Vol.2013-HPC-140 No.25
2013/8/1

ストリーミングデータを扱うワークフローの外部モジュールの扱いに着目した低遅延実行

中谷 翔[1,a)]　Ting Chen[1,b)]　田浦 健次朗[1,c)]

概要：計算機が扱うデータの量は増大しており，かつて主流であったようなデータを蓄積してから処理を適用する方法では，処理がデータの流入速度に追いつかないような用途が出てきている．そのような用途に対して，ストリーミング処理が近年重要さを増してきている．ストリーミング処理系は商用のものを含め利用されているが，本稿ではオペレータとして外部モジュールを使用できるような Continuous Workflow の処理系に着目する．Continuous Workflow によって記述される処理の中には，(1) 流入してくるデータに対する処理は極めて短い時間で完了するべき，(2) 処理に使用される外部モジュールの一部が比較的大きな稼働コストを持つ，といった特徴を併せ持つものがある．本稿では，稼働コストの大きい外部モジュールの性能への影響を考慮した上で，Continuous Workflow の低遅延実行を実現するための枠組みを提案する．

キーワード：Continuous Workflow, ストリーミング処理, 低遅延, 外部モジュール

A Low-latency Continuous Workflow Execution Method Focusing on External Module's Running Cost

Sho Nakatani[1,a)]　Ting Chen[1,b)]　Kenjiro Taura[1,c)]

Abstract: Since the amount of data computers process is increasing, processing data stored on disk cannot meet the data flow speed in some cases. Streaming processing is an important key for such calculations. While streaming processors, including commercial ones, are widely spread, we especially focus on Continuous Workflow processor. Some Continuous Workflow applications have such charcteristics: (1) Processing data input into the workflows should be done in quite short time. (2) Some modules, or programs, used in the workflows have large running cost. In this paper, we propose a framework to realize low-latency Continuous Workflow execution taking modules with large start-up cost into account.

Keywords: Continuous Workflow, Streaming Processing, Low-latency, External Module

1. 背景

Web 解析，センサネットワーク，Complex Event Processing (CEP) [18] のような大量のデータを扱う領域では，得られたデータを蓄積することなくその場で処理するストリーミング処理が有効とされる．ストリーミング処理の一例としては，Web を定期的にスクレイピングし，得られたテキストデータを様々なテキスト処理モジュール (構文解析やインデキシング) によって解析し，より構造化された情報を得るようなものが挙げられる．

一方，大規模なデータに対して単純ではない処理を適用するための枠組みとして，ワークフロー [19] が以前より用いられてきた．ワークフローはデータに対して適用する

[1] 東京大学
University of Tokyo, 7-3-1 Hongo Bunkyo-ku, Tokyo 113–0033, Japan
[a)] nakatani@eidos.ic.i.u-tokyo.ac.jp
[b)] chenting@eidos.ic.i.u-tokyo.ac.jp
[c)] tau@eidos.ic.i.u-tokyo.ac.jp

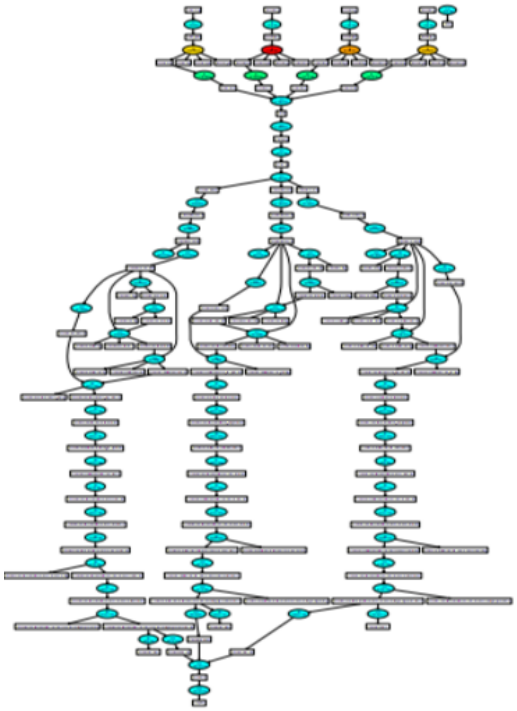

図 1　ワークフロー記述例
Fig. 1　An example workflow description

一連の処理を図 1 のような DAG の形式で記述したものである．ここで DAG のノードはオペレータと呼ばれ，任意の処理を表す．これを高性能に並列実行する研究も数多く行われてきた [2], [4], [11], [17], [22], [23]．ただワークフロー処理というと，処理対象の大量のデータを一旦二次記憶装置に蓄積し，それに対して一度だけオペレータ群を適用するバッチ処理を指すことが多いが，上述のストリーミング処理を行うための Continuous Workflow と呼ばれるワークフロー処理も存在する．

Continuous Workflow は，無限に流入してくるデータに対してオペレータ群を適用するワークフローである．Continuous Workflow アプリケーションの中でも，流入してくるデータをとりわけ短い時間で処理することが求められるものがある．例えば，ニュースサイトからドキュメントを定期的に取得し，インデクシングをはじめとする自然言語処理に掛け，速報性の高い情報源として提供するようなアプリケーションだ．中には，数秒単位の時間で最新のコーパスを検索可能にすることを求めるアプリケーションの要求もある．このようなアプリケーションの特徴は，低遅延性が求められることだけでなく，自然言語処理のような複雑な処理をオペレータとして使用することだ．このような複雑な処理を，ワークフロー処理系の求める特定の API に従って一から記述する必要があるのだとすれば，それは各種応用分野で行われてきた高度な解析処理プログラムを再利用する場合と比べ，生産性の低下を招いていると言えるだろう．我々は，シェル上から発行できるコマンドを 外部モジュール (または モジュール) としてワークフローのオペレータに組込むことが，生産性の点において重要だと考える．

