Vol.2013-DPS-156 No.5
Vol.2013-GN-89 No.5
Vol.2013-EIP-61 No.5
2013/9/11

Indexer Bullet におけるノンブロッキング同期による並列実行方式

藤田 昭人[1,a)]　石橋 勇人[2]

概要：Indexer Bullet （iBullet）はインターネット上に存在する独立した複数の情報リソースを取得し、利用者の目的に応じたデータを抽出してオブジェクトに集約し、そのインデックスを生成、管理するシステムである。独立した情報リソースからのインデックス生成は並列実行が可能であるが、本講ではその実装に先立ちスレッド同期の効率性を確認するため CompareAndSwap(CAS) 操作による Non-Blocking Synchronization による方式と Coarse-Grained Synchronization による方式との性能比較を試みる。

Parallel Execution with Non-Blocking Synchronization in Indexer Bullet

Fujita Akito[1,a)]　Ishibasi Hayato[2]

Abstract: Indexer Bullet is a system for creating and managing enormous data that are aggregated from several information resources on the Internet and tailored for each user. We plan concurrent index generation from individual internet information resource. To archive this, We try to figure out thread synchronization, and compire Non-Blocking Synchronization to Coarse-Grained Synchronization.

1. はじめに

Indexer Bullet(iBullet)[1] はインターネット上に存在する独立した複数の情報リソースを取得し、利用者の目的に応じたデータを抽出してオブジェクトに集約し、そのインデックスを生成、管理するシステムである。

iBullet はコンテンツキャッシュをベースにしたビッグデータ解析向けのプラットホームであり、様々なストレージシステムによるヘテロジニアスな構成に対応する。またインターネットの情報リソースの取得、データの抽出には拡張モジュールを利用する。[2] はビッグデータ解析に必要な複雑なクエリーを実装するためにユーザー定義関数 (UDF) が活用できることを指摘しているが、iBullet では拡張モジュールがこの役割を果たす。

本論文では拡張モジュールでのプログラミング・モデルとその実行を制御するエンジンの実現について述べる。通常、ビッグデータ解析では処理時間を縮退するためにデータ処理の並列実行を試みるが、ビッグデータを対象にした複雑なデータ処理では幾つかの独立した処理アルゴリズム実装を組み合わせてデータ処理全体を多段構成にすることが多い。例えば MapReduce で複雑なデータ処理を行う場合には、Map 関数や Reduce 関数を複数用意したり、複数の MapReduce 処理を繋ぎ合わせて、目的のビッグデータ解析を独立したデータ処理実装のシーケンスとしてテクニックが知られている [3]。

ビッグデータ解析を “複数の独立したデータ処理実装のシーケンス” として記述するためには、個々のデータ処理実装を表現するソフトウェアフレームワークとデータ処理実装間を中継するデータ構造の定義が必要となる。

iBullet が提案する Fetch-Carry モデルでは、関数 Fetch/-Carry が拡張モジュールで実装されるが、両者の実行順序を規定することによりデータ処理全体のシーケンスを記述できるよう配慮されている。また中継データ構造として “

[1] IIJ イノベーションインスティテュート
IIJ Innovation Institute
[2] 大阪市立大学大学院創造都市研究科
Graduate School for Creative Cities, Osaka City University
[a)] akito-fujita@iij-ii.co.jp

Vol.2013-DPS-156 No.5
Vol.2013-GN-89 No.5
Vol.2013-EIP-61 No.5
2013/9/11

インデックス” と呼ぶオブジェクトも用意している。インデックスは “Key/Value リストを表現する抽象的なデータストレージ” と定義されるが、MapReduce において Map 関数と Reduce 関数の間で引き渡される Key/Value ペアのリストと理解するとわかり易い。

以降、セクション２では Fetch-Carry モデルについて、セクション３では iBullet でのデータ処理の並列実行について述べる。セクション４では iBullet の並列処理の前提となるスレッドの同期について一般的な手法を概観し、セクション５ではノンブロッキング同期を含むスレッド同期のベンチマーク結果を示す。セクション６では本論文を要約し、今後の研究について述べる。

