MapReduce：大規模クラスタでのデータ処理の簡素化

ジェフリー・ディーン、サンジェイ・ゲマワット

jeff@google.com, sanjay@google.com

Google, Inc.

概要

MapReduceは、大規模データセットを処理・生成するためのプログラミングモデルと関連する実装である。ユーザは、キー/値のペアを処理して中間キー/値のペアのセットを生成するマップ関数と、同じ中間キーに関連するすべての中間値をマージするreduce関数を指定する。このモデルでは、論文にあるように、多くの実世界のタスクが表現可能である。

この機能スタイルで書かれたプログラムは自動的に並列化され、大規模なコモディティマシンのクラスタ上で実行される。ランタイムシステムは、入力データの分割、マシンセット間のプログラム実行のスケジューリング、マシン障害の処理、必要なマシン間通信の管理などの詳細を考慮する。これにより、並列分散システムの経験のないプログラマが、大規模な分散システムのリソースを容易に利用できるようになる。

我々のMapReduceの実装は、大規模なコモディティマシンのクラスタ上で実行され、非常にスケーラブルである:典型的なMapReduceの計算は、数千のマシン上で何テラバイトものデータ処理を行う。何百ものMapReduceプログラムが実装され、毎日1000以上のMapReduceジョブがGoogleのクラスタ上で実行されている。

1 はじめに

過去5年間、Googleの著者や他の多くの研究者は、クロールされた文書、ウェブリクエストログなど、大量の生データを処理する数百の特別な目的の計算を実装し、反転インデックス、ウェブ文書のグラフ構造の様々な表現、ホストごとにクロールされたページ数の要約、ある日に最も頻繁に行われたクエリのセットなど、様々な種類の派生データを計算してきた。

このような計算のほとんどは、概念的に簡単である。しかし、入力データは通常大きく、妥当な時間で終了するためには、計算を数百から数千のマシンに分散させる必要がある。計算を並列化し、データを分散させ、失敗を処理する方法の問題は、これらの問題に対処するために、大量の複雑なコードで元の単純な計算を不明瞭にするように共謀する。

この複雑さへの反応として、我々は、我々が実行しようとしている単純な計算を表現することを可能にする新しい抽象化を設計したが、並列化、フォールトトレランス、データ分布、および負荷分散の厄介な詳細をライブラリに隠す。我々の抽象化は、Lispや他の多くの機能言語に存在するマップとリデュースプリミティブに触発されたものである。我々は、ほとんどの計算が、中間キー/値のペアのセットを計算するために、入力の各論理「レコード」にマップ演算を適用し、次に、導出されたデータを適切に結合するために、同じキーを共有するすべての値にreduce演算を適用することを含むことに気づいた。ユーザが指定したマップとリデュース操作を持つ関数モデルを使用することで、大規模な計算を容易に並列化することができ、フォールトトレランスの主要なメカニズムとして再実行を使用することができる。

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にするシンプルで強力なインターフェースと、商品PCの大規模クラスタ上で高い性能を達成するこのインターフェースの実装を組み合わせたことである。

第2節では、基本的なプログラミングモデルを説明し、いくつかの例を挙げる。セクション3では、我々のクラスタベースのコンピューティング環境に合わせたMapReduceインターフェースの実装について説明する。セクション4では、我々が有用と判断したプログラミングモデルのいくつかの改良について述べる。セクション5では、様々なタスクに対する我々の実装の性能測定を行う。セクション6では、GoogleにおけるMapReduceの利用について、

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

本番インデックスシステムの書き換えの基礎として使用した経験を含めて検討する。セクション7では、関連する仕事と今後の仕事について述べる。

2 プログラミングモデル

計算は、入力キー/値のペアのセットを受け取り、出力キー/値のペアのセットを生成する。MapReduceライブラリのユーザーは、計算を2つの関数として表現する: MapとReduceである。

ユーザが書いたMapは、入力ペアを受け取り、中間キー/値のペアのセットを生成する。MapReduceライブラリは、同じ中間キーIに関連するすべての中間値をグループ化し、Reduce関数に渡す。

Reduce関数は、同じくユーザーが書いたもので、中間キーIとそのキーの値のセットを受け入れる。これらの値を統合して、より小さな値の集合を形成する。通常、Reduceの呼び出しごとに出力値が0か1つだけ生成される。中間値は、イテレータを介してユーザのreduce関数に供給される。これにより、大きすぎてメモリに収まらない値のリストを扱うことができる。

