QueueLinker: パイプライン型アプリケーションのための分散処理フレームワーク

上田 高徳[(1),(2)] 片瀬 弘晶[(1)] 森本 浩介[(1)] 打田 研二[(3)] 油井 誠[(4)] 山名 早人[(5),(6)]

(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) 早稲田大学メディアネットワークセンター 〒169-8050 東京都新宿区戸塚町 1-104
(3) 早稲田大学 基幹理工学部 コンピュータ・ネットワーク工学科 〒169-8555 東京都新宿区大久保 3-4-1
(4) 日本学術振興会特別研究員 (PD) 〒102-8471 東京都千代田区麹町 5-3-1
(5) 早稲田大学 理工学術院 〒169-8555 東京都新宿区大久保 3-4-1
(6) 国立情報学研究所 〒101-8430 東京都千代田区一ツ橋 2-1-2

E-mail: t-ueda@fuji.waseda.jp, {katase, morimoto, k_uchida, makoto-y, yamana}@yama.info.waseda.ac.jp

あらまし 本論文ではパイプライン処理に特化した分散処理フレームワーク QueueLinker について述べる．QueueLinker を用いるプログラマは，アプリケーションのモジュールを実装し，モジュール間のデータの流れを指定することで，分散処理を行うことができる．Hadoop のような MapReduce 型フレームワークでは，Map と Reduce の度に発生するストレージアクセスやネットワーク通信がボトネックとなり得るが，QueueLinker は分散処理の戦略をプログラマが詳細に制御することができるため，パイプライン処理や完全なオンメモリ処理を行うことでボトルネックを回避することが可能である．計算機 28 台の環境において，Web クローラのログデータ(3.68TB)に記録された多数の重複を含む約 462 億個の URL に対して一意な ID を振るナンバリング処理を QueueLinker と Hadoop で比較したところ，QueueLinker は Hadoop よりも 3.59 倍高速に処理することができた．

キーワード 分散処理，分散処理フレームワーク，データ処理

QueueLinker: A Distributed Framework for Pipelined Applications

Takanori UEDA[(1),(2)] Hiroaki KATASE[(1)] Kousuke MORIMOTO[(1)] Kenji UCHIDA[(3)]
Makoto YUI[(4)] and Hayato YAMANA[(5),(6)]

(1) Graduate School of Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
(2) Media Network Center, Waseda University, 1-104 Totuka-cho, Shinjuku-ku, Tokyo 169-8050 Japan
(3) Faculty of Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
(4) JSPS Research Fellow (PD), 5-3-1 Koujimachi, Chiyoda-ku, Tokyo 102-8471 Japan
(5) Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
(6) National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan

E-mail: t-ueda@fuji.waseda.jp, {katase, morimoto, k_uchida, makoto-y, yamana}@yama.info.waseda.ac.jp

Abstract This paper describes QueueLinker, a distributed computation framework for pipeline processing. QueueLinker allows programmers to realize distributed computation by implementing application modules and specifying data flows between the modules. MapReduce like framework, such as Hadoop, causes bottlenecks of large network traffic and storage accesses required in every “Map” and “Reduce.” Programmers who use QueueLinker are able to control the distribution strategy completely and achieve both pipeline parallelism and on-memory processing to avoid the bottlenecks. In the experiment, 46.2 billions URLs, including many duplicated URLs, in a Web crawler's log data (3.68TB) were numbered uniquely by QueueLinker and Hadoop with 28 computers. As a result, QueueLinker outperformed Hadoop more than 3.59 times the execution time.

Keyword Distributed Computing, Distributed Computation Framework, Data Processing

1. はじめに

クリックストリームやセンサデータに代表される大量のデータを処理するために，分散処理への興味が高まってきている．これまで，分散処理は大量の計算機を所有できる企業や研究機関にしか馴染みがなかったが，いまではクラウドサービスが個人や中小企業にも分散処理の可能性をもたらしている．しかし，分散処理を効率的に行うためには，計算機の故障・ネット

ワーク障害・データ配置・スケジューリング，といった分散環境特有の問題に対応しなければならないため，プログラマの負担が大きくなり得る．MPI [1]のようなライブラリは分散アプリケーションを作成するための補助になるが，通信方法の検討や冗長性の実現，パフォーマンスチューニングはプログラマの責任となる．