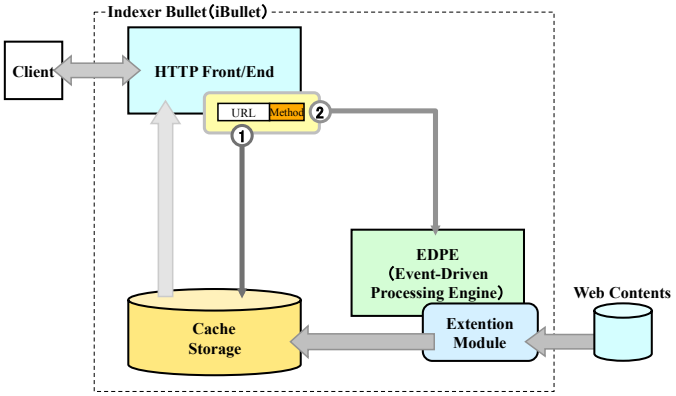

図 1 iBullet のシステム構成

2. Fetch-Carry モデル

Fetch-Carry モデルは任意のビッグデータ解析処理を “複数の独立したデータ処理実装のシーケンス” として記述する目的で検討された。この目的が達成されればビッグデータ解析で頻繁に用いられるアルゴリズム実装の部品化を促し、ビッグデータ解析に伴う各種実装作業の効率化を図る事ができる。

当初より独立したデータ処理実装（以降単位データ処理）を独立したワーカースレッドで実行するモデルを検討したが、ここでの重要な問題はビッグデータ解析処理の一連のシーケンスの中に含まれる可能性のある並列実行の同期処理のサポート方法であった。単位データ処理の中に他の単位データ処理の完了がその起動の前提とする可能性がある。そのため一連のシーケンスにおいて単位データ処理の間で何らかの同期を行う必要があるが、通常それは実装を担当する開発者に委ねられる。

本研究では Fetch と Carry の並列実行について異なる特性を持つ関数を定義する事により、単位データ処理間での同期処理をサポートする。

Fetch 関数は拡張モジュールの中で重複して起動できない関数と定義し、対応する Carry 関数を一連のシーケンスのなかで複数起動、監視できる機能を有するものとする。Carry 関数は必ず対をなす Fetch 関数から起動され、その実行終了も Fetch 関数に通知する。

拡張モジュール内では Fetch/Carry の両関数を定義することができるが、Fetch 関数は１つのみ必須で定義しなければならない。一方 Carry 関数は任意に複数定義することができる。その際、Fetch 関数にはこれらの Carry 関数の起動、監視をするためのコードも実装しなければならない。すなわち拡張モジュール内では Fetch 関数と Carry 関数は１対Nの関係で定義される。

このモデルでは並列実行できるデータ処理は Carry 関数として実装され、その起動と監視は対応する Fetch 関数に記述される。拡張モジュールとしての挙動は Fetch 関数の実行が先行し、その終了後に複数の Carry 関数が実行される deploy 動作と複数の Carry 関数の実行が先行し、全ての Carry 関数の実行が終了した後に Fetch 関数が実行される aggregate 動作が考えられる。

拡張モジュールの実行終了は Fetch 関数の実行結果に基づき決定される。したがって拡張モジュール間での処理の移行は Fetch 関数の実行終了が契機となる。任意のビッグデータ解析処理は一連の拡張モジュールのシーケンスとして記述される。

3. iBullet のシステム実装

本セクションではセクション２で述べた Fetch-Carry モデルによるビッグデータ解析処理エンジンを搭載する予定の iBullet のシステム実装について述べる。

iBullet はコンテンツキャッシュ・サーバーの実装をベースに拡張モジュールの実行エンジンを結合したシステムで、イベント駆動型アーキテクチュア [4] に基づく設計を行っている。図 1 に iBullet のシステム構成を示す。

iBullet は次のコンポーネントから構成される。

- HTTP Front End:
  HTTP プロトコルによるアクセスの受付
- CacheStorage:
  キャッシュ・データを格納するストレージ
- Event-Driven Processing Engine(EDPE):
  キャッシュ・データの取得およびデータ抽出・集約を実行するエンジン

iBullet は原則としてウェブコンテンツ・キャッシュと同様の挙動を行う。HTTP リクエストは HTTP Front End によって受け付けられ、まず CacheStorage に問い合わせ対応するキャッシュデータの存在を確認する。キャッシュデータが存在しない場合は EDPE を呼び出す。EDPE は HTTP リクエストを解釈して対応する拡張モジュールを選択、実行し、データの取得やデータの抽出・集約を行った後、その結果を CacheStorage に格納する。