低遅延実行が求められる Continuous Workflow に外部モジュールを組込むことを考えると，一部の外部モジュールの起動時間が無視出来なくなる．Enju [5] という英文を構文解析するモジュールは，ミドルエンドのサーバ上での起動時間に 8.2 秒を要した．起動時の言語モデルデータを読み込む処理が重たいためである．絶えず流入してくるデータが Enju に入力される度にこの起動時間による遅延を受ける必要があるとしたら，数秒単位の遅延しか許されないアプリケーションにおいて Enju を使用することは難しくなる．外部モジュールの起動遅延を避けるために，それらを常駐化 (或いはサーバ化) するという手段も考えられる．しかし，この場合には外部モジュールを常駐させることによるリソースの消費を考えねばならない．Enju では，テキストの入力を待ち受けているだけの状態でも，247MB のメモリを消費していることが確認された．これは現在のコモディティー・コンピュータのメモリ容量を顧みると無視できない上，Enju のようにメモリを大量消費するプロセスは 1 つのノード上で複数走ることも考えられる．外部モジュールによるメモリ消費量が増大すると，その分 Continuous Workflow 処理でデータをオンメモリに加工していくための余裕が減り，ワークフロー全体の性能の劣化に繋がる．

本稿は，外部モジュールをあるタイミングでは起動させ続け，別のタイミングでは終了させる 死活管理 を伴う Continuous Workflow の低遅延スケジューリングを主眼として扱う．これは，外部モジュールを使用した生産性の高い Continuous Workflow 記述を低遅延に実行するための重要な着眼点であると考える．本稿で示すスケジューラは現時点で十分に設計・実装されているとは言えないが，外部モジュールの死活管理が性能に与える影響について示唆を与えることと期待する．

以降の構成は次の通りである．まず 2 章で，本研究の関連研究を示す．そして，3 章において我々が開発中の Continuous Workflow 処理系のモデルを示す．ここには，本稿の主眼である外部モジュールの死活管理についても記述する．4 章において低遅延スケジューリングの現時点での手法を紹介し，5 章においてその評価を行う．最後に 6 章において本稿をまとめる．

2. 関連研究

2.1 Continuous Workflow 処理系

我々は，Continuous Workflow を低遅延実行する処理系を作成することを目指している．対象とする Continuous Workflow は，Filter や Join のような基本的なデータ処理のみならず，シェル上から発行するようなモジュールもオ

ペレータとすることができるものとする．

Nova [20] は Continuous Workflow を扱う処理系の一つである．Nova は Web 解析の分野で効果を発揮するシステムとして開発された．Nova の扱うワークフローのオペレータは Pig Latin [21] で記述され，それらは Pig [3] で Hadoop [1] ジョブに変換されて分散実行される．Hadoop のジョブは巨大なデータを対象にしたバッチ処理には非常に高いスループットを示すが，一方でジョブの立ち上げのコストが大きいため，低遅延性が求められる処理には向かないことが知られている．更に，Nova のモデルでは 1 つずつのオペレータを Hadoop で分散処理することを前提にしているため，同時刻に異なるオペレータがスケジュールされることは考慮されておらず，低遅延実行に効果のあるデータのパイプライン処理は実現されない．彼らも [20] の中で *"Map-reduce-style systems cannot achieve the ultra-low latencies required in some contexts"* と述べているように，Nova は比較的粒度の大きなデータを流入させる Continuous Workflow を対象にした処理系であると言えよう．一方で，我々の処理系は，例えば Twitter のツイートデータような速報性の高いデータを秒単位の遅延で処理する応用に焦点を当てている．

2.2 ストリーミング処理系

Continuous Workflow のオペレータを，外部モジュールや Hadoop ジョブなどよりも粒度の小さな処理に置き換えて考えると，データフローを対象にしたストリーミング処理に行き着く．Data Streaming Management System (DSMS) [16] は数多く研究がされているストリーミング処理系の一派で，SQL にウィンドウというデータの有限区間を指定する概念を追加した CQL [12], [13] を用いて，データストリームに対する処理を記述する．

また近年では，機械学習アルゴリズムを並列実行できるストリーミング処理系の Jubatus [8] や，Complex Event Processing (CEP) [18] と呼ばれるイベント処理をオンラインに行うための Esper [6] など，商業分野でも利用されるようなストリーミング処理系も多く出てきている [7], [9], [10]．しかし，このようなシステムの中には，既存の外部モジュールをデータフロー処理のオペレータとして使用することができ，更にはその稼働コストまでを考慮したスケジューリング手法を実装したものは見受けられない．

2.3 Continuous Workflow のスケジューリング手法

本稿では，Continuous Workflow のスケジューリング手法について，外部モジュールの稼働コストを考慮することを提唱する．

Continuous Workflow のスケジューリングに関する Carney 等の研究 [14] は単一ノードでの実行のみが考察されているが，我々は並列に動作する処理系のスケジューラを念頭に置いている．

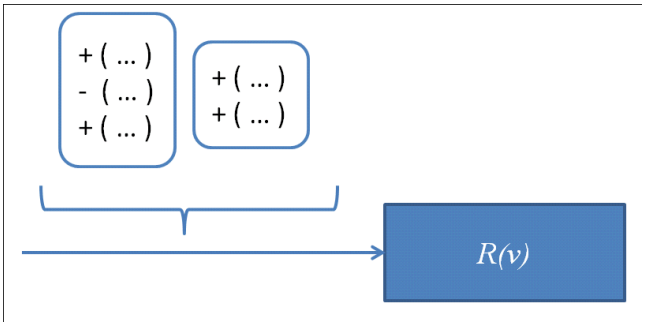

図 2 リレーション，ブロック，タプルの関係．タプルの +/- は，それぞれ insertion 操作と deletion 操作を表す．

Fig. 2 Relation, block, and tuple. +/- next to tuples represents insertion and deletion respectively.

