MapReduce: Simplified Data Processing on Large Clusters

MapReduce: 大規模クラスタ上の簡単データプロセッシング

Jeffrey Dean, Sanjay Ghemawat

MapReduce はプログラミングモデルであり，また様々な実際のタスクにおける大規模なデータセットを処理し，生成するための実装である．ユーザが map 関数と reduce 関数の演算処理を指定すると，下位のランタイムシステムが自動的に演算処理を大規模クラスタ上で並列化し，マシンの障害を処理し，ネットワークとディスクを効率的に使うためにマシン間通信のスケジュールを行う．MapReduce はプログラマにとって使いやすいシステムで，Google では過去 4 年間に 1 万以上の異なった MapReduce プログラムが実装されている．毎日平均 10 万の MapReduce ジョブが Google のクラスタ上で実行されており，1 日あたり合計 20 ペタバイト以上のデータを処理している．

1 はじめに

MapReduce の開発より前は，Google のエンジニアたちは大量の未加工データの処理を行うために数百もの専用の演算処理を実装していた．未加工データとは，例えばクロールしたドキュメントや，Web リクエストログなどである．それらのデータから，転置インデックス，Web ドキュメントのグラフ構造の様々な表現，ホストごとにクロールしたページ数，ある 1 日の中で最も頻出したクエリのセット等の様々な種類の派生データを計算していた．このような計算のほとんどは理論的に簡単な計算だ．しかし，入力データは通常大きく，妥当な時間で計算を終わらせるためには，数百～数千のマシンで分散処理を行わなければならない．どのように並列計算を行うのか，どのようにデータ分散を行うのか，どのようにエラー処理を行うのかという問題に対処するための複雑なコードが大量に加えられた結果，元のシンプルな演算処理が分かりにくいものとなった．

この複雑さへの対応として，我々は新たな抽象化の設計を行った．実行しようとしていた簡単な計算を表現でき，並列化，耐故障性，データ分散，負荷分散の面倒な詳細をライブラリの中に隠蔽する．我々の抽象化は，Lisp や他の多くの関数型言語に表れる map と reduce プリミティブから着想を得ている．我々は，我々の計算の大部分が，入力の各論理レコードに map 操作を適用し中間 key/value ペアの集合を計算し，それから同じ key を共有するすべての値に対する reduce 処理を適用し派生データを得ていることに気がついた．ユーザ指定の map/reduce 操作による関数型モデルの利用により，大規模計算の並列化が簡単に行え，さらに耐故障性の主要な仕組みとしての再実行が可能となる．

本研究の主な貢献は，大規模計算の自動並列化と分散を可能とする単純かつ強力なインターフェースと，コモディティ PC の大規模クラスタ上で高性能を達成するそのインターフェースの実装である．このプログラミングモデルは，マルチコアマシンでの並列化にも使用することができる．

第 2 節では基本的なプログラミングモデルを述べ，いくつかの例を挙げる．第 3 節では，クラスタベースのコンピューティング環境へ向けた MapReduce インターフェースの実装について述べる．第 4 節では我々が見つけた役に立つプログラミングモデルのいくつかの拡張機能を述べる．第 5 節では様々なタスクにおける性能計測の結果を報告する．第 6 節では，Google 内での MapReduce の利用について，特にプロダクション用のインデクシングシステムの書き換えのための基盤として利用した時の経験を述べる．第 7 節では，関連研究と今後の課題について議論する．

2 プログラミングモデル

計算タスクは key と value のペアのセットを入力とし，key と value のペアのセットを出力として作り出す. MapReduce ライブラリのユーザは，map と reduce という 2 つの関数を使って計算タスクを表現する．

map はユーザによって書かれる．入力ペアのセットから，中間 key/value ペアを生成する．MapReduce ライブラリは同じ中間 key I に関連付けられたすべての中間 value をまとめ，それらを reduce 関数へ渡す．

reduce 関数もまたユーザによって書かれる．中間 key I と，その key に関連付けられた value のセットを受け取る．そして value のセットをマージし，より小さな value のセットを形成する．基本的には reduce ごとに 0 個か 1 個の value が生成される．中間 value はイテレータを用いてユーザの reduce 関数に提供される．これにより，メモリ内に収まりきらないとても多くの value のリストを扱うことができる．