このため，分散処理に関わる通信や冗長性の実現を自動化する分散フレームワークの開発が行われている[2,3,4]．分散フレームワークでは，分散処理をプログラマに意識させずにアプリケーションの作成を可能にするため，様々なプログラミングモデルを採用している．例えば分散フレームワークのひとつである MapReduce[2]では，Key-Value ペアを生成する Map 関数と，Key ごとに集約演算を行う Reduce 関数を用いる集約型の演算を強制することで，分散処理を実現している．

MapReduce は分散フレームワークとして代表的であるが，問題点も指摘されるようになってきている[5]．その 1 つに，Map と Reduce ごとにデータがストレージに書きこまれるため，I/O ボトルネックが大きいことがある．これは，Map が終了するまで Reduce が，Reduce が終了するまで次の Map 処理が開始できないということであり，複数の処理をパイプライン的に実行する必要があるアプリケーションに MapReduce が不向きということである．

本論文では，パイプライン処理に適した分散フレームワーク QueueLinker について述べる．QueueLinker はマルチスレッドプログラミングにおける定石である Producer-Consumer モデルをプログラミングモデルとして採用する．QueueLinker では，プログラマが分散の戦略を柔軟に制御することができ，MapReduce が苦手とするパイプライン処理が必要なアプリケーションにも適している．約 462 億個の実 URL への一意な ID の割り付けを行う実験により，提案手法が競合手法に対して 3.59 倍の性能が得られる場合があることを示す．

本論文の構成は次のとおりである．2 節において，QueueLinker のプログラミングモデルについて概要を述べる．3 節において関連研究と QueueLinker を比較する．4 節で提案手法の評価を行い，5 節でまとめる．

2. QueueLinker

本節では，分散フレームワーク QueueLinker について説明する．QueueLinker はプログラムの記述に Producer-Consumer モデルを採用する．これは，マルチスレッドプログラミングにおける，Producer-Consumer モデルと同様のもので，アプリケーションモジュールがキューを介して通信する．

2.1. Producer-Consumer モデル

Web クローラを例として Producer-Consumer モデルを説明する．Web クローラにおける HTML のダウンロードと HTML 重複削除は独立した処理であるので並列実行できる．ここで，1 つのスレッドコンテキスト上で，ダウンロードと重複削除を順に行ったとする．この場合，ダウンロードと重複削除という異なる作業が 1 つのコンテキスト内に存在することになり，プログラムが煩雑になる．また，重複削除処理は HTML ページのダウンロードが完了するのを待たなくてはならない．ダウンロードの場合，Web サーバからのレスポンスに遅延があるため，マルチスレッド処理により多くのリクエストを並行で送信した方が，ダウンロード処理のスループットを向上できる．つまり，スレッド数を増やすことで，wait 時間を償却できる．一方で，重複削除はダウンロードよりも CPU 支配的な処理である．この場合は，CPU のコア数（あるいはハードウェアスレッド数）よりもスレッド数を増やすと，コンテキストスイッチが増え，オーバーヘッドとなる．

Producer-Consumer モデルは前記の問題を解決するための，マルチスレッドプログラミングにおける定石である．Producer-Consumer モデルでは，処理を細かい単位に分割してプログラミングを行い，それぞれの処理に専用のスレッドを割り当てる．ある処理が完了すると，次の処理に必要な情報をキューに追加する．次の処理を行うスレッドはキューから情報を取り出し実行する．図 1 は Producer-Consumer モデルで記述した Web クローラの例である．この例では，HTML ダウンロードモジュール（図中のm_5）が 1 つの Web ページのダウンロードを完了すると，HTML データと URL がキュー（図中の Q）に追加され，HTML 重複削除モジュール（図中のm_6）は，キューから HTML と URL を取得して処理を行う．スレッド間の接点がキューであるため，HTML ダウンロードと HTML 重複削除にはそれぞれ任意のスレッド数を容易に割り当てることができる．また，キュー容量の範囲内でモジュール間の速度差を吸収できるため，HTML 重複削除の動作速度に影響されず，HTML ダウンロードを行うことができる．

