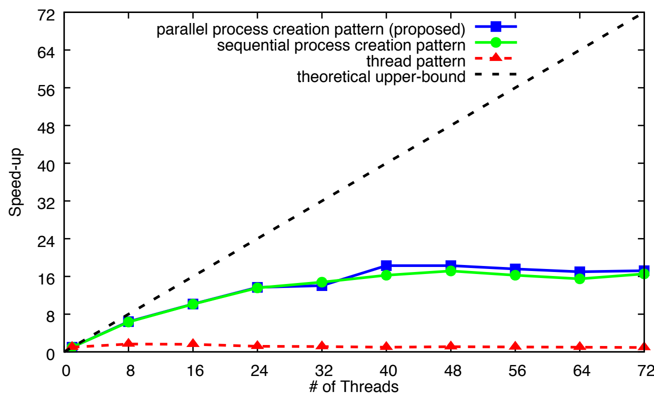

図 5-5 実験 3) 完全準同型暗号のアプリケーションのスケーラビリティ

図 5-1 より，スモールチャンクのメモリ確保処理はプロセスベースの並列処理の場合や異なるメモリアロケータを用いたスレッドベースの並列処理において，スケーラビリティが低い．一方，ラージチャンクのメモリ確保はスレッド・プロセスベースの並列処理のみ，高速化している．これらの結果から，malloc-intensive なアプリケーションにおいて，スレッドベースの並列処理はスケーラブルなメモリアロケータを用いたとしても，スケーラビリティの向上は期待できないが，スレッド・プロセスベースの並列処理によって，高速化の余地があることがわかる．例えば，72 スレッドでメモリアロケータとして glibc malloc を利用したラージチャンクのメモリ確保処理の場合を考える．1 スレッド利用時を基準にしたとき，72 スレッドを利用した場合の理論的な限界は単純に考えると，72 倍であるが，スレッド・プロセスベースの並列処理の並列処理は 10.6 倍，スレッドベースの並列処理は 1.14 倍の速度向上となっている．これは，特にメモリアロケータのレイヤが関与できないレイヤでのボトルネックによるものと考えられる．

図 5-2 より，スレッド・プロセスベースの並列処理はスレッドベースの並列処理と比べて，スケーラビリティが高い．また，std::map への新規挿入率が低いワードカウント処理の方が，新規挿入率が高い ID の重複確認処理よりもスケーラビリティが高いことがわかる．72 スレッドの場合を比較すると，スレッド・プロセスベースの並列処理はスレッドベースの並列処理に対して，ワードカウント処理，ID の重複確認処理の順に，それぞれ 4.84 倍，9.83 倍の高速化を達成している．

実験 1)および実験 2)の結果から，malloc-intensive なアプリケーションの中で，malloc(3)の呼び出し頻度が低い場合にはスレッド・プロセスベースの並列処理の恩恵が得られないが，呼び出し頻度が高すぎる場合には異なるレイヤの処理のボトルネックによって，スケーラビリティの向上が図れない．したがって，適切な呼び出し頻度の状況下において，本提案手法によるスケーラビリティの向上が見込めることがわかる．

実験 3)のメモリ使用量は実験 1)と実験 2)と比較して，巨大であるため，直列子プロセス生成バージョンと並列子プロセス生成バージョンの 2 パターンとスレッドベースの並列処理のプロセスまたはスレッドの生成時間の比較実験を行った．図 5-3 より，並列子プロセス生成の方が直列子プロセス生成よりも高速である．このとき，表 5-1 の並列数に対するプロセス数と各プロセスのスレッド数の関係により，スレッド数が増加してもプロセス生成数が変わらない平坦な区間がある．次に，図 5-4 と図 5-5 より，スレッドベースの並列処理のスケーラビリティは低く，スレッド・プロセスベースの並列処理のスケーラビリティは高い．特に，図 5-5 より，並列子プロセス生成が全体の処理時間と比較しても，高速化に貢献していることがわかる．72 スレッドの場合を比較するとスレッド・プロセスベースの並列処理がスレッドベースの並列処理と比較して 14.5 倍高速である．