2.1 例題

大規模な文書コレクションにおける各単語の出現回数を数える問題を考えてみよう。ユーザーは以下の擬似コードに似たコードを書く:

```
map(String key, String value):
  // key: document name
  // 値:ドキュメントの内容
  各単語wの値に対して
    中間(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  エミット(AsString(結果));
```

map関数は各単語と関連する出現回数を放出する(この単純な例では'1'だけ)。reduce関数は、特定の単語に対して放出されたすべてのカウントを合計する。

さらに、ユーザーは、入力ファイルと出力ファイルの名前、およびオプションのチューニングパラメータで、mapreduce仕様オブジェクトを埋めるコードを書き込む。次に、ユーザーはMapReduce関数を起動し、仕様オブジェクトを渡す。ユーザーのコードはMapReduceライブラリ(C++で実装)と一緒にリンクされている。付録Aには、この例のプログラム全文が含まれている。

2.2 タイプ

前の擬似コードが文字列の入力と出力で書かれているにもかかわらず、概念的には、ユーザーから提供されるマップ関数とリデュース関数は関連するタイプを持っている:

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

すなわち、入力キーと値は、出力キーと値とは異なるドメインから引き出される。さらに、中間キーと中間値は出力キーと中間値と同じドメインのものである。

我々のC++実装は、ユーザー定義の関数に文字列を渡す。また、文字列と適切な型との間の変換はユーザーコードに委ねる。

2.3 その他の例

以下は、MapReduceの計算として簡単に表現できる興味深いプログラムの簡単な例である。

分散Grep: マップ関数が供給されたパターンに一致する場合、線を発する。reduce関数は、供給された中間データを出力にコピーするだけの恒等関数である。

URLアクセス頻度のカウント: map関数はウェブページリクエストのログを処理し、hURL, liを出力する。reduce関数は、同じURLのすべての値を加算し、hURL、total countiのペアを発行する。

リバースWebリンクグラフ: マップ関数は、sourceという名前のページにあるターゲットURLへのリンクごとに、htarget, sourceiのペアを出力する。reduce関数は、与えられたターゲットURLに関連するすべてのソースURLのリストを連結し、以下のペアを発行する: htarget, list(source)i

ホストごとの用語ベクトル: 用語ベクトルは、文書または文書の集合に出現する最も重要な単語を、hword, f requencyi pairsのリストとして要約する。map関数は、各入力文書に対してhhostname, term vectori pairを発行する(ここで、ハイパーリンクは文書のURLから抽出される)。reduce関数は、与えられたホストの文書ごとの用語ベクトルをすべて渡す。これらの項ベクトルを足し合わせ、頻度の低い項を捨て、最終的なhhostname、項ベクトルiのペアを発する。

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL`, `1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL`, `total count`$\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle$`target`, `source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target`, $list$(`source`)$\rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname`, `term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname`, `term vector`$\rangle$ pair.