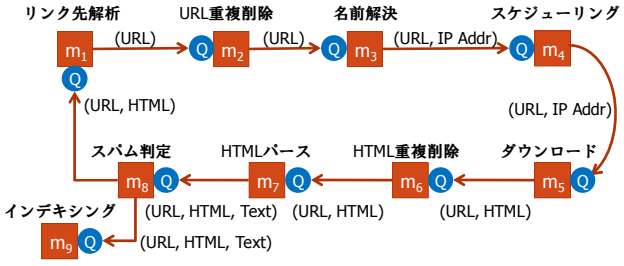

図 1. Producer-Consumer モデルで示した Web クローラ

2.2. QueueLinker を用いたアプリケーションの作成

QueueLinker は Producer-Consumer モデルに基づいたモジュール間通信を行う．ユーザが作成するモジュール間はキューで通信され，ユーザが作成するアプリケーションロジックが計算機間の境界を意識することなしに分散処理を行うことが可能である．プログラマは図 1 の矩形内のモジュールのアプリケーションロジックを作成し，モジュール間の接続関係を指定する．モジュールはデータを送受信するが，その通信方法は QueueLinker によりアプリケーションからは隠蔽される．すなわち，データはオンメモリで次のモジュールへ渡されるかもしれないし，ネットワークを介して送信されるかもしれない．この隠蔽により，QueueLinker は利用可能な計算ノード内にモジュールを自動配置し，分散処理を実現する．

QueueLinker は Java で実装されており，分散の単位となるモジュールには，1 つの Java クラスが相当する．モジュールが入出力するデータをアイテムと呼ぶ．各モジュールは，モジュールの接続関係を表した論理グラフの頂点 (Vertex) となることができる．グラフの辺 (Edge) はモジュール間の接続を示す．

QueueLinker を用いて分散処理を行うには，QueueLinker が定義したクラスを継承してモジュールを実装し，モジュール間のデータの流れを論理グラフにより表現する．QueueLinker は指定された論理グラフに基づいて，利用可能な計算機にモジュールを分散配置し実行する．言い換えると，QueueLinker は論理グラフを入力として受け取り，実際の計算ノード上に物理グラフとして展開する．

2.2.1. Vertex と Edge のモード

QueueLinker のモデルにおいては，どの計算ノードにどのモジュールを何スレッド配置するかが，アプリケーション全体の性能を大きく左右する．また，モジュールによっては特定のノードで実行されなければならない場合がある．QueueLinker ではこれらの分散に関わる戦略はユーザが自由に制御することが可能になっている．論理グラフの Vertex ごとに，どのノードに設置するか，何スレッドを割り当てるかといったオプションを指定できる．例えば，全ノードに分散する場合は FullDistribution モード，特定のノードで実行する場合は FixNode モードを指定する．

Edge には，Vertex 間のデータ通信経路，入力データの集約方法などを指定する．Edge のモード例を表 1 に示す．例えば StreamMergeSort モードは，分散しているモジュールの出力を集約してソート順に集める場合に利用する．FullDistribution モードの Vetrtex v_1 と FixNode モードの Vertex v_2 を StreamMergeSort モードの Edge で接続すると，全ノード上にある v_1 の出力がマージソートされた順で v_2 に入力される．なお，StreamMergeSort モードは，各モジュール出力の先頭のみをチェックしてマージソートするため，各ノードの v_1 それぞれはソート順でアイテムを出力しなければならない．この他，Edge を LocalOnly に設定した場合，同一ノード内で転送されることが保証されるため，ネ

```
public class PrimeBigIntegerGenerator
             extends OutputOnlyModule<BigInteger> {
  @Override
  public void launch(OutputOnlyMediator<BigInteger> mediator) {
    Random rand = new Random(System.nanoTime());
    while (true) {
      BigInteger bigInt = new BigInteger(1024, 1000000, rand);
      mediator.putNextItem(bigInt);
    }
  }
}
```

リスト 1. 疑似素数乱数列生成モジュール

```
public class BigIntegerPrinter
             extends InputOnlyModule<BigInteger> {
  @Override
  public void launch(InputOnlyMediator<BigInteger> mediator) {
    BigInteger value = null;
    while ((value = mediator.getNextItem()) != null) {
      System.out.println(value.toString());
    }
  }
}
```