2.1 例

大規模な文書コレクションから，各単語の出現頻度を数える問題について考える．ユーザは次の疑似コードのように書くだろう．

map 関数では各単語と出現カウント（この例の場合，すべて 1）のペアを生成する．reduce 関数では，特定の単語で生成された出現カウントを合計する．

また，ユーザは入力，出力ファイル名とオプショナルなチューニングパラメータを埋めることにより MapReduce 仕様オブジェクトのコードを書く．ユーザはその仕様オブジェクトを渡し，MapReduce 関数を呼び出す．ユーザのコードは（C++ で実装された）MapReduce ライブラリにリンクされる．我々の MapReduce の原論文には，この例についての全プログラムリストが含まれている[8].

```
map(String key, String value):
  // key: 文書名
  // value: 文書の内容
    for each word w in value:
      EmitIntermediate(w, “1”);
reduce(String key, Iterator values):
  // key: 単語
  // values: 出現カウントのリスト
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Google では MapReduce を使って 1 万以上の異なるプログラムが実装されている．それには，大規模なグラフを処理するアルゴリズム，テキスト処理，データマイニング，機械学習，統計的機械翻訳，その他のさまざまな分野が含まれている. MapReduce の特定のアプリケーションの詳細については別の論文にある[8, 16, 7].

2.2 型

前述の疑似コードは文字列の入出力の観点で書かれているが，概念的には，ユーザ定義の map 関数と reduce 関数は関連付けられた型を持っている．

map	(k1,v1) → list(k2,v2)
reduce	(k2,list(v2)) → list(v2)

すなわち，入力となる key と value は出力となる key と value とは異なるドメインであるが, 中間的な key と value は出力となる key と value と同じドメインである．

3 実装

MapReduce インターフェースは様々な異なる実装が可能である．その正しい選択は環境に依存している．たとえば，小規模な共有メモリマシンに適した実装, 大規模な NUMA 型マルチプロセッサに適したもの，さらに大規模なネットワークで接続されたマシンに適

した実装がある．我々の原論文が発表されて以来，幾つかのオープンソースによるMapReduceの実装が開発されてきた[1, 2]．また, MapReduceのさまざまな問題領域への適用について研究されてきた[7, 16].

この節では, Googleで広く利用されているコンピューティング環境，ギガビットイーサネットで接続されたコモディティ PC の大規模クラスタをターゲットとした MapReduce の実装について述べる[4]．我々の環境では，マシンは通常 x86 のデュアルプロセッサで Linux が動作している．メモリはマシン一台当たり 4 – 8GB．個々のマシンは通常 1 gigabit/second のネットワークバンド幅を持つ．ただし，マシン 1 台あたりが利用可能なバイセクションバンド幅は，1 gigabit/secondより大幅に小さい．コンピューティングクラスタは数千のマシンからなるため，マシン障害は常に起こる．ストレージは個々のマシンに直接接続された安価なIDEディスクによって提供されている．GFS という分散ファイルシステムを自社で開発し[10]，これらのディスクを管理している．ファイルシステムは，信頼性の低いハードウェアで可用性と信頼性を実現するため，複製を利用する．

ユーザはスケジューリングシステムにジョブを投入する．それぞれのジョブはタスクの集合からなり，スケジューラによりクラスタ内の利用可能なマシンの集合にマップされる．

3.1 実行の概要

Map 処理は，入力データを自動的に M セットに分割することによって，複数のマシンで分散して実行される．分割した入力は，異なるマシンで並列に処理することができる．Reduce 処理は，中間 key 空間を分割関数（例えば $hash(key) \bmod R$）によって R 個に分割することで分散して実行される. 分割数 R と，分割関数はユーザが指定する．

図 1 は我々の実装における MapReduce 操作の全体の流れを表している．ユーザプログラムが MapReduce 関数を呼ぶと，次のアクションが順に発生する．（リストの番号は，図 1 中の番号に対応する．）

1. ユーザプログラム内の MapReduce ライブラリは，はじめに入力ファイルを通常 16 -64MB の大きさの M ピースに分割する（オプションパラメータを通じてユーザによって制御可能）．その後，クラスタのマシンでプログラムのコピーを一斉に起動する．
2. プログラムのコピーのうちの一つはマスタとなり，特別な役割をもつ．残りはワーカとなり，マスタによって割り当てられた仕事を行う．割り当てられるべき M 個の map タスクと R 個の reduceタスクがあるとする. マスタは，タスク割り当てのないワーカを選び，それぞれに map タスクか reduce タスクを割り当てる．