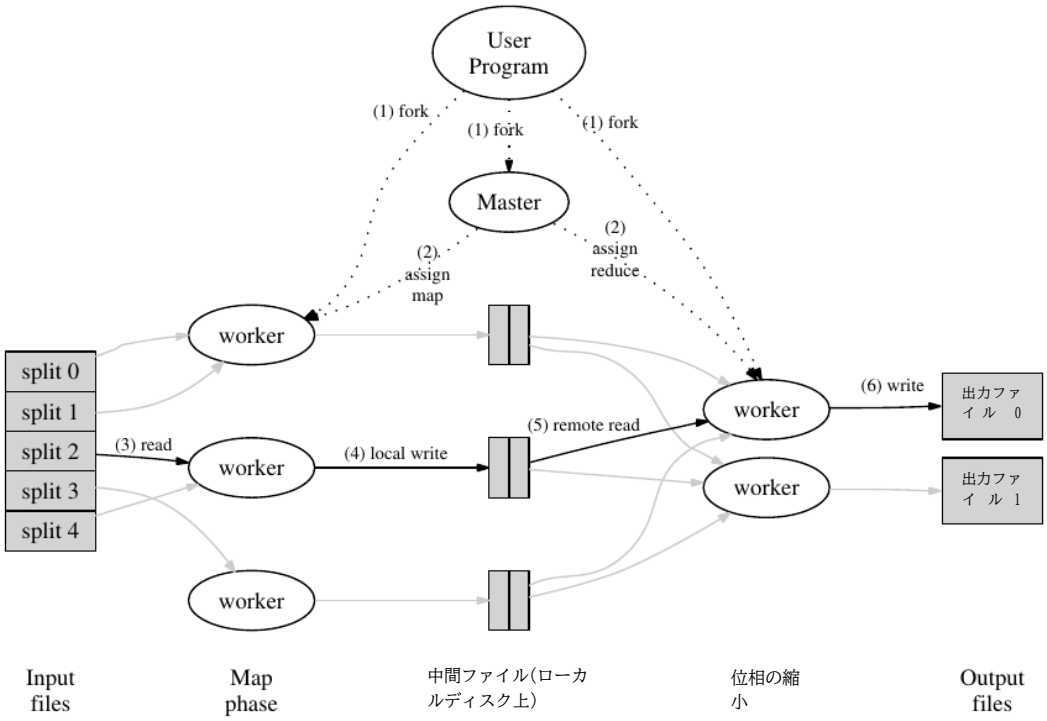


図1:実行の概要

逆インデックス: マップ関数は各文書を解析し、hword, 文書IDiのペアのシーケンスを生成する。reduce関数は与えられた単語のすべてのペアを受け入れ、対応する文書IDをソートし、hword, list(document ID)iのペアを発行する。すべての出力ペアの集合は単純な逆インデックスを形成する。単語の位置を追跡するために、この計算を拡張するのは簡単である。

分散ソート: マップ関数は各レコードからキーを抽出し、hkey, recorbi ペアを発行する。reduce関数はすべてのペアを変更せずに出力する。この計算は、4.1節で説明したパーティショニング機能と、4.2節で説明した順序付け特性に依存する。

3 実装方法

MapReduceインターフェースの多くの異なる実装が可能である。正しい選択は環境に依存する。例えば、ある実装は小さな共有メモリマシンに適し、別の実装は大きなNUMAマルチプロセッサに適し、さらに別の実装はさらに大きなネットワークマシンのコレクションに適するかもしれない。

このセクションでは、Googleで広く使われているコンピューティング環境をターゲットにした実装について説明します: スイッチドイーサネット[4]と一緒に接続されたコモディティPCの大規模なクラスタ。我々の環境では

(1) マシンは通常、Linux を実行するデュアルプロセッサ x86 プロセッサで、1 台あたり 2〜4GB のメモリを持つ。

(2) コモディティ・ネットワーキング・ハードウェアが使用される。通常、マシン・レベルでは100メガビット/秒または1ギガビット/秒のいずれかであるが、全体的な二等分帯域幅の平均はかなり小さい。

(3) クラスタは数百台から数千台のマシンで構成されており、したがってマシンの故障は一般的である。

(4) ストレージは、個々のマシンに直接取り付けられた安価なIDEディスクによって提供される。これらのディスクに保存されているデータを管理するために、社内で開発された分散ファイルシステム[8]が使用される。ファイルシステムは、信頼性の低いハードウェアの上で、可用性と信頼性を提供するために複製を使用する。

(5) ユーザがジョブをスケジューリングシステムに提出する。各ジョブはタスクの集合からなり、スケジューラによってクラスタ内の利用可能なマシンの集合にマッピングされる。