以上の実験結果から，スレッド・プロセスベースの並列処理はスレッドベースの並列処理と比較してスケ

ーラビリティが高いことがわかる．さらに，アプリケーションのメモリ使用量が巨大である場合には，並列子プロセス生成を行うことで，さらなる高速化を期待できる．

6. おわりに

本稿では，メニーコア・共有メモリマシン上での並列処理をスレッドベースのみで処理せず，プロセスベースと併用して処理する手法を提案した．また，プロセスの保持するメモリマッピングサイズに応じて，fork(2)の処理時間が増大する問題に対して，並列に子プロセスを生成することにより時間を短縮した．評価実験を 72 コアのマシンで行い，スレッドベースの並列処理と比較して，C++ STL の std::map へのデータ挿入では最大で 9.83 倍，完全準同型暗号のアプリケーションでは 14.5 倍の高速化を達成した．

今後の課題として，並列処理の内容や実行環境に応じて並列処理の効率を定量的に予測する枠組みを構築することで，子プロセス数とスレッド数を自動的に決定する機能が挙げられる．

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] Kuszmaul, B. C., “Supermalloc: A Super Fast Multithreaded Malloc for 64-bit Machines,” Proc. of ACM SIGPLAN Int’l Symp. on Memory Management (ISMM), pp. 41-55, 2015.

[2] Umayabara, A. and Yamana, H. “MCMalloc: A Scalable Memory Allocator for Multithreaded Applications on a Many-Core Shared-Memory Machine,” Proc. of IEEE Int'l Conf. on Big Data, pp. 4764-4766, 2017.

[3] Ding, C., et al., “Software Behavior Oriented Parallelization,” Proc. of ACM SIGPLAN Int’l Conf. on Programming Language Design and Implementation (PLDI), pp. 223-234, 2007.

[4] Lopes, L. M. B. and Silva, F. M. A., “Thread- and Process-based Implementations of the pSystem Parallel Programming Environment,” Software-Practice and Experience, vol. 27, issue 3, pp. 329-352, 1997.

[5] Inoue, H. and Nakatani, T., “Performance of Multi-process and Multi-thread Processing on Multi-core SMT Processors,” Proc. of IEEE Int’l Symp. on Workload Characterization (IISWC), 2010.

[6] Chen, L., et al., “Parallelizing a Machine Translation Decoder for Multicore Computer,” Proc. of Int’l Conf. on Natural Computation (ICNC), vol. 4, pp. 2220-2225, 2011.

[7] Barnes, Angelo, et al., “Improving the Throughput of the AES Algorithm with Multicore Processors,” Proc. of IEEE Int’l Conf. on Industrial and Information Systems (ICIIS), 2012.

[8] Lu, X., et al., “Performance Evaluation and Enhancement of Process-Based Parallel Loop Execution,” Int’l J. of Parallel Programming, 2017.

[9] Google Inc., “gperftools: Fast, multi-threaded malloc() and nifty performance analysis tools,” http://code.google.com/p/gperftools/.

[10] Berger, E. D., et al., “Grace: Safe Multithreaded Programming for C/C++,” Proc. of ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 81-96, 2009.

[11] Evans, J., “A Scalable Concurrent malloc(3) Implementation for FreeBSD,” Proc. of BSDCan Conf., 2006.

[12] Kukanov, A. and Voss, M. J., “The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks,” Intel Technology J., vol. 11, issue 4, pp. 309-322, 2007.

[13] Liu, R. and Chen, H., “SSMalloc: A Low-Latency, Locality-Conscious Memory Allocator with Stable Performance Scalability,” Proc. of ACM Asia-Pacific Workshop on Systems (APSys), No. 15, 2012.

[14] Seo, S., et al., “SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Manycores,” Proc. of IEEE Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 253-263, 2011.

[15] Aigner, M., et al., “Fast, Multicore-scalable, Low-fragmentation Memory Allocation through Large Virtual Memory and Global Data Structures,” Proc. of ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 451-469, 2015.

[16] Schneider, S., et al., “Scalable Locality-Conscious Multithreaded Memory Allocation,” Proc. of ACM SIGPLAN Int’l Symp. on Memory Management (ISMM), pp. 84-94, 2006.

[17] Gentry, C., “A Fully Homomorphic Encryption Scheme,” Ph.D. Thesis, Stanford University, 2009.

[18] Imabayashi, H., et al., “Secure Frequent Pattern Mining by Fully Homomorphic Encryption with Ciphertext Packing,” Proc. of Int’l Workshop on Data Privacy Management (DPM), LNCS vol. 9963, pp. 181-195, 2016.