Vol.2013-DPS-156 No.5
Vol.2013-GN-89 No.5
Vol.2013-EIP-61 No.5
2013/9/11

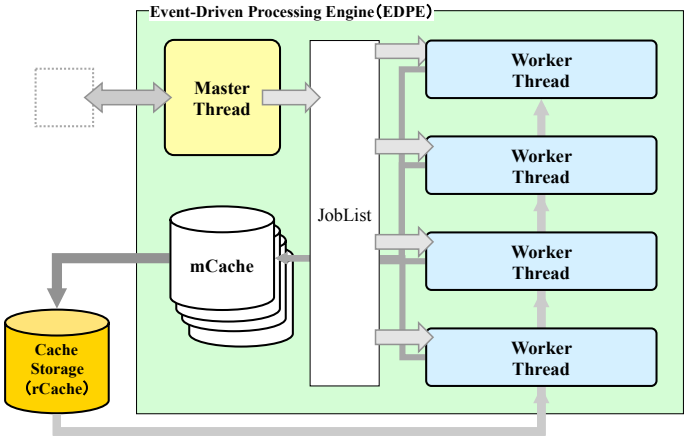

図 2 EDPE の内部構成

3.1 Event-Driven Processing Engine(EDPE)

Fetch-Carry モデルによるビッグデータ解析処理エンジンは EDPE に搭載する。図 2 に EDPE の内部構造示す。

EDPE は制御を司るマスタースレッドと起動時に指定される数のワーカースレッドから構成される。マスタースレッドとワーカースレッドの間では次のオブジェクトが共有される。

- ジョブリスト:
  拡張モジュールの関数を起動する要求を一時的に格納するリスト。EDPE が管理するワーカースレッドは順次ジョブリストから起動要求を取り出し実行する。
- rCache(CacheStorage):
  拡張モジュールから見た場合 iBullet の CacheStorage はキャッシュ済データを管理するデータベース rCache として見える。拡張モジュールはキャッシュ済データの参照のみを行う。
- mCache:
  ワーカースレッドで生成したデータを格納するデータベース。同一拡張モジュールから起動されたスレッド間で共有され、スレッドが全て終了した後、その格納内容は EDPE により CacheStorage に反映される。

Fetch-Carry モデルによるビッグデータ解析処理エンジンを導入する上での現在の EDPE の実装の課題はマスタースレッドとワーカースレッドの間でのスレッド同期のコストである。主たる原因はジョブリストを介した起動要求の受け渡しには大きなコストを要していることである。特にスケーラビリティでは問題があり、現在の実装では起動スレッド数を１６以下に抑制している。これは Fetch-Carry モデルによる拡張モジュールの並列実行をサポートする上での大きな障害となっている。

4. ジョブリストを介したスレッド同期

ジョブリストは iBullet 内で動作する複数のスレッドから発行されるジョブ起動要求を受け付ける。各スレッドで発行されたジョブ起動要求は EDPE の API を介してジョブリストに追加され、各ワーカースレッドはジョブリストから順次新たなジョブ起動要求を取得し、該当ジョブを実行する。

4.1 リンクリストでのスレッド同期

ジョブリストは一般的なリンクリストとして実装することができるが、スレッドによる並行アクセスを許容するためにはスレッドの同期を行う必要がある。すなわちリンクリストを構成する単一のデータ構造に複数のスレッドがアクセスする場合、任意の 1 つのスレッドが他のスレッドの動作をブロックし排他的にデータ構造へのアクセスを行う。

複数スレッドのアクセスによるデータ構造内の矛盾発生を回避するためにはこのような排他制御は不可欠であるが、ブロックされているスレッドの処理は停止するのでレイテンシは低下する。このようなスレッド同期に伴うコストを最小に抑えることは並行システムでの一般的な課題であるが、特に iBullet では全てのジョブがジョブリストを経由して起動されるので、その効率性は iBullet 全体のスループットに直接影響する。