3.1 実行の概要

Mapの呼び出しは、入力データを自動的にM個の分割セットに分割することで、複数のマシンに分散される。

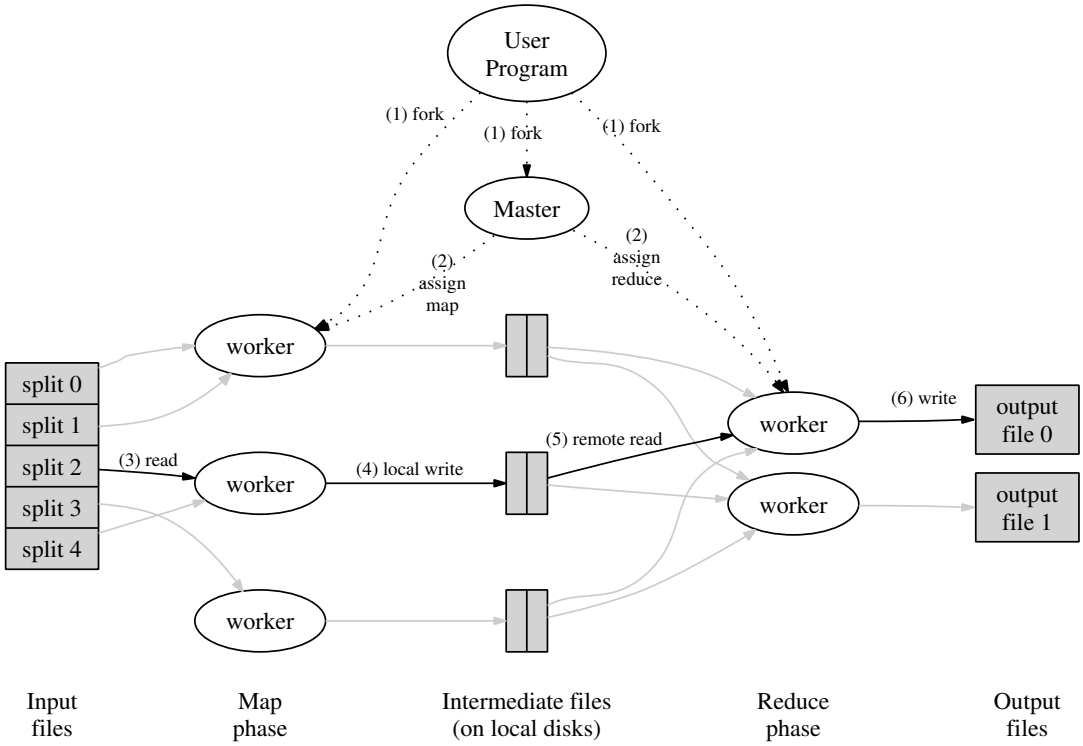


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle$`word`, `document ID`$\rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle$`word`, $list$(`document ID`)$\rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle$`key`, `record`$\rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

入力分割は異なるマシンで並列に処理できる。削減呼び出しは、分割関数(例えば、hash(key) mod R)を使用して、中間キー空間をR個の断片に分割することによって分配される。パーティション数(R)とパーティション関数はユーザが指定する。

図1は、本実装におけるMapReduce操作の全体フローを示している。ユーザープログラムがMapReduce関数を呼び出すと、以下の一連の動作が発生します(図1の番号付きラベルは、以下のリストの番号に対応します)。

1. ユーザープログラムのMapReduceライブラリは、まず入力ファイルを1個あたり通常16メガバイトから64メガバイト(MB)のM個に分割します(オプションのパラメータでユーザーが制御できます)。その後、マシンのクラスタ上でプログラムの多くのコピーを開始する。

2. プログラムのコピーの1つは特別なものです - マスター。残りは、修士によって仕事を割り当てられた労働者である。M個のマップタスクとR個のリデュースタスクがある。マスターはアイドル状態のワーカーを選び、それぞれにマップタスクとリデュースタスクを割り当てる。

3. 地図タスクが割り当てられたワーカーは、対応する入力分割の内容を読み取る。入力データからキー/値のペアを解析し、各ペアをユーザー定義のMap関数に渡す。Map関数によって生成される中間キー/値ペアはメモリ上でバッファリングされる。

4. 周期的に、バッファリングされたペアはローカルディスクに書き込まれ、分割関数によってR個の領域に分割される。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位置はマスターに渡され、マスターはこれらの位置をリデュースワーカーに転送する役割を担う。

5. リデュースワーカーがマスターからこれらの位置について通知されると、リモートプロシージャコールを使用して、マップワーカーのローカルディスクからバッファリングされたデータを読み込む。リデュースワーカーがすべての中間データを読み取ったとき、同じキーのすべての出現がグループ化されるように、中間キーでソートする。ソートが必要なのは、通常、多くの異なるキーが同じリデュースタスクに対応しているからである。中間データの量が多すぎてメモリに収まらない場合は、外部ソートを使用する。

6. reduceワーカーはソートされた中間データを反復し、遭遇するユニークな中間キーごとに、キーと対応する中間値のセットをユーザーのReduce関数に渡す。Reduce関数の出力は、このreduceパーティションの最終出力ファイルに付加される。

7. すべてのマップタスクとリデュースタスクが完了すると、マスターはユーザープログラムを起動する。この時点で、ユーザープログラムのMapReduce呼び出しはユーザーコードに戻る。

完了に成功した後、mapreduce実行の出力はR個の出力ファイル(reduceタスクごとに1つ、ユーザーによって指定されたファイル名)で利用可能である。通常、ユーザーはこれらのR出力ファイルを1つのファイルに結合する必要はない。これらのファイルを別のMapReduce呼び出しの入力として渡すか、複数のファイルに分割された入力に対応できる別の分散アプリケーションから使用することが多い。

3.2 マスターデータ構造

マスターはいくつかのデータ構造を保持する。各マップタスクとリデュースタスクについて、状態(アイドル、進行中、完了)とワーカーマシンのID(アイドルでないタスクの場合)を格納する。