3. map タスクを割り当てられたワーカは対応する分割された入力の内容を読み込む．入力データから key/value のペアを解析して，ユーザ定義の map 関数に渡す．中間 key/value ペアが map 関数で生成され，メモリ内にバッファリングされる．
4. バッファリングされた中間 key/value ペアは分割関数によって R 領域に分割され，定期的にローカルディスクに書き出される．ローカルディスク上のこれらのバッファリングされたペアの所在情報はマスタに返される. マスタは，その所在情報を reduce ワーカに伝える．
5. マスタが reduce ワーカにそれらの所在情報を伝えると，reduce ワーカは遠隔手続き呼出を使って map ワーカのローカルディスクにバッファリングされたデータを読み込む．reduce ワーカが分割内の全ての中間データを読み込んだら，中間 key によりソートし，同じキー同士をまとめる．通常たくさんの異なるkeyが同じreduceタスクに割り当てられるため，ソートは必要である．中間データ量が大きすぎてメモリに収まらない場合，外部ソートが使用される．
6. reduceワーカはソートされた中間データ全体について反復し，異なる中間 key と対応する中間 value のセットをユーザ定義の reduce 関数に渡す．reduce 関数の出力は，この reduce 分割のための最終出力ファイルに追記される．

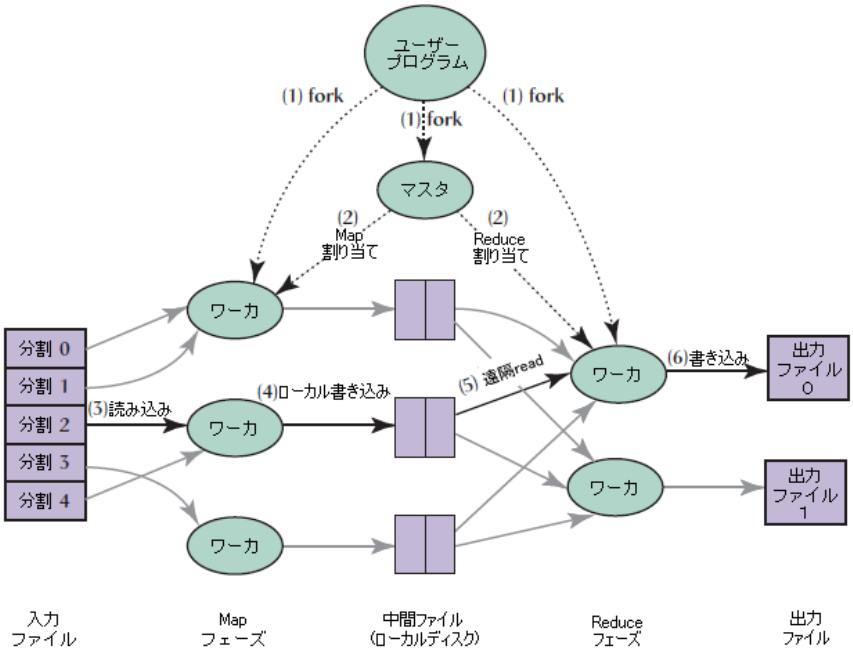

図 1 実行の概要

7. 全ての map タスクと reduce タスクが終了すると，マスタはユーザプログラムを起こす．この時点で，ユーザプログラム内の MapReduce 呼出はユーザコードに制御を戻す．

正常終了した後，MapReduce の実行結果の出力は R 個の出力ファイルとなる．（reduce タスクごとにできる．ファイルネームはユーザによって指定される．）ほとんどの場合，これらの R 個のファイルを 1 ファイルにする処理は必要とはならない．ユーザは，しばしばそれらの出力ファイルを別の MapReduce 呼出の入力として渡したり，複数ファイルに分割された入力を扱うことができる別の分散アプリケーションで利用したりするからである．

3.2 マスタのデータ構造

マスタは幾つかのデータ構造を保持する．それぞれの map タスクと reduce タスクについては，（*未実行*，*実行中*，*実行完了*）の実行状態，および（未実行状態でないタスクのために）ワーカマシンの識別情報を保持する．

マスタは，中間データファイルの所在情報を map タスクから reduce タスクに伝える役目を持っている．そのため，処理が終わった map タスクそれぞれにおいて，map タスクが生成した，R 分割された中間データファイルの所在情報とサイズを保持する．所在情報とサイズ情報は map タスクが完了すると更新される．この情報は reduce タスクを*実行中*のワーカにインクリメンタルに伝えられる．

3.3 耐故障性

MapReduce ライブラリは数百から数千のマシンを使って，大量のデータを処理するために設計されているので，うまくマシン障害に対応しなければならない．

ワーカ障害処理

マスタは全てのワーカに定期的に ping を送る．もし一定時間以内にワーカから何のレスポンスもなければ，マスタは，そのワーカを故障したとマークする．そのワーカで処理された map タスクは全て初期の*未実行*状態に戻され，ほかのワーカに再スケジューリングされる．同様に，障害が発生したワーカで実行中の map および reduce タスクも*未実行*状態に戻され，再スケジューリングの対象となる．

*実行完了*状態の map タスクがマシン障害時に再実行されるのは，map タスクの出力はローカルディスクに格納され，アクセスできなくなるためである．完了済み reduce タスク

は，グローバルファイルシステムに出力が格納されるため，再実行の必要はない．

あるmapタスクが，まずはワーカAで実行され，（Aに障害が発生し）後にワーカBで実行された場合，全てのreduceタスクを実行しているワーカは再実行の知らせを受ける．ワーカAからまだデータを読み込んでいないreduceタスクは，ワーカBからデータを読み込むことになる．

MapReduceは大規模なワーカ障害から回復することができる．例えば，あるMapReduce処理中に，実行中のクラスタのネットワークメンテナンスにより80台のマシンが一斉に数分間停止したとする．MapReduceのマスタは，停止したワーカで実行完了したタスクを，単純に再実行させることで処理を続け，最終的にはMapReduce処理は完了する．

障害発生時のセマンティクス

ユーザから与えられたmapとreduce処理が入力値に対して決定的な関数であるとき，我々の分散実装では，障害なしでプログラム全体を逐次実行した場合と同じ出力が得られる．