4.2 スレッド同期の手法

The Art of Multiprocessor Programming[5] では、並行アクセスされるリンクリストについて要素の追加 (add)、探索 (contain)、削除 (remove) の操作を規定し、各種のスレッド同期の手法を解説している。

Coarse-Grained Synchronization は単一のロックを活用してリンクリスト全体をブロックする手法である。ロックの獲得、解放に関わるコストは最小だが、全てのスレッドが 1 つのロックで競合するため並行するスレッド数が増大するとレイテンシ、スループット共に低下する。

Fine-Grained Synchronization はリンクリストを構成する要素毎にブロックを行う手法である。Coarse-Grained Synchronization に比べロックが競合する頻度は低く抑えられるので並行スレッド数の増大の影響は低く抑えられるが、各要素毎にロックの獲得、解放を繰り返すのでそれに伴うコストが増大する。

Optimistic Synchronization は Fine-Grained Synchronization と同様に要素毎にブロックを行うが、ロックの獲得、解放に伴うコストを抑制する手法である。要素の追加と削除においてブロックすることなくリンクリストを走査してリスト中の要素の挿入（削除）する位置を特定した後、前後となる要素のみをロックして到達可能性を確認するため再度リンクリストの走査を行う。この手法はロックの獲得、解放の頻度を抑制できるが、到達が確認できない可能性が残る。

Lazy Synchronization は Optimistic Synchronization を改良した手法で、リンクリストからの要素の削除を論理的

Vol.2013-DPS-156 No.5
Vol.2013-GN-89 No.5
Vol.2013-EIP-61 No.5
2013/9/11

```
1  inline unsigned long int
2  compare_and_swap(const char *func,
3    unsigned long int *ptr,
4    void *expref, unsigned char expmrk,
5    void *newref, unsigned char newmrk) {
6    unsigned long int expv;
7          unsigned long int newv;
8
9    expv   = expmrk & 0xF;
10   expv <<= 60;
11   expv  |= (unsigned long int) expref;
12   newv   = newmrk & 0xF;
13   newv <<= 60;
14   newv  |= (unsigned long int) newref;
15
16   *ptr = expv;
17   r = __sync_val_compare_and_swap(ptr, expv, newv);
18   return(r);
19 }
```

図 3 compaire and swap (CAS) の実装

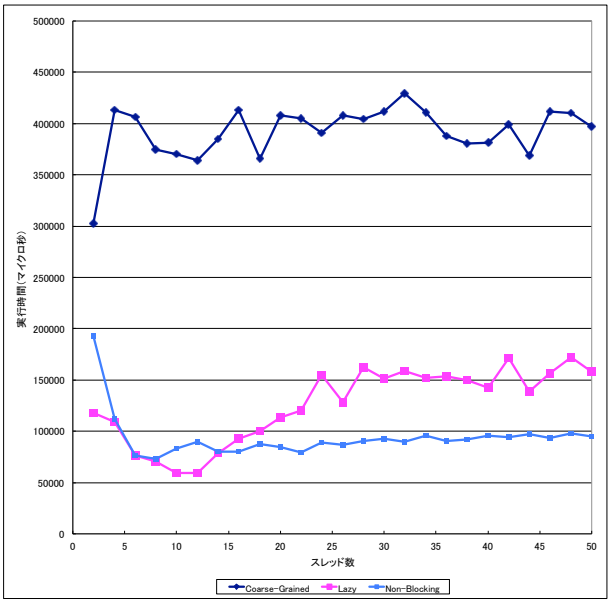

図 4 スレッド同機手法による実行時間の比較

と物理的の２段階に分ける事によりリンクリストをブロックする事なく要素の探索ができ、要素の追加、削除でもリンクリストの走査を 1 回に抑制する。

Non-Blocking Synchronization は Lazy Synchronization のアルゴリズムを compaire and swap (CAS) を使って Lock-Free で実現した手法である。要素の追加と削除ではリンクリストへの挿入（削除）が完了するまで試みを繰り返すが、並行するスレッドをブロックするあるいはブロックされることはない。