マスターは、中間ファイル領域の位置がマップタスクから伝搬され、タスクを減らすための導管である。したがって、完了したマップタスクごとに、マスターはマップタスクによって生成されたR個の中間ファイル領域の位置とサイズを保存する。この位置情報とサイズ情報の更新は、地図タスクが完了すると受信される。情報は、進行中の削減タスクを持つワーカーに漸進的にプッシュされる。

3.3 フォールトトレランス

MapReduceライブラリは、数百台または数千台のマシンを使用して非常に大量のデータを処理するのを助けるように設計されているため、ライブラリはマシンの故障を優雅に許容しなければならない。

ワーカーの失敗

マスターはすべてのワーカーを定期的にピングする。ある時間内に作業員から応答が得られなかった場合、マスターは作業員を失敗とマークする。ワーカーが完了したマップタスクは、最初のアイドル状態にリセットされるため、他のワーカーでのスケジューリングに適格となる。同様に、失敗したワーカーで進行中の地図タスクや縮小タスクもアイドルにリセットされ、再スケジューリングの対象となる。

完成したマップタスクは、その出力が故障したマシンのローカルディスクに保存されるため、アクセスできないため、故障時に再実行される。縮小タスクの出力はグローバルファイルシステムに保存されているため、再実行する必要はない。

マップタスクがワーカーAによって最初に実行され、その後ワーカーBによって実行された場合(Aが失敗したため)、

into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \bmod R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

リデュースタスクを実行するすべてのワーカーに再実行が通知される。作業者Aのデータがまだ読んでいないリデュースタスクは、作業者Bのデータを読む。

MapReduceは大規模なワーカーの失敗に強い。例えば、あるMapReduceの運用中、実行中のクラスタ上でのネットワーク保守により、一度に80台のマシンのグループが数分間到達できなくなった。MapReduceマスターは、到達不可能なワーカーマシンが行った作業を再実行し、前進を続け、最終的にMapReduceの操作を完了した。

マスターの失敗

マスターデータ構造体の周期チェックポイントをマスターに書くのは簡単である。マスタータスクが死亡した場合、最後のチェックポイント状態から新しいコピーを開始することができる。しかし、マスターが1つしかないことを考えると、その失敗は起こりにくい。したがって、現在の実装では、マスターが失敗した場合、MapReduceの計算を中止する。クライアントはこの条件をチェックし、希望すればMapReduce操作を再試行することができる。

失敗が存在する場合の意味論

ユーザが提供するmap演算子とreduce演算子が入力値の決定論的関数である場合、我々の分散実装は、プログラム全体の非障害順次実行によって生成されたのと同じ出力を生成する。

この特性を達成するために、マップのアトミックコミットとタスク出力の削減に依存する。各進行中のタスクは、その出力をプライベートな一時ファイルに書き込む。reduceタスクはそのようなファイルを1つ生成し、mapタスクはそのようなファイルをR個生成する(reduceタスクごとに1つ)。マップタスクが完了すると、ワーカーはマスターにメッセージを送信し、R個の一時ファイルの名前をメッセージに含める。マスタがすでに完了したマップタスクの完了メッセージを受信した場合、そのメッセージを無視する。そうでなければ、マスターデータ構造でRファイルの名前を記録する。

reduceタスクが完了すると、reduceワーカーは一時的な出力ファイルを最終的な出力ファイルにアトミックにリネームする。同じリデュースタスクが複数のマシンで実行される場合、同じ最終出力ファイルに対して複数のリネームコールが実行される。最終的なファイルシステムの状態が、reduceタスクの1回の実行によって生成されたデータだけを含むことを保証するために、基礎となるファイルシステムによって提供されるアトミックリネーム操作に依存する。

我々のマップとリデュース演算子の大部分は決定論的であり、この場合、我々のセマンティクスは逐次実行と等価であるという事実は、プログラマがプログラムの動作を推論することを非常に容易にする。

map演算子および/またはreduce演算子が非決定論的である場合、より弱いが妥当なセマンティクスを提供する。非決定性演算子が存在する場合、特定の削減タスクR_1の出力は、非決定性プログラムの逐次実行によって生成されるR_1の出力と等価である。しかし、異なる削減タスクR_2の出力は、非決定性プログラムの異なる逐次実行によって生成されたR_2の出力に対応する可能性がある。