Olston は Nova [20] の基礎となるスケジューリング手法 [15] を提案した．このスケジューラはストリーミングデータの低遅延処理を目指したものであり，手法はストリーミングデータを有限サイズに区切った各ブロックが辿り得るオペレータのパスを列挙し，オペレータの選択率 (Selectivity) から計算されるコストを最小化するパスを探索する比較的単純なものであるが，シミュレーションによる評価で短いスケジューリング時間で Continuous Workflow 実行時間の大幅な削減を実現した．しかし，このコストモデルにはオペレータの起動時間や消費メモリへの考慮が欠けている．Continuous Workflow の低遅延実行にはこれらを考慮したコストモデルが重要であると我々は考える．

3. Continuous Workflow 処理系

3.1 ワークフローモデル

本節では，我々の開発する Continuous Workflow 処理系が対象とするワークフローのモデルを示す．このモデルは Nova [20] のものを参考にし，一部を詳細化した．

ワークフローはリレーション，オペレータ，エッジから構成される．図 6 はワークフロー記述の一例であり，本稿ではオペレータを楕円，リレーションを長方形，エッジを単方向の矢印で表現する．

リレーション　リレーションは，ワークフローを流れるデータの意味的最小単位のタプルの集合である．リレーション・タプルという名称はリレーショナルデータベースの用語だが，ここではタプルは必ずしも構造化データである必要はない．更に，処理の最小単位として，タプルの集合のブロックがある．リレーションは，前後のオペレータとブロックを入出力するものである．これらをまとめたものが図 2 になる．

リレーションの永続化されたスナップショットを $R(v)$ と表記する．v はバージョン番号である．リレーションに入力されるブロックはスナップショットとの差分

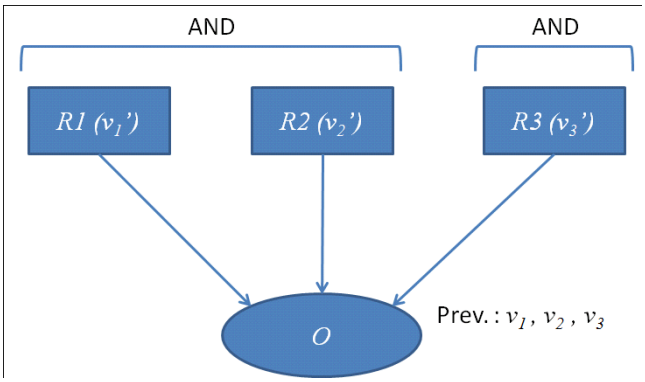

図 3　リレーションの AND グループ
Fig. 3　AND group of relations

情報 (insert, delete, update) をタプルの形式で持っていて，ブロックを現在のスナップショットに適用する操作を *merge* と呼ぶ．即ち，b をブロックとして $merge(R(v), b) = R(v+1)$ である．ただし，$R(v+1)$ は $R(v)$ に b の持つ差分情報を適用したもの．永続化されないリレーションは，$R(v) = \phi$　for any v であると考えることができる．

オペレータ　オペレータは，ブロック群を入力とし，ブロックを出力する演算子である．オペレータには単純なデータ処理関数 (Filter, Join) や，外部モジュールを使用することができる．オペレータがブロック群を受け取ってコアに割り当てることが可能になる条件は，次のエッジの説明に記す．

エッジ　エッジはリレーションからオペレータ，或いはオペレータからリレーションを単方向に指し，ブロックの流れを表す．リレーションに生えるエッジ・リレーションから生えるエッジは高々 1 本 (0 本の場合はそれぞれワークフローの開始点，終了点) であり，オペレータに生えるエッジ・オペレータから生えるエッジは少なくとも 1 本である．オペレータから複数のエッジが生えている場合は，オペレータが処理を終えてブロックを生成した際にエッジの先の複数のリレーションにそのブロックが渡される．一方でオペレータに向かって複数のエッジが生えている場合は，次の条件を満たされた場合にそのオペレータがコアにスケジュール可能になり，データブロックをエッジの根本のリレーション群から受け取ることになる．図 3 の用に，1 つのオペレータに向かってエッジを生やすリレーションには AND グループが定義される．図 3 の O は自分が以前スケジュール可能になった際の入力側のリレーションのバージョン (v_1, v_2, v_3) を記憶しており，それを現在のリレーションのバージョン (v_1', v_2', v_3') と比較して，$(v_1' > v_1 \ \&\& \ v_2' > v_2) \ || \ (v_3' > v_3)$ が成立する場合に限って再びがスケジュール可能となる．

リレーションからオペレータに生えるエッジには ALL, NEW のいずれかの属性が，オペレータからリレーション生えるエッジには Δ, B のいずれかの属性が付与される．ALL は，オペレータに現在のスナップショット $R(v)$ を全て渡すことを示す．一方で NEW は，リレーションの現在のスナップショット v' から，オペレータが以前にブロックを得た時点でののリレーションのスナップショット v の差分，即ち $diff(R(v'), R(v))$ を渡すことを示す．ここで，$diff$ 演算は $merge$ の逆演算である．また，Δ エッジで入力されたブロックは $merge(R(v), b) = R(v+1)$ でスナップショットにマージされ，B エッジで入力されたブロックは，$merge$ 適用前に一旦現在のスナップショットを空 (ϕ) にしてからマージすることを表す．