リスト 2. 標準出力への出力モジュール

```
VertexGraph graph = new VertexGraph();
OutputOnlyVertex<BigInteger> outVertex
   = new OutputOnlyVertex<BigInteger>(PrimeBigIntegerGenerator.class);
InputOnlyVertex<BigInteger> inVertex
   = new InputOnlyVertex<BigInteger>(BigIntegerPrinter.class);
inVertex.fixNodeMode(true, "masterNode");
outVertex.fullDistributionMode(true);
graph.makeEdge(outVertex, inVertex);
```

リスト 3. 論理グラフの記述

表 1. Edge のモード例

モード	機能
RoundRobin	アイテムごとにラウンドロビンで分散させる
LocalOnly	同一計算ノード内でデータを送受信する
StreamMergeSort	入力をマージソートする

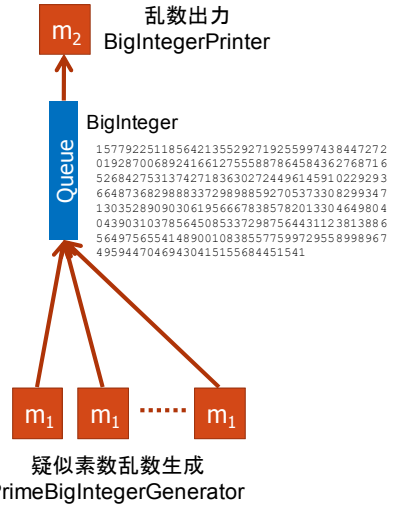

図 2. プログラム例のモジュール接続関係

ットワーク通信量の削減に効果がある．これらの機能は 4 節の評価実験で用いている．

2.2.2. プログラム例

Java API にある BigInteger を利用して巨大な疑似素数乱数列を出力し続けるプログラム例を示す．リスト 1 が疑似素数の生成モジュール，リスト 2 が標準出力に出力するモジュールである．リスト 3 でモジュールの接続関係を示す論理グラフを構築している．論理グラフは図 2 のようになる．疑似素数生成は独立した処理なので，分散することでより多くの疑似素数を生成できる．そのためモジュールm_1 は FullDistribution モードとする．特定のノード上に出力するために，モジュールm_2 は FixNode モードとしている．

3. 関連研究と QueueLinker

分散フレームワークとして Google MapReduce[2]とそのオープンソース実装である Apache Hadoop[6]が代表的である．MapReduce を用いるプログラマは，Key-Value ペアを生成する Map 関数と，Key ごとに集約演算を行う Reduce 関数を実装する．Map が出力した Key-Value ペアは分散ファイルシステムに書き込まれ，Shuffle フェーズにおいて Key に基づいてソートされる．そして，まとめた同一の Key ごとに Reduce 関数を適用して最終結果を生成する．Map 関数による Key-Value の生成は入力データを分割することで同時実行が可能であり，異なる Key に対しては異なる計算ノードで Reduce 処理を行うことが許されるため，自動的な分散処理を実現できる．

MapReduce の弱点として，データ構造にスキーマがないためデータの入出力はプログラマが責任を負わなければならないこと[5]，Shuffle フェーズにおけるネットワーク転送やストレージアクセスがボトルネックになること[7]，などが指摘されている．また，並列データ処理は並列データベースの分野で長く研究されており，データベースに対する命令，すなわち SQL で記述できるならば，並列データベースの方が高速に処理できるという主張がある[5]．リスト 4 のような給与の高い順にナンバリングするといった処理は，MapReduce のような Key に基づいた集約演算に適さない[5]．

このような MapReduce の弱点を補う試みとして，PostgreSQL と Hive[8]を連携することで関係データベースの強みを活かした HadoopDB[9]，QueueLinker に類似したモデルを採用した Microsoft Dryad [3,10,11]，PageRank のようなグラフに対する演算を分散処理するための PEGASUS[7]などがある．Dryad は我々の QueueLinker に類似したプログラミングモデルを採用している．Dryad はモジュール間のデータの流れを無閉路有向グラフ(Direct Acyclic Graph)として記述する．Dryad はデッドロックの危険を避けることを主要な理由に，グラフに閉路を持つことを許していない一方で，我々の QueueLinker は閉路を持つ有向グラフを入力できるという違いがある．このため，QueueLinker ではデッドロックの危険を排除することはプログラマの責任であるが，図 1 のクローラのように閉路があるアプリケーションの実現が可能であるという利点がある．また，MapReduce では Map 関数が終了してから Reduce 関数が動作するためパイプライン的な動作が困難であるが，QueueLinker は，キューにアイテムが追加されると，モジュールがすぐに動作するため，パイプライン処理を行うアプリケーションに適している．

```
SELECT Emp.name, Emp.salary,
       RANK() OVER (ORDER BY Emp.salary)
FROM Employees AS Emp
```