マップタスクMを考え、タスクR_1とR_2を減らす.e(R_i)をコミットしたR_i の実行とする(そのような実行はちょうど1つである)。e(R_1) はMの1回の実行で生成された出力を読み、e(R_2) はMの異なる実行で生成された出力を読み取った可能性があるため、弱い意味論が生じる。

3.4 局 所 性

ネットワーク帯域幅は、我々のコンピューティング環境では比較的少ないリソースである。入力データ(GFS [8]で管理)がクラスタを構成するマシンのローカルディスクに保存されていることを利用して、ネットワーク帯域幅を節約する。GFSは各ファイルを64MBのブロックに分割し、各ブロックのコピーを複数個(通常3個)異なるマシンに格納する。MapReduceマスターは、入力ファイルの位置情報を考慮し、対応する入力データのレプリカを含むマシン上でマップタスクのスケジューリングを試みる。その失敗は、マップタスクをそのタスクの入力データのレプリカの近くにスケジュールしようとする(例えば、データを含むマシンと同じネットワークスイッチ上にあるワーカーマシン上)。クラスタ内のワーカーのかなりの割合で大規模なMapReduce操作を実行する場合、ほとんどの入力データはローカルに読み込まれ、ネットワーク帯域幅を消費しない。

3.5 タスクの粒度

マップ位相をM個に細分化し、リデュース位相をR個に細分化する。理想的には、MとRはワーカーマシンの数よりはるかに大きいべきである。各ワーカーに多くの異なるタスクを実行させることで、動的負荷分散が改善され、ワーカーが失敗したときに回復が早まる。ワーカーが完了した多くのマップタスクは、他のすべてのワーカーマシンに分散させることができる。

マスターはO(M + R)のスケジューリング決定を行い、O(M * R)の状態をメモリ上に保持しなければならないので、我々の実装ではMとRがどの程度大きくなり得るかについて実用的な境界がある。(ただし、メモリ使用量の定数係数は小さい:O(M * R)個の状態は、マップタスク/リデュースタスクのペアごとに約1バイトのデータで構成される)。

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

さらに、Rは、各リデュースタスクの出力が別々の出力ファイルに行き着くため、ユーザーによって制約を受けることが多い。実際には、個々のタスクがおよそ16MBから64MBの入力データになるようにMを選択する傾向があり(上述の局所性最適化が最も効果的になるように)、Rを我々が使用すると予想されるワーカーマシンの数の小さな倍数にする。M=200,000、R=5,000で、2,000台のワーカーマシンを使ってMapReduce計算を行うことが多い。

3.6 バックバックタスク

MapReduce操作にかかる総時間を長くする一般的な原因の1つは「ストラグラー」である。つまり、最後の数マップのいずれかを完了したり、計算のタスクを減らしたりするのに、異常に長い時間を要するマシンである。ストラグラーは、多くの理由で発生する可能性がある。例えば、ディスクが悪いマシンでは、修正可能なエラーが頻繁に発生し、読み取り性能が30MB/sから1MB/sに低下することがある。クラスタスケジューリングシステムは、CPU、メモリ、ローカルディスク、ネットワーク帯域幅の競合により、マシン上で他のタスクをスケジューリングし、MapReduceコードの実行速度を遅くしている可能性がある。私たちが経験した最近の問題は、プロセッサのキャッシュを無効にしたマシン初期化コードのバグでした。影響を受けたマシンでの計算が100倍以上遅くなった。

我々は、ストラグラーの問題を軽減するための一般的なメカニズムを持っている。MapReduce操作が完了に近い場合、マスターは残りの進行中のタスクのバックアップ実行をスケジュールする。タスクは、プライマリまたはバックアップの実行が完了するたびに完了するとマークされる。このメカニズムは、通常、演算に使用される計算資源を数パーセント以下にするように調整した。これにより、大規模なMapReduce操作を完了する時間が大幅に短縮されることがわかった。例として、5.3節で説明したソートプログラムは、バックアップタスク機構を無効にした場合、44%長く完了する。

4 リファイン

Map関数とReduce関数を単純に書くだけで、基本的な機能はほとんどのニーズに十分であるが、いくつかの拡張機能が有用であることがわかった。これらについては、このセクションで説明する。

4.1 パーティショニング関数

MapReduceのユーザーは、希望するリデュースタスク/出力ファイルの数を指定する(R)。データは中間キーのパーティショニング関数を使用して、これらのタスクに分割される。

ハッシュを使用するデフォルトのパーティショニング関数が提供されています(例:"hash(key) mod R")。これはかなりバランスの取れたパーティションになる傾向がある。しかし、場合によっては、鍵の他の