mapとreduceの出力はアトミックにコミットされるという性質を利用し，この特性を実現する．それぞれの実行中のタスクはプライベートな一時ファイルに出力を行う．reduceタスクはそのような1ファイルを生成する．mapタスクはそのような（各reduceタスクにつき1ファイルの）Rファイルを生成する．mapタスクが完了すると，ワーカはマスタにR個の一時ファイルの名前を伝える．もしマスタが既に完了済みのmapタスクの完了メッセージを重複して受け取った場合，そのメッセージを無視する．そうでない場合は，R個のファイル名をマスタのデータ構造に記録する．

reduceタスクが完了すると，reduceワーカは一時ファイルのファイル名を最終出力ファイル名にアトミックに変更する．もし，同一のreduceタスクが複数のマシンで実行されていた場合，同一の最終出力ファイル名への変更が複数回実行される．ファイルシステムによるアトミックな名前変更操作の性質により，最終的なファイルシステムの状態はどれか一つのreduceタスクの実行結果だけであることが保証される．

大半のmapとreduce処理は決定的であるため，逐次実行と同等というセマンティクスは，プログラマにとって非常にわかりやすいプログラムの挙動である．mapとreduce処理のどちらか，もしくは両方が非決定的である場合，弱いがまだ合理的なセマンティクスを提供する．非決定的な処理が存在する場合，ある特定のreduceタスクR_1の出力は，非決定的なプログラムのある逐次実行による出力となるかもしれないし，別のreduceタスクR_2は異なる逐次実行による出力となるかもしれない．

mapタスクMとreduceタスクR_1，R_2があるとする．$e(R_i)$を，R_iの（ある）コミットされた実行とする． $e(R_1)$はMのある実行の結果を入力とするかもしれないし，$e(R_2)$はMのまた異なる実行の結果を入力とするかもしれないため，このような弱いセマンティクスとなる．

3.4 局所性

ネットワークバンド幅は，我々のコンピューティング環境の中で比較的十分でないリソースである．我々はネットワークバンド幅を節約するために，（GFS [10]によって管理されている）入力データは我々のクラスタを構成するマシンのローカルディスクにあるという事実を利用する．GFSは各ファイルを64MBのブロックに分割し，数個（典型的には3つ）のコピーを異なるマシンに格納する．MapReduceのマスタは，入力ファイルの位置情報を考慮して，対応する入力ファイルのコピーを持つマシンにmapタスクをスケジュールするよう試みる．スケジューリングに失敗した場合，入力ファイルのコピーの近くに（例えば，データを格納するマシンと同じネットワークスイッチに接続されたワーカマシンに） mapタスクをスケジューリングするよう試みる．クラスタ内の大部分のワーカを使った大規模なMapReduce処理を実行すると，ほとんどの入力データはローカルディスクから読み込まれ，ネットワークバンド幅は消費されない．

3.5 タスクの粒度

既に述べたように，我々はタスクをマップフェイズで M 分割，reduce フェイズで R 分割している．理想的には，M と R はワーカ数よりもかなり多い方がよい．各ワーカに多くの異なるタスクを処理させることにより，動的な負荷分散が改善するだけでなく，(障害のあったワーカで）完了した多くのマップタスクを，それ以外の全てのワーカで処理することができるため，ワーカの障害からの迅速な回復処理が可能となる．

しかしながら，M と R の大きさには実装上の制限がある．マスタは $O(M+R)$ のスケジューリングを行うために，メモリ上に $O(M*R)$ の状態を記憶しておかなければならないためである．(しかしながら，メモリ利用における係数は小さい．$O(M*R)$ の状態は，map タスクと reduce タスクのペアごとに 1 バイト程度である)．

さらに，reduce タスクの結果は異なる出力ファイルにまとめられるため，R はしばしばユーザにより制限される．実際には，(既に述べた局所性の最適化が最もうまく働くように）それぞれのタスクの入力データが 16MB から 64MB になるように M を設定し，R は利用するワーカ数の数倍とすることが多い．我々は，ワーカ数 2,000 に対して，M = 200,000，R = 5,000 の MapReduce 計算をよく行う．

3.6 バックアップタスク

MapReduce の処理時間が増えるよくある原因は，落ちこぼれのためである．すなわち，最後のいくつかの map 処理や reduce 処理に非常に時間がかかってしまうノードのためである．この落ちこぼれは，様々な理由により起こりえる．例えば，障害の発生しているディスクを持つマシンでは頻繁にエラー修正が起こり，読み込み速度が 30MB/s から 1MB/s に低下してしまう．また，クラスタのスケジューリングシステムにより同一マシンに他のタスクが同時に割り当てられた場合，CPU，メモリ，ローカルディスクやネットワークバンド幅の競合が発生し，MapReduce コードの実行速度がより低下してしまう．我々が最近経験した問題としては，マシンの初期化処理のためのコードのバグが原因でプロセッサキャッシュが使用できなくなるというものがあった．このバグが影響を及ぼしたマシンでは，100 倍以上の速度低下がおきた．