リスト 4．MapReduce での実行が困難な SQL クエリ[5]

4. 評価実験

約 462 億個の URL に対してソート順にユニークな連続 ID を振るナンバリング処理を QueueLinker と Hadoop を用いて実装し，実行時間を比較した．実験データには e-Society プロジェクト[12]による Web クローリングログを用いた．ログファイルには収集した Web ページごとにクローリング時刻・文字コード・リンク先 URL といった情報がテキスト形式で格納されている．ログファイルは容量や Web ページの言語に基づいて分割され，それぞれ gzip 圧縮されている．本実験では，事前に選んだ一部のログファイルを処理の対象とし，リンク先として出現する URL 全てに対してナンバリングを行った．Web ページからのリンク先 URL であるから重複がある．すなわち本実験では，gz ファイルを展開してログファイルからリンク先 URL を抽出し，重複除去して連続 ID を振り，最終結果を単一ファイルにまとめる処理の実行時間を測定した．

この処理を関係データベースで実現するならば，リンク先 URL のテーブル Urls を作成したのち，リスト 5 のクエリを実行することに相当する．これは MapReduce で実行が難しいとされるリスト 4 のクエリに似ているが，重複を削除する DISTINCT 句とソート順に出力する ORDER BY が追加され，分析関数の

```
SELECT DISTINCT DENSE_RANK() OVER
(ORDER BY Urls.url), Urls.url
FROM Urls
ORDER BY Urls.url
```

リスト 5．実験で行う処理を表現した SQL クエリ

RANK は，同じ URL があっても順位が不連続にならない DENSE_RANK になっており，より複雑である．

4.1. 実験環境と実験データ

実験ではマスタノード 1 台と，計算ノード 27 台を用いた．各ノードの性能を表 4 に示す．マスタノードに gzip 圧縮されたクローリングログが保存されている．実験に使用したログは 4897 ファイル，合計容量は圧縮状態で 531GB であり，展開後のログ総容量は 3.68TB になる．ログファイルから抽出される URL は 461 億 9076 万個であり，ASCII 換算での容量は 2.36TB である．このうち 93.2%が重複 URL なため，最終的には 31 億 4286 万個の URL を出力することになる．最終結果はマスタノード上の単一ファイルにまとめる．Hadoop のバージョンは 0.19.1 であり，データのレプリケーションは無しの設定としている．

4.2. 実装の詳細

前述の処理を行うために QueueLinker と Hadoop で行った実装を説明する．

4.2.1. QueueLinker を用いた実装

QueueLinker を用いた実装では，ログデータを gz ファイルのまま計算ノードに分配した上で，各ノードで gz ファイルの展開し，マージソートを行った．メモリ量の関係で一括ソートができないためローカルディスクを利用した外部（マージ）ソートを行う．メモリに収まる個数の URL を平衡二分木（TreeSet）を用いてソートした上で，ソート済みの分割ファイル作成する．これらの分割ファイルをマージソートして，マスタノードに集約しつつナンバリングすることで最終結果を得る．

以上の処理を実現するため，表 2 の 5 つのモジュールを作成し，図 3 のように接続した．表 3 は Edge のモードである．モジュールm_1 と m_2 間の Edge は，RoundRobin モードで，gz ファイルごとにラウンドロビンでノードに分散される．モジュールm_2 と m_3 間の Edge は LocalOnly モードで，計算ノードのモジュールm_2 が出力する TreeSet インスタンスが，そのノードのm_3 に直接渡される．これは，ネットワーク経由の転送を避けるためである．ここまでがステージ 1 であり，全ての処理が完了後，モジュールm_4 と m_5 が動作を開始する．m_3 が作成した分割ファイルごとにモジュールm_4 が 1 スレッド生成され，分割ファイルから URL を順に読み込み出力する．モジュールm_4 と m_5 間の Edge が StreamMergeSort モードなので，QueueLinker は全てのモジュールm_4の出力に対してマージソートを行い，モジュールm_5に渡す．すなわち，モジュールm_5 にはマージソートされた順に入力されるため，プログラマがマージソートを直接的に実装す