3.2　モジュールの死活管理

我々の目指す Continuous Workflow 処理系は，上述のモデルに従って動作するだけでなく，インテリジェントなモジュールの死活管理も行うことを目指す．モジュールの中には稼働コストが大きなものがあり，そのようなモジュールを立ち上げたままにしておくか処理の区切りのたびに終了させるかは，Continuous Workflow の低遅延実行実現において考慮に入れるべきと言える．これは，大量のデータを蓄積し，それに対してワークフロー処理をバッチ処理的に実行する際には顕在化しなかった問題である．何故ならば，バッチ処理では稼働コストの大きなモジュールは 1 度処理の区切りに達すると 2 度と処理をしない場合が多く，仮に 2 回目以降の処理があるとしてもそのモジュールの入力データは大きいため，モジュール起動時間はデータ処理の時間に比べて無視できる場合が大抵だったからだ．一方でデータをストリーミング処理する Continuous Workflow 処理系においては，データブロックは任意のオペレータに無限に入力される上，アプリケーションによってはブロックサイズは KB 単位の大きさの場合もある．我々は，モジュールをワークフロー定義に再利用することは分散アプリケーション開発の生産性高めるものであり，モジュールの死活管理を行うことが Continuous Workflow の 低遅延実行の実現に重要だと考える．

モジュールの死活管理はまず単純に，

- データブロック群の処理が終了するたびに処理に使用していたモジュールを終了させる．
- モジュールを常駐化[*1]させる．

という 2 つの手法が考えられる．常駐化する場合のメリットは，そのモジュールが起動に時間が掛かるものであった場合に，そのペナルティを払う回数を減らせることにある．

[*1] 例えば入力データを標準入力から読み取るようなアプリケーションにを常駐化するには，データブロック群の読み取りを 1 回終えた時点で EOF コードを送らず，そのまま入力待ちをさせればよい．

一方で，そのモジュールが起動中に (特別な計算ををしていなくても) メモリやネットワーク帯域を消費するようなものであった場合には，適宜モジュールを終了させることも他のデータ処理の妨げにならないために重要である．

我々の目指すインテリジェントなモジュールの死活管理は，これらのトレードオフを踏まえ，Continuous Workflow アプリケーション全体の実行時間を短くするための動作を目指す．そのコンセプト実装として，4 章においてモジュールの死活管理を含んだスケジューラを提案し，5 章で評価を行う．

4. スケジューリング手法

本稿では，稼働コストの大きい外部モジュールの死活管理に着目し，3 種のスケジューラを考案し評価する．これらのスケジューラは，モジュールの死活管理に関する部分以外では共通の動作をする．

KillEverytime 入力データブロック群を処理するたびにモジュールを終了させる．

NoKill 一度あるコアで立ち上がったモジュールはそのままそのコアに常駐させる．

PlanningKill モジュールの起動時間やメモリ消費を考慮し，モジュールが入力データブロック群を処理するたびに，式 (1) で表される 2 値関数で終了するかを判定する．ただし，$LaunchMem^*$ はモジュールが起動中に消費するメモリ量，$NumCoreWithModule^*$ はそのモジュールを既に立ち上げている他のコアの数，$LaunchTime^*$ はモジュールの起動に要する時間，$OpWithModule^*$ はそのモジュールを実体としてもつオペレータの個数の関数であり，いずれも 0 以上 1 以下の実数値に正規化されている．また，各定数は $0 \leq a_i, b_j \leq 1,\ \sum a_i = \sum b_j = 1,\ -1 \leq c \leq 1$ を満たす．式 (1) の左辺は，モジュールを終了させることで得られる利益を表すように構成したものである．

$$\begin{aligned} & a_0 \times LaunchMem^* + a_1 \times NumCoreWithModule^* \\ - \ & b_0 \times LaunchTime^* - b_1 \times OpWithModule^* \\ < \ & c \end{aligned} \tag{1}$$

5. 評価

本章では，4 章に示した 3 種類のスケジューラの評価を行う．3 章で記述した Continuous Workflow モデルを処理するシステムは開発中であり，本稿の評価にはシミュレータを用いた．

評価は 2 種類のワークフローで行う．はじめのワークフローは，*NoKill* スケジューラではオンメモリなデータ処理を達成することが難しく，*PlanningKill* スケジューラが役立つものである．もうひとつのワークフローは，単純だがより実際的なワークフローアプリケーションに近いものである．ここでは *NoKill* スケジューラが低遅延実行を達成するが，*PlanningKill* スケジューラも *NoKill* スケジューラと同等の動作をすることを示す．これらの評価によって，*PlanningKill* スケジューラが行うような外部モジュールの死活管理の有効性を示す．

5.1 シミュレータの構成

5.1.1 シミュレータの状態遷移

シミュレータは，ワークフローの実行状態とリソースの状態を持ち，その状態をサイクル毎に変化させることで動作する．より具体的には，

- リレーションのスナップショット
- リレーションに *merge* される前のブロック
- オペレータの処理状態

などがワークフローの実行状態であり，

- 各コアの処理しているオペレータ
- 各メモリの所有するリレーションのスナップショットやブロック
- 各メモリの消費量

などがリソースの状態である．

シミュレータは，JSON 形式で記述されたワークフロー定義とリソース定義を読み，それを初期状態として動作を開始する．ワークフロー定義の例を下記に示す．この定義は，図 6 のワークフロー記述を一部簡単化して抜粋したものである．

```
{
  "R_tweets": {  # リレーション
    "prev": "",
    "next": "O_content_extractor",
    "in_type": "delta",
    "out_type": "NEW",
  },
  "O_content_extractor": {  # オペレータ
    "prevs": [["R_tweets"]],  # 内側のリスト: ANDグループ
                              # 外側のリスト: ORグループ
    "nexts": ["R_tweet_contents"],
    # オペレータの実体であるモジュール
    "module": "content_extractor",
    # 入力データサイズと出力データサイズの比
    "in_out_ratio": 0.7,
    # オペレータの実行時間．入力データサイズ d の関数．
    "run_sec": "(lambda d: 0.01 * 1e-3 * d)",
    # 外部モジュールのデータ処理中の消費メモリ
    "run_mem": "(lambda d: 2 * d)",
    # 外部モジュールの立ち上げ時間
    "launch_sec": 1e-3,
    # 外部モジュールの立ち上がり中の消費メモリ
    "launch_mem": 100 * 1e3,
  },
  "R_tweet_contents": {
    "prev": "O_content_extractor",
    "next": "O_english_extractor",
    "prev_type": "delta",
    "next_type": "NEW"
  },
  ...
}
```