この落ちこぼれ問題の一般的な対策として，MapReduce 処理が終わりに近づいた時，マスタは残っている*実行中*のタスクをバックアップタスクとしてスケジューリングする．そして，元のタスクとバックアップタスクのどちらかの実行が完了したらタスクは終了したと見なす．我々はこの対策による計算資源の増加を典型的には数パーセント以内にするよう最適化した．この対策により，重たい MapReduce 処理に要する時間を大幅に短縮できることが分かった．例えば，5.3 節で述べるソートプログラムは，バックアップタスクを用いない場合，実行時間が 44%伸びた．

4 拡張機能

単に map 関数と reduce 関数を記述するだけで，基本機能としては多くの場合に十分であるが，我々が発見した便利な機能拡張を以下に示す．

- 中間 key/value データを R 分割する方法を指定するためのユーザ定義分割関数
- 順序の保証．我々の実装では，分割されたそれぞれの reduce 処理の中では，中間 key/value ペアはキーの昇順で処理されることを保障している．
- (ネットワークを転送しなければならない中間データの量を減らすため）同一 map タスクの中で，同一 key で生成される中間 value を部分的に結合するためのユーザ定義結合関数．
- 新しい入力フォーマットで読み込み，新しい出力フォーマットで出力するための入力・出力のカスタムデータタイプ．
- デバッグや小規模テストのための単一マシン実行モード

これらの詳細については原論文[8]で議論されている．

5 性能評価

この章では，大規模クラスタシステム上で 2 種類の計算処理における MapReduce の性能を評価する．ひとつは，約 1 テラバイトのデータから特定のパターンを探索するものである．また，もう一つは，約 1 テラバイトのデータの整列である．

これら 2 つのプログラムは MapReduce のユーザが書く実際のプログラムの大部分を占める代表である．一つ目の問題クラスは，ある表現から別の表現へデータをシャッフルするもの，もう一つのクラスは，大規模データセットから興味のある小さなデータを取り出すものである．

5.1 クラスタ構成

全てのプログラムは，約 1,800 台のマシンからなるクラスタ上で実行された．各マシンは，ハイパースレッディングを有効にした 2GHz デュアル Intel Xeon プロセッサ，4GB メモリ，2 台の 160GB IDE ディスク，ギガビットイーサネットを持つ．各マシンは 2 階層木構造のスイッチで接続され，ルートで利用可能なバンド幅は約 100～200Gbps である．全てのマシンはすべて同じ施設内にあり，ネットワーク往復遅延時間はどのマシン間も 1 ミリ秒未満である．

4GB のメモリのうち，約 1～1.5GB が同時に実行されていた他のタスクによって利用されていた．評価プログラムは，週末の午後，CPU，ディスク，ネットワークがもっとも利用されていない時に実行された．

5.2 Grep

Grep プログラムは 1 レコード 100 バイトからなる合計 10^{10} のレコードから，比較的まれな 3 文字のパターンを探索する（このパターンは 92,337 レコードに現れる）．入力は約 64MB ずつ分割され（$M = 15{,}000$），出力全体は一ファイルである（$R = 1$）．

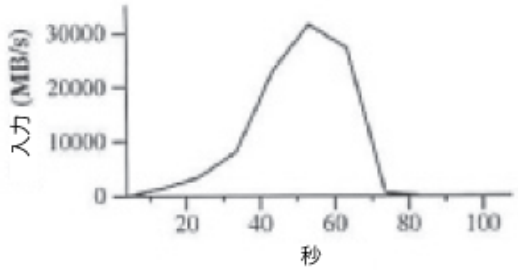

図 2 時間ごとのデータ転送レート (mr-grep)

図 2 は，時間に応じた計算状況の推移を表す．Y 軸は入力データの読み込み速度である．この MapReduce の処理で割り当てられるマシンが増加するにつれて，読み込み速度は徐々に増加している．最大では，1,764 のワーカが割り当てられ 30GB/s 以上の性能が出ている．map の処理が終わると，読み込み速度も下がり始め，約 80 秒で 0 となる．計算処理全体では開始から終了まで約 150 秒かかっている．これは，1 分ほどの立ち上げオーバヘッドを含んでいる．このオーバヘッドは，すべてのワーカにプログラムを行き渡らせるための時間，GFS による 1,000 入力ファイルのオープンにかかる遅延時間，局所性による最適化に必要な情報の取得時間などである．

5.3 整列

整列プログラムは，1 レコード 100 バイトからなる合計 10^{10} のレコード（約 1 テラバイトのデータ）の整列を行う．TeraSort ベンチマーク[12]を参考にしている．

整列プログラムはユーザコードで 50 行以下のプログラムである．最終的な整列結果は複製を二つもつ GFS ファイル（つまり，プログラムの出力としては 2 テラバイト）となる．はじめに，入力データを 64MB ずつに分割する（$M = 15{,}000$）．また，整列された出力データは 4,000 ファイル（$R = 4{,}000$）に分割される．分割のための関数としてはレコードの key の先頭 1 バイトを用いた．

今回のベンチマークプログラムで用いた分割関数は，key の分布の知識を使っている．一般の整列問題に対しては，前処理として MapReduce により key のサンプルを集め，その key のサンプルの分布を利用して，最終的な整列処理のための分割点を計算する．