表 2. QueueLinker を用いた実装で作成したモジュール一覧

	モジュール	Input	Output	配置オプション	機能
Stage1	m_1	なし	byte[] (gz file)	マスタノードに 1 スレッド	gz 圧縮のログファイルをローカルファイルシステムから検索し，そのまま出力する．
	m_2	byte[] (gz file)	TreeSet	計算ノードに 1 スレッドずつ	gz ファイルを先頭から読み込みながら展開し，リンク先 URL を抽出し，TreeSet に追加する．一定個数の URL を追加したら TreeSet を出力する．新しい TreeSet を new し処理を続ける．
	m_3	TreeSet	なし	計算ノードに 1 スレッドずつ	入力された TreeSet を Iteration し，ソート順に全要素を TreeSet ごとに異なる分割ファイルに書き込む．TreeSet は破棄する．
Stage2	m_4	なし	String	分割ファイル 1 つごとに 1 スレッド	各分割ファイルの先頭から順に URL を出力する．
	m_5	String	なし	マスタノードに 1 スレッド	入力順に URL をファイルに書き込む．

表 3. Edge のモード

Edge	モード
$m_1 \Rightarrow m_2$	RoundRobin
$m_2 \Rightarrow m_3$	LocalOnly
$m_4 \Rightarrow m_5$	StreamMergeSort

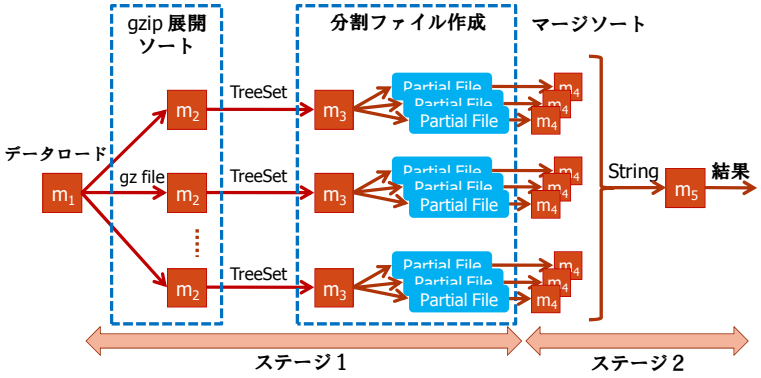

図 3. QueueLinker による論理グラフ

る必要はない．

以上のように QueueLinker ではユーザが分散実行プランを考え，モジュールを実装し，モジュールの配置方法を設定する．プログラマの負担は増える可能性があるが，Hadoop よりも細かい制御が可能である．

4.2.2. Hadoop を用いた実装

Hadoop では Map 関数で生成した Key-Value ペアは Key 順にソートされて Reduce 関数に入力される．この性質を利用すると，URL を Key として Map し，Reduce 関数を単一インスタンスで実行すれば目的の処理が可能である．しかし，この方法ではディスク容量不足が発生して正常に処理を完了することができなかった．このディスク容量不足は，Reduce が単一の計算ノードで実行される場合，全 URL のデータが 1 つの計算ノードに書き込まれてしまうために発生する．

そこで，Hadoop でも QueueLinker と同様に，外部ソートの戦略を取るための実装を行った．Map 関数はデータを読み込み，URL を Key として出力する．出力は，Key (URL) から求めたハッシュ値の Reduce 数に対する剰余（以下単にハッシュ値剰余）に基づいて計算ノードローカルのファイルに分割して書き込む．この分割ファイルは Hadoop によりソートされたあと，ハッシュ値剰余ごとにマージソートされる．そして，同一のハッシュ値剰余を持つ URL は同一の Reduce 関数に入力され重複削除される．Reduce 関数は，各計算ノードに 1 つのソート済みファイルを生成する．最後に，各計算ノードのソート済みファイルをマスタノードに転送し，マスタノード上でマージソートを行う．