5.1.2 オペレータのとる状態

各オペレータは，次の状態のいずれかをとる．

unscheduled いずれのコアにも割当てられていない状態．

receivable ある 1 つのコアに割当てられ，リレーションから入力データを受け取るのを待っている状態．以降の状態には，そのコアに割当てられたまま遷移する．

receiving リレーションから入力データを受け取っている状態．本シミュレータでは，データ転送は非同期に

図 4　シミュレータの可視化
Fig. 4　Simulator Visualization

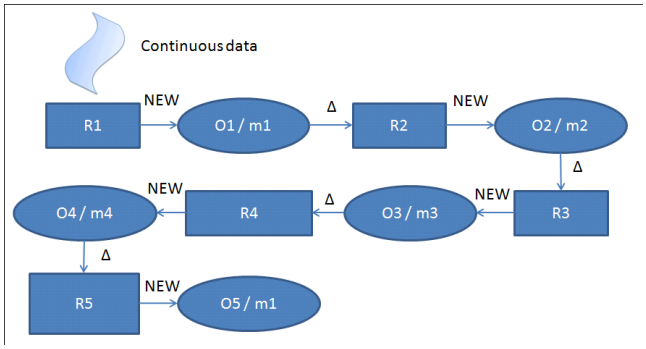

図 5　稼働コストの大きいモジュールを含む合成ワークフロー
Fig. 5　Artificial workflow w/ heavy modules

表 1　可能な最大のデータ流入速度 (1 block = 1KB)

スケジューリング手法	流速 [秒/block]
KillEverytime	0.245
NoKill	0.425
PlanningKill	0.145

行われるものと仮定して，receiving 中のオペレータは CPU 時間を消費しないものとしている．

recceived　入力データの受け取りが完了した状態．オペレータの実体であるモジュールが既に起動していたら runnable に，そうでなければ launching に移行する．

launching　モジュールを起動している状態．

runnable　コアで実行可能な状態．

running　コアで実行中の状態．1 つのコアは同時に 1 つのオペレータのみを実行するが，公平性のため，一定の CPU 時間を消費すると，コアに割当てられた別のオペレータに処理を譲る．

5.1.3　スケジューラのカスタマイズ API

本シミュレータは，種々のスケジューラをプロトタイプ実装するための API を持つ．スケジューラの実装者は仕様に従って API 関数を記述することで，シミュレータにその関数を呼ばせることができる．例えば，**runnable** なオペレータから **running** にする 1 つを選出する API は，5.1.1 節に記した状態を受け取り，1 つ以下のオペレータを返す仕様を満たすものである．

本稿で評価する *KillEverytime* , *NoKill* , *PlanningKill* の 3 つのスケジューラの違いは，オペレータが実行を終えて **running** から再び **unscheduled** 状態に移行する際に，実体であるモジュールを終了させるか否かの 2 値判定を行う API の実装にある．

逆に，その他の API 関数の実装は，性能最適化の行われていない単純なものになっている．

5.1.4　シミュレータ GUI

シミュレータは多岐に渡る状態を長いサイクルを掛けて遷移させる．シミュレータの動作，ひいてはスケジューラの動作を視覚的に認識できるようにするため，簡易的な GUI を作成した．詳細な説明は本稿の主旨と外れるために省略するが，図 4 のウィンドウの左半分がワークフローの状態を，右半分がリソースの状態を表している．

5.2　評価 A: 稼働コストが大きい外部モジュールを複数含むワークフロー

はじめの評価は，*NoKill* スケジューラではオンメモリなデータ処理を達成することが難しく，*PlanningKill* スケジューラが有効に働くワークフローが存在することを示すことを目的としたものである．

ワークフロー定義は図 5 であり，各モジュール *m1* から *m4* は起動時間に 7 ミリ秒を要し，起動中に 200MB のメモリを消費する．それとは別に，オペレータの処理中はデータサイズに比例したサイズのメモリを使用する．このワークフローを，コア数 1，メモリサイズ 1GB の 1 ノードで実行するものとする．3 種類のスケジューラが達成する実行の低遅延性の評価は，ワークフローに流入するデータの流入速度で行う．流入速度が速すぎると，オペレータがブロックを処理するのが追いつかず，多量のメモリがブロックによって消費され，オンメモリでの処理を続けられなくなる．表 1 に，各スケジューリング手法において，オンメモリ処理が可能な最速のデータ流入速度を示す．

モジュールの死活管理を行う *PlanningKill* が最も高性能であり，モジュールを立ち上げたままにしておく *NoKill* が最も性能が悪い結果となった．

NoKill スケジューラを使用している場合，*m1* から *m4* のモジュールが全て立ち上がった後は，多くとも 200MB のメモリしか残らない．従って，少し流速を上げるだけでデータブロックがメモリをはみ出すようになってしまう．

一方で *KillEverytime* スケジューラではモジュールは同時に高々 1 個しか立ち上がらないため，常に 800MB のメモリをデータブロックのオンメモリな処理に使用することができる．しかし，毎回消費することになるモジュールの立ち上げ時間が流速を上げることを妨げている．