図 3 に通常実行時の整列プログラムの経過を示す．最上段のグラフが入力の読み込み速度を表す．ピーク性能は 13GB/s ほどである．200 秒が経過するまでに，すべての map 処理が終了するため急速に 0 となる．Grep よりも読み込み速度が遅いのは，整列の map タスクは中間データのローカルディスクへの書き出しに約半分の時間と I/O バンド幅を利用しているためである．一方の Grep では，中間データの出力は無視できるほどのサイズである．

また，シャッフルと出力処理速度よりも入力処理速度のほうが速い．これは局所的な最適化がなされているためである．ほとんどのデータはローカルディスクから読み込み，ネットワークのバンド幅の制約を回避している．また，シャッフル処理の方が出力処理速

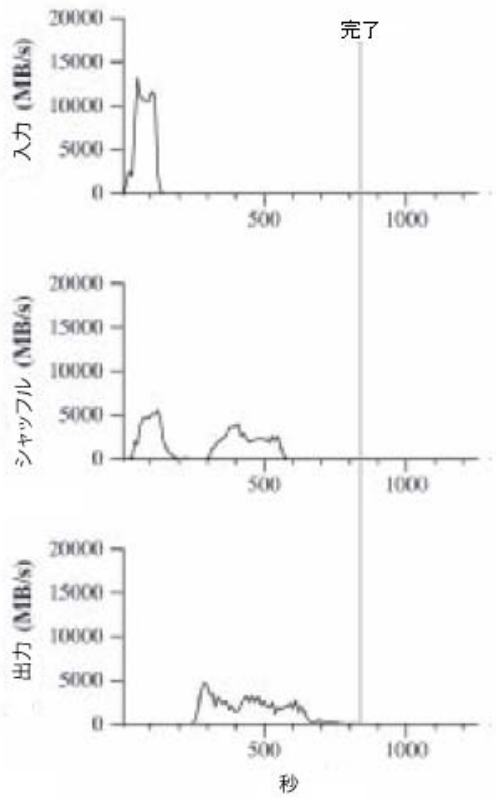

図 3 時間ごとのデータ転送レート
(mr-sort)

度よりも速いのは，出力フェイズでは整列結果の複製を二つ作成するためである（出力ファイルの複製を二つ作るのは信頼性と可用性の理由による）．複製を二つ作成するのは，それが我々の下位のファイルシステムが提供する信頼性と可用性を保証するメカニズムだからである．もしも，ファイルシステムが複製ではなく erasure coding [15]を用いていれば，書き込みに要求されるネットワークバンド幅を減らすことができるだろう．

原論文では，バックアップタスクによる効果や耐故障性についてさらなる実験を行っている[8].

6 経験

我々は，2003 年 2 月に最初のバージョンの MapReduce ライブラリを書き，2003 年の秋には，局所性を利用した最適化やワーカマシン間の動的なタスク配置の負荷分散などの大幅な機能強化を行った．それ以来，我々が取り組んできた様々な問題について MapReduce ライブラリがいかに幅広く適用可能であったか，ということに我々は喜ばしい意味で驚いている．Google の内部では以下のような多岐にわたる分野で用いられている．

- 大規模機械学習問題
- Google News や Froogle のためのクラスタリング問題
- データを抜き出して人気のあるクエリのレポートを作る処理（例えば，Google Zeitgeist や Google Trends など）
- 新しい実験的機能や製品のためのウェブページのプロパティの抽出（例えば，ローカル検索のためのウェブページの大規模コーパスからの地理的情報の抽出など）
- 衛星からの画像データの処理
- 統計的機械翻訳のための言語モデル処理
- 大規模グラフ計算

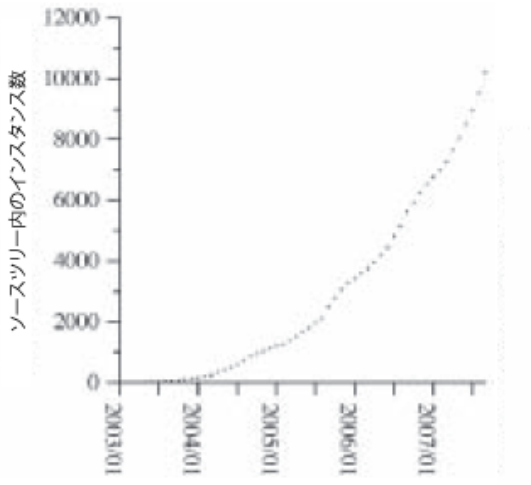

図 4 時間ごとの MapReduce インスタンス数

図 4 に，我々の主要なソースコード管理システムで管理している MapReduce プログラムの数が時間とともに著しく伸びている様子を示す．2003 年初頭の 0 からはじまり，2004 年の 9 月には 900，2006 年の 3 月には 4,000 にも達している．MapReduce がこれほどまで成功した理由は，プログラムがシンプルに書け，しかも 30 分で 1,000 台のマシン上で効率的に実行が可能であり，開発やプロトタイピングにかかる時間が劇的に短縮化されるためである．その上，MapReduce によって，分散システムや並列システムでのプログラミング

経験がなくても簡単に大量の資源を使用することが可能となる.

表 1 MapReduce の利用統計

	2004/4	2006/3	2007/9
ジョブ数 (1000s)	29	171	2,217
平均計算時間 (秒)	634	874	395
利用 CPU 年	217	2,002	11,081
map 入力データ (TB)	3,288	52,254	403,152
map 出力データ (TB)	758	6,743	34,774
reduce 出力データ (TB)	193	2,970	14,018
ジョブあたりの平均マシン台数	157	268	394
異なる実装数			
map	395	1958	4083
reduce	269	1208	2418

MapReduce ライブラリは各ジョブの終了時に，ジョブが使用した計算資源についての統計情報のログをとっている. 表 1 は, Google 内部で数ヶ月間の間に実行されていた MapReduce ジョブについての利用統計である．MapReduce の使用増加が顕著であり，Google において必要とされるほぼすべてのデータ処理に対して事実上標準的な選択肢となっている.

6.1 大規模インデックス

MapReduce の最も重要な利用例の一つは，Google のウェブ検索サービスに用いられるデータ構造を生成するプロダクションのインデクシングシステムを完全に MapReduce により書き直したことである．インデクシングシステムは入力としてクローリングシステムで収集した大規模な文書の集合をとる．文書の集合は GFS ファイルの集合として格納されている. 加工していない生の文書は 20 テラバイト以上にもなる．2003 年に MapReduce を用いてインデクシングシステムを書きなおしたとき，8 回の MapReduce 処理で実行された．それ以来，新しいフェイズを付け加えるのが簡単にできるために，インデクシングシステムには様々なフェイズが付け加えられてきた.（以前のアドホックな分散処理をしていたインデクシングシステムに比べ）MapReduce には様々な利点がある.