4.3. 実験結果

実行時間を表 5 に示す．合計時間では QueueLinker が Hadoop よりも 3.59 倍高速な結果となっている．Hadoop はパイプライン処理が行えないため，ファイルの転送（表 5 の「ファイル転送」）が完了してからソート作業（表 5 の「分割ソート」）を行う．しかし，QueueLinker では gz ファイルを転送すると同時に，先頭からシーケンシャルに展開し，ソートを並行して行う．すなわち，QueueLinker ではネットワーク経由でデータが到着する待ち時間を利用してパイプライン的に展開を行っているため，その分高速化することができる．最終のマージソートも同様で，QueueLinker はデータの集約とマージソートを並行して行うことができるが，Hadoop はパイプライン処理ができないため，データをマスタノードにコピー（表 5 の「データの集約」）したあと，マージソートを行うことになる（表 5 の「マージソート」）．以上のようにパイプライン処理が行える場合に QueueLinker を用いると，モジュールを作成して論理グラフを適切に設定すれば，Hadoop よりも高速な処理を実現することができる．以上の結果より，上記の設定において我々の提案手法に確かな優越があることを確認した．

表 4. 実験環境

	マスタノード	計算ノード
CPU	Xeon 5160	Core 2 6700
RAM	16GB	4GB
HDD	LVM RAID 8TB	SATA 500GB
NETWORK	1000BASE-T	

表 5. QueueLinker と Hadoop での実験結果

	QueueLinker	Hadoop
ファイル転送	3h18m	1h43m
分割ソート		14h25m
データの集約	4h17m	1h06m
マージソート		10h01m
計	7h35m	27h15m

5. おわりに

本論文では，我々が開発している分散フレームワーク QueueLinker について述べた．QueueLinker はプログラミングモデルとして Producer-Consumer モデルを採用し，パイプライン型処理を行うアプリケーションの分散実行処理をサポートする．QueueLinker の特徴として閉路をもつグラフの実行をサポートすることがある．この利点により，図 1 のような Web クローラをはじめ，継続的な処理を必要とする継続的クエリ処理 (continueus query processing) やストリーム処理にも応用が可能である．

約 462 億個の URL に対するユニーク ID のナンバリング処理を行う評価では，MapReduce のフリー実装である Apache Hadoop よりも 3.59 倍の実行速度を得られることを確認した．これは，データの転送・展開・ソートといった動作をパイプライン実行できることの効果である．

QueueLinker の今後の課題として障害対応の自動化と自動スケジューリング機能の実現がある．障害対応の自動化は分散フレームワークには必須である．加えて，各モジュールの消費リソース量やアイテム入出力量をモニタリングすることでモジュールの性質を把握し，論理グラフで指定された制約の下で，スループットを最大化するようにモジュールの配置や割り当てスレッド数を自動スケジューリングすることを目指している．

謝辞

本研究の一部は，ＪＳＴ情報基盤戦略活用プログラム「多メディア Web 解析基盤の構築及び社会分析ソフトウェアの開発」及び科学研究費補助金「挑戦的萌芽研究（21650010)」の助成による．

文　献

[1] Message Passing Interface (MPI) Forum Home Page, http://www.mpi-forum.org/, visited Feb. 2010.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (OSDI), pp.137-149, San Francisco, US-CA, Dec. 2004.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (EuroSys), pp.59-72, Lisbon, Portugal, Mar. 2007.

[4] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," In *Proceedings of the 35th International Conference on Management of Data* (SIGMOD), pp.1099-1110, Vancouver, Canada, Jun. 2008.

[5] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," In *Proceedings of the 35th International Conference on Management of Data* (SIGMOD), pp.165-178, Providence, US-RI, Jun. 2009.

[6] Welcome to Apache Hadoop, http://hadoop.apache.org/, visited Feb. 2010.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations," In *Proceedings of the 9th International Conference on Data Mining* (ICDM), pp.229-238, Miami, US-FL, Dec. 2009.

[8] Welcome to Hive!, http://hadoop.apache.org/hive/, visited Feb. 2010.

[9] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," In *Proceedings of the 35th International Conference on Very Large Data Bases* (VLDB), Lyon, France, Aug. 2009.

[10] Dryad, http://research.microsoft.com/en-us/projects/dryad/, visited Feb. 2010.

[11] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pp.1-14, San Diego, US-CA, Dec. 2008.

[12] e-society project, http://www.yama.info.waseda.ac.jp/e-society/, visited Feb. 2010.