PlanningKill スケジューラでは，式 (1) に示した評価値

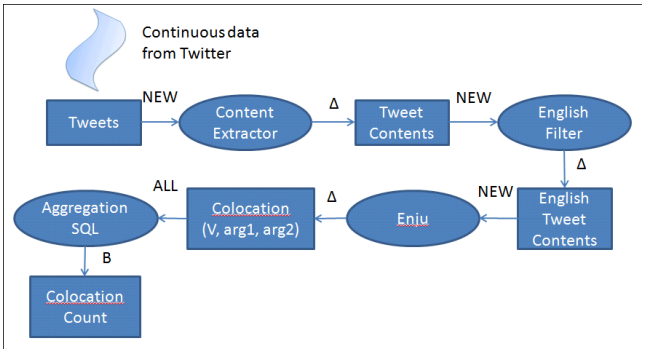

図 6 Twitter から英文コロケーションを解析する処理を模したワークフロー

Fig. 6 Mock workflow to analyze English collocation from Twitter contents

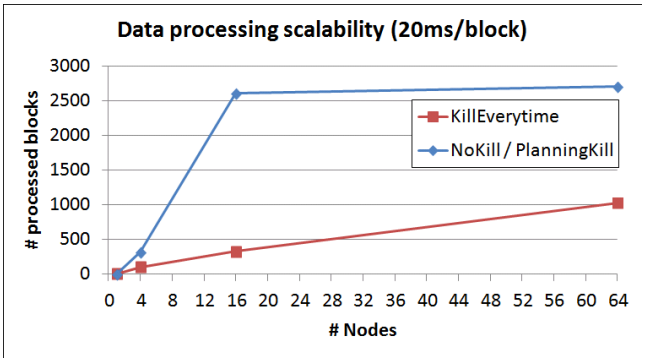

図 7 ノード数に応じた処理ブロック数 (流速 20 ミリ秒/block)

Fig. 7 Number of blocks processed with each number of nodes (Input speed: 20 msec/block)

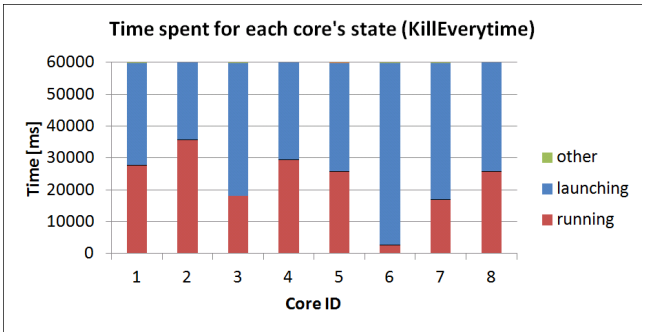

図 8 各コアで各オペレータ状態が費やした時間 (4 ノード 8 コア, *KillEverytime* , 流速 20 ミリ秒/block)

Fig. 8 Time spent for each core's state (4 nodes & 8 cores, *KillEverytime* , input speed 20 msec/block)

が $m1$ については偽で $m2$ から $m4$ については真となった．即ち，ワークフローの 2/5 を占めるモジュール $m1$ のみを立ち上げたままの状態にすることで，モジュールによる起動時間を抑えつつ，メモリ消費も最小限に留めることに成功している．

この評価から，稼働コストが大きい外部モジュールが利用可能なリソースに比べて多くある場合には，モジュールの起動時間と消費メモリを考慮して死活管理を行うスケジューリングが有効であることが示せた．

5.3 評価 B: より実用的なワークフロー

この評価では，単純ながらもより実際的なワークフローアプリケーションに近いものを用いる．使用するワークフローは，モジュールの稼働コストが起動時間の点で大きく，メモリ量の点では比較的小さい．定性的には，このケースではモジュールを常に立ち上げておくのが最良の戦略であるが，式 (1) に示した評価値を用いると， *PlanningKill* スケジューラは *NoKill* スケジューラと同等の動作を示した．

実際に使用したワークフロー定義は図 6 のものである．これは，次のような計算を想定して作られた．

(1) Twitter コーパスからテキストをスクレイピングする．
(2) ツイート本文のみを抽出する．
(3) 英語以外のツイートを排除する．
(4) 英文構文解析器 Enju を適用し，各動詞について，その主語や目的語をリレーションに記憶する．
(5) リレーションから各動詞についての主語・目的語のパターン数 (コロケーション) を計算する．Enju を実体とするオペレータ以外は，起動時間，消費メモリ共にさほど大きくない．

一方で， Enju のオペレータは， Enju の実際の動作に合わせ，起動時間に 8 秒，立ち上げ時のメモリ消費を 100MB とした．

このワークフローは，コア数 2，メモリ量 1GB のノードを 1,4,16,64 台使用して 60 秒間動かすことを想定する．ノード数を増加させると，一定の時間内により多くの流入ブロックを処理できるようになる．

まず，ワークフローへのデータ流入速度を 20 [ミリ秒/block] とした場合における各スケジューラの処理ブロック数を図 7 に示す．ただし，流入ブロックサイズは 1[KB/block] とした．

KillEverytime の結果を見ると，ノード数が多くなるにつれて処理できるブロック数が増加することが分かる．これはノード数を増やすとオペレータを **running** 状態にする CPU 総時間が長くなるためである．しかし，ノード数が 4 台から 16 台に増加するときの処理ブロック数の増加率は 3.2 倍，4 台から 64 台の時は 10.0 倍であり，完全にスケールはしていない．コア p において任意のオペレータを走らせている時間を $T_{run}(p)$ と表記すると，処理可能なブロック数は $\sum T_{run}(p)$ が大きくなるに連れ増加すると考えられる．コア数が少ない時，1 つのオペレータ処理に伴うブロックは増加するため， $T_{run}(p)$ は大きくなる．つまり，コア数が大きくなると相対的に **running** 状態に割ける割合が少なくなるために，理想的なスケールはしないと言える．図 8,9 によって実際にそれを確認することができる．図 8 は 4 ノード使用した場合に各コアが **running**, **launching**, その他の状態のいずれに時間を消費している