- インデクシングのためのコードがよりシンプルに，小さく，理解しやすくなった. これは, 耐故障性, データ分散, 並列化のためのコードが MapReduce ライブラリにより隠蔽されるからである．例えば，ある計算フェイズのコードサイズは，C++の約 3,800 行は MapReduce で表現されると 700 行まで減少した.
- MapReduce ライブラリは十分な性能があるために，余計なデータ処理を回避するために概念的には関係のない計算処理をひとまとまりにするようなことをしなくともすむ．これにより，インデクシングプロセスの変更が簡単になる．例えば，以前のインデクシングシステムでは数か月が必要となるような変更も，ものの数日で実装できるようになる.
- インデクシングプロセスの扱いがより簡単になった. これは, マシンの障害, 遅いマシン，ネットワークの一時的な障害により引き起こされる多くの問題が，オペレータの介入なしに MapReduce ライブラリにより動的に処理されるからである．さらには，クラスタにマシンを追加することにより簡単に性能を向上させることが可能である.

7 関連研究

多くのシステムが制限されたプログラミングモデルを提供し，その制限の下で自動並列化を行ってきた．例えば，結合則を満たす関数は，並列プレフィックス計算[6, 11, 14]により，N台のプロセッサで$\log N$時間でN要素の全配列のプレフィックス計算が可能である. MapReduceは，我々の大規模実世界計算における経験に基づいた，それらのモデルの単純化，洗練化と見なすことができる．より重要なこととして，我々は数千のプロセッサに対しスケールする耐故障性を備えた実装を提供した．一方で，ほとんどの並列処理システムはより小規模でしか実装されず，また，マシン故障の詳細な処理をプログラマに任せてしまう.

我々の局所性の最適化はactive disks [13,17]などの技術, I/Oシステムやネットワークに流すデータ量を減らすため計算処理をローカルディスクに近いPE（プロセッシングエレメント）に移動させる，にヒントを得たものである．

MapReduce ライブラリの一部になっている整列は，Now-Sort [3]の処理と同様である．起点マシン（map ワーカ） は整列すべきデータを分割し，*R* reduce ワーカの一つに送信する. それぞれの reduce ワーカはそのデータを（可能であればメモリ上で）整列させる．我々のライブラリはユーザ定義可能な map と reduce を持つため, 広範囲に適用可能であるが，もちろん NOW-Sort はそのようなユーザ定義可能な map と reduce を持たない．

BAD-FS [5]と TACC [9]は耐故障性の実装メカニズムとして再実行を用いているシステムである．

原論文[8]では関連研究についてより詳細に扱っている．

まとめ

MapReduce プログラミングモデルは，Google において，多くの様々な目的に首尾よく用いられてきた．我々はこの MapReduce の成功が以下に述べるような理由によっていると考えている．第一は，MapReduce のプログラミングモデルが非常に簡単に使えることである．分散システムや並列システムを扱ったことのないプログラマでも，並列化や耐障害性，局所性を意識した最適化，負荷分散などの詳細を意識する必要はない．第二は，多岐にわたる問題が MapReduce の計算として表現可能であることである．例えば，Google のプロダクションのウェブ検索サービスのためのデータ生成処理，整列，データマイニング，機械学習やその他の多くのシステムにおいて MapReduce は活用されている．第三は，数千台のマシンからなる大規模クラスタまでスケールするように実装されていることである．この実装によりこれらのマシン資源を効果的に利用することができ, それゆえ, Google で必要となるような大規模な計算問題にも適用可能である．

プログラミングモデルを制限することにより，並列・分散計算が簡単になり，耐故障性も簡単に得られるようなった．次に，ネットワークバンド幅は不十分である．そのため，数々の最適化は，ネットワークに流れるデータ量を抑えるためのものである．例えば，局所性を意識した最適化により，ローカルディスクから入力データを読み込むことができ，中間ファイルの複製の一つをローカルディスクへ書き込むことで使用ネットワークバンド幅を抑えることができる．最後に，冗長なプログラム実行により，低速なマシンの影響を少なくできるだけでなく，マシン障害やデータ消失への対処が可能となる．

謝辞

Josh Levenberg には MapReduce のユーザレベル API の改定時や機能拡張時に有益な助言を多数いただきました．また，特に共に MapReduce のために働いた方々と Google において MapReduce を利用したすべての方々からは，非常に有益なフィードバック，提案やバグレポートを頂きました．ここに感謝いたします．

文献

1. Hadoop: Open source implementation of MapReduce.http://lucene.apache.org/hadoop/.
2. The Phoenix system for MapReduce programming.http://csl.stanford.edu/~christos/sw/phoenix/.
3. Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Culler, D. E., Hellerstein, J. M., and Patterson, D. A. 1997. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data.* Tucson, AZ.
4. Barroso, L. A., Dean, J., and Urs Hölzle, U. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro 23*, 2, 22-28.
5. Bent, J., Thain, D., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Livny, M. 2004. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

6. Blelloch, G. E. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput. C-38*, 11.
7. Chu, C.-T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G., Ng, A., and Olukotun, K. 2006. Map-Reduce for machine learning on multicore. In *Proceedings of Neural Information Processing Systems Conference* (*NIPS*). Vancouver, Canada.
8. Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of Operating Systems Design and Implementation* (*OSDI*). San Francisco, CA. 137-150.
9. Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*. Saint-Malo, France. 78-91.
10. Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *19th Symposium on Operating Systems Principles*. Lake George, NY. 29-43.
11. Gorlatch, S. 1996. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds. *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science, vol. 1124. Springer-Verlag. 401-408
12. Gray, J. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.
13. Huston, L., Sukthankar, R., Wickremesinghe, R., Satyanarayanan, M., Ganger, G. R., Riedel, E., and Ailamaki, A. 2004. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*.
14. Ladner, R. E., and Fischer, M. J. 1980. Parallel prefix computation. *JACM 27*, 4. 831-838.
15. Rabin, M. O. 1989. Efficient dispersal of information for security, load balancing and fault tolerance. *JACM 36*, 2. 335-348.
16. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of 13th International Symposium on High-Performance Computer Architecture* (*HPCA*). Phoenix, AZ.
17. Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D. Active disks for large-scale data processing. *IEEE Computer*. 68-74.

Jeff Dean（jeff@google.com）は Google フェローである．現在カリフォルニア州マウンテンビューの Google 施設にて様々な大規模分散システムを研究開発している．

Sanjay Ghemawat（sanjay@google.com）は Google フェローである．カリフォルニア州マウンテンビューの Google 施設で，同社のほとんどのプロダクトで使用されている分散コンピューティング基盤を研究開発している．

訳：平賀弘平（筑波大学・第三学群情報学類）