5. ベンチマーク

スレッドの同期手法の性能を比較するためリンクリストに対する追加、削除を行うベンチマークを行った。

5.1 シナリオ

このベンチマークはリンクリストに対し 10000 個の要素の追加、削除を並行して行う。同数の追加スレッドと削除スレッドを起動して、全てのスレッドが停止するまでの時間を計測する。なお、同期のコストを把握する事が目的なので要素の追加、削除の正否は考慮していない。

5.2 compaire and swap (CAS) の実現方法

ベンチマークプログラムは C++ で実装したが、Non-Blocking Synchronization で利用する compaire and swap (CAS) は GCC 4.X がサポートする組み込み関数を利用した。ソースコードを図 3 に示す。

この関数は引数として渡される２組の参照ポインタとマーカーを 64bit 整数にパックした後、CAS を実行する。[*1][5] で解説される Non-Blocking Synchronization では Lazy Synchronization のアルゴリズムと同様に要素の論理的削除のためにマーカーを利用するが、CAS を利用する際には参照ポインタと不可分な情報として格納する必要がある。

5.3 ベンチマーク結果

ベンチマークではスレッド数を 2〜50 まで 2 刻みで変化させ、各々１０回の試行の実行時間をマイクロ秒で計測して、その平均値を算出した。ベンチマーク結果を図 4 に示す。

Coarse-Grained Synchronization を用いた場合実行には概ね 400 ミリ秒を要するが、Lazy Synchronization は概ね 150 ミリ秒、Non-Blocking Synchronization は 100 ミリ秒要する。この差は同期の粒度の違いに起因するもので、Lazy Synchronization と Non-Blocking Synchronization がリンクリストの一部の構成要素を同期対象とするのに対し、Coarse-Grained Synchronization がリンクリスト全体を同期対象とするため、競合するスレッドが多いことからスレッド同期の時間的コストが増大していると考えられる。

また Lazy Synchronization と Non-Blocking Synchronization との違いは同期手段（ロックと CAS）に起因する。スレッド数が比較的小さな場合にはむしろ Lazy Synchronization の方が高速であるが、追加スレッドと削除スレッドが各々１６スレッド（したがって全体では３２スレッド）以上では Non-Blocking Synchronization が高速であり、スレッド数が増大するに従い Lazy Synchronization の実行時間も緩やかに増大するのに対し、Non-Blocking Synchronization の実行時間は概ね 100 ミリ秒に留まっている。したがって Non-Blocking Synchronization のほう

[*1] Java の AtomicMarkableReference と等価な機能の実現を企図している

がスケーラビリティが高いと言える。

6. おわりに

本研究では iBullet に Fetch-Carry モデルを導入するため、拡張モジュールの並列実行を予定している。EDPE のビッグデータ解析処理エンジンへの改修に先立ち、ジョブリストによるスレッド同期の効率改善について検討した。

ベンチマーク結果に基づけば Lazy Synchronization と Non-Blocking Synchronization は現在の EDPE で採用している Coarse-Grained Synchronization に比べて 3〜4 倍高速であり、特に Non-Blocking Synchronization は高いスケーラビリティも得られる事がわかった。

今後は EDPE のスレッド同期方式を置き換え、Fetch-Carry モデルに着手する予定である。

参考文献

[1] 藤田昭人，石橋勇人：Indexer Bullet – インターネットの情報リソースを集約するシステムの設計，技術報告 2，IIJ イノベーションインスティテュート, 大阪市立大学大学院創造都市研究科 (2013).
[2] Dean, J. and Ghemawat, S.: MapReduce: a flexible data processing tool, *Commun. ACM*, Vol. 53, No. 1, pp. 72–77 (online), DOI: 10.1145/1629175.1629198 (2010).
[3] 中野猛, 山下真一, 猿田浩輔, 上新卓也, 小林隆：Hadoop Hacks: プロフェッショナルが使う実践テクニック，オライリー・ジャパン (2012).
[4] Wikipedia: *Event-driven architecture*, `http://en.wikipedia.org/wiki/Event-driven_architecture` (2013).
[5] Herlihy, M. and Shavit, N.: *The Art of Multiprocessor Programming*, Morgan Kaufmann (2012).