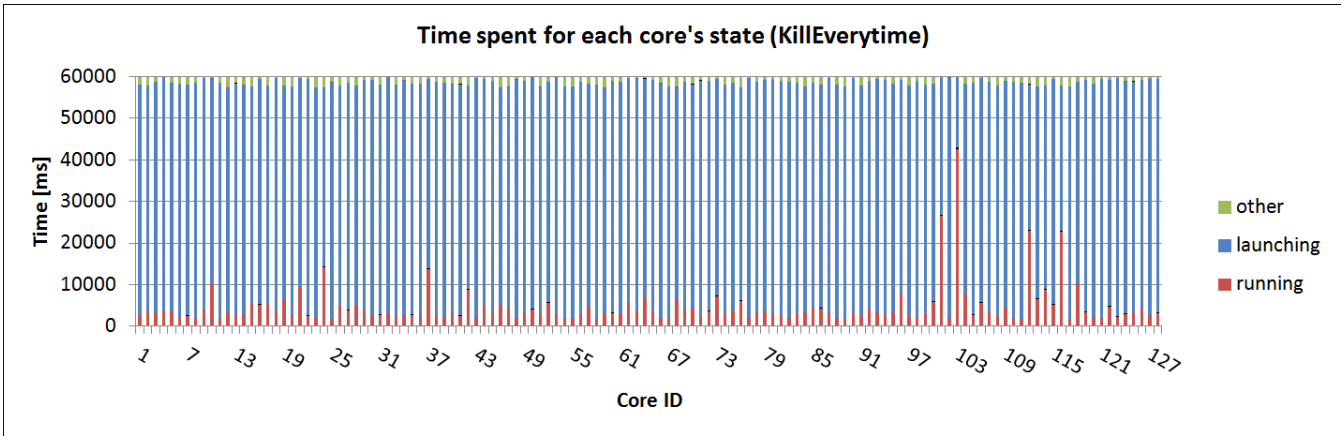

図 9　各コアで各オペレータ状態が費やした時間 (64 ノード 128 コア, *KillEverytime* , 流速 20 ミリ秒/block)

Fig. 9　Time spent for each core's state (64 nodes & 128 cores, *KillEverytime* , input speed 20 msec/block)

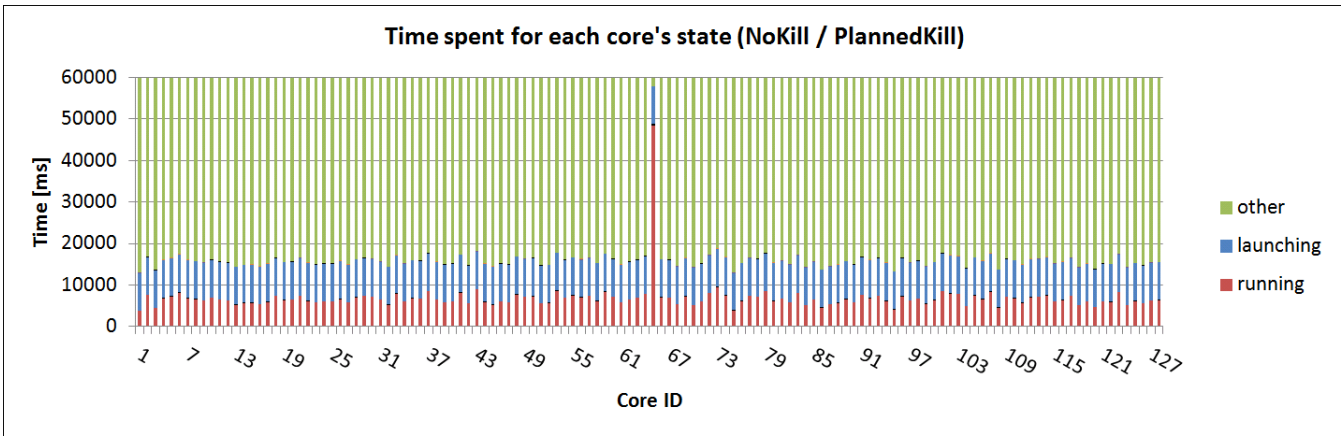

図 10　各コアで各オペレータ状態が費やした時間 (64 ノード 128 コア, *NoKill* , 流速 20 ミリ秒/block)

Fig. 10　Time spent for each core's state (64 nodes & 128 cores, *NoKill* , input speed 20 msec/block)

かを示したものであり，図 9 は 64 ノード使用した場合のものである．4 ノードを使用している時のほうが，16 ノードを使用しているときよりも **running** に割く時間の割合が大きいことが見て取れる．

次に， *NoKill* の結果を見る．このワークフローアプリケーションでは， *PlanningKill* が *NoKill* と全く同じ動作，即ちどのモジュールも立てたままにしておくことを選択したので， *NoKill* 戦略の評価と *PlanningKill* 戦略の評価結果は同一である．*NoKill* では処理可能なブロック数が 2700 個程度で頭打ちになる．ワークフローは 60 秒間動作させたが，どのコアもこの内の 8 秒間は Enju の初回起動に費やしている．従って，実際にブロックの処理に当てられる時間は高々 52 秒だが，ワークフローへのブロックの流入速度と合わせて考えると，処理可能なブロック数の上限は 52 [秒] / 0.020 [秒/block] = 2700 [block] となる．つまり， *NoKill* スケジューリングはノード数を 16 台まで増やした時点で，流入ブロックをほぼ全て詰まりなく処理できるようになっていたことが分かる．実際に図 10 から，多くの時間をモジュール動作やモジュール立ち上げ以外の待ち時間に要していることが分かる．

図 11 には，データの流入速度を 15 [ミリ秒/block] にした場合の *NoKill* スケジューラの処理ブロック数が示されている．ただし， *KillEverytime* 戦略ではこの流入速度においてオンメモリな処理を継続できなかったため，評価対象としていない．流入速度が大きいほうが当然処理ブロック数も増加するが，ここでも上述と同様の理由で処理ブロック数の頭打ちが起きている．

以上の評価 A, B から，次のことが言える．Continuous Workflow 処理をオンメモリに行う場合でも，モジュールがあまりメモリを消費しないのであれば，そのモジュールは常駐化して起動遅延をなくすのが性能向上に繋がる．適切なモジュールの死活管理を行うスケジューラはそのよう

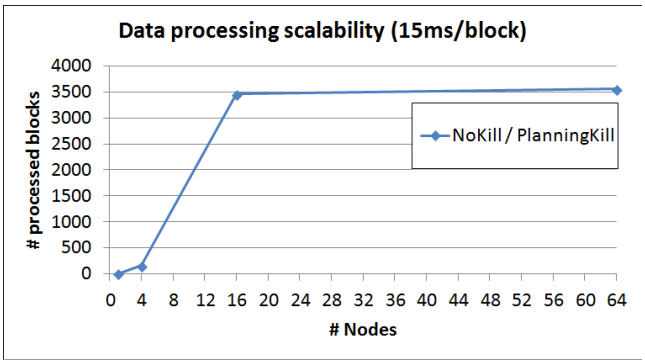

図 11　ノード数に応じた処理ブロック数 (流速 15 ミリ秒/block)
Fig. 11　Number of blocks processed with each number of nodes (Input speed: 15 msec/block)

な常駐化戦略を行うのと同時に，評価 A で見たように，モジュールのメモリ消費が大きな場合は毎回モジュールを終了させることによりブロックのオンメモリな処理を妨げないように動作する．式 (1) で表される *PlanningKill* スケジューラは未だ稚拙なものであるが，モジュールの死活管理が Continuous Workflow の低遅延実行に役立つ可能性を示すものである．

6. まとめ

本稿では，アプリケーションの生産性を高めるためにシェルコマンドのようなモジュールをオペレータとして扱えるような Continuous Workflow をモデル化し，それを対象とするモジュールの稼働コストを考慮したスケジューラを提案した．このスケジューラは未だ設計・実装に不十分な点はあるが，消費メモリの大きいモジュールを多く擁するワークフローではモジュールを処理のたびに終了させる手法やモジュールを常駐化させる手法に比べ，データのより低遅延処理が可能であることをシミュレータにより示した．また，単純ながらより実用的なワークフローアプリケーションでは，提案スケジューラがモジュールを常駐化させる動きを見せ，そのワークフローにおいて高効率で動作することを示した．

今後の課題を述べる．本稿においてはモジュールの死活管理のみに着目したコスト関数を用いてスケジューラを作成したが，それだけでなくデータブロックの配置や，各ノードのリソース消費などの基本的なパラメタを含めて低遅延スケジューラを再考察する必要がある．また，スケジューリング自体に要する時間の評価は行っていないので，Continuous Workflow 処理系の開発を進め，それも含めた評価をする必要がある．更に，本研究で着目しているモジュールの死活管理を取り入れたスケジューリングが有効であるような実アプリケーションの十分な調査も必要である．

参考文献

[1] Apache Software Foundation. Hadoop. http://hadoop.apache.org.
[2] Apache Software Foundation. Oozie: Hadoop workflow system. http://yahoo.github.com/oozie/.
[3] Apache Software Foundation. Pig. http://pig.apache.org/.
[4] Cascading. http://www.cascading.org/.
[5] Enju - A fast, accurate, and deep parser for English. http://www.nactem.ac.uk/enju/.
[6] EsperTech: Event Stream Intelligence. http://www.espertech.com/.
[7] Fluentd: Open Source Log Management. http://fluentd.org/.
[8] Jubatus : Distributed Online Machine Learning Framework. http://jubat.us/en/.
[9] S4: Distributed Stream Computing Platform. http://incubator.apache.org/s4/.
[10] YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[11] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 1:1–1:13, New York, NY, USA, 2012. ACM.
[12] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.
[13] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.
[14] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 838–849. VLDB Endowment, 2003.
[15] Olston Christopher. Modeling and Scheduling Asynchronous Incremental Workflow. Technical report, Yahoo! Research, 2011.
[16] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi, editors. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer, 2012 edition, 6 2013.
[17] Yi Gu and Qishi Wu. Maximizing Workflow Throughput for Streaming Applications in Distributed Environments. In *ICCCN*, pages 1–6, 2010.
[18] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[19] Bertram Ludäscher, Ilkay Altintas, Shawn Bowers, Julian Cummings, Terence Critchlow, Ewa Deelman, David De Roure, Juliana Freire, Carole Goble, Matthew Jones, Scott Klasky, Timothy McPhillips, Norbert Podhorszki, Claudio Silva, Ian Taylor, and Mladen Vouk. Scientific Process Automation and Workflow Management. In Arie Shoshani and Doron Rotem, editors, *Scientific Data Management*, Computational Science Series, chapter 13. Chapman & Hall, 2009.
[20] Christopher Olston, Greg Chiou, Laukik Chitnis,

Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: Continuous Pig/Hadoop Workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1081–1090, New York, NY, USA, 2011. ACM.

[21] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[22] Masahiro Tanaka and Osamu Tatebe. Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 356–359, New York, NY, USA, 2010. ACM.

[23] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun ichi Tsujii. Design and Implementation of GXP Make - A Workflow System Based on Make. In *eScience*, pages 214–221. IEEE Computer Society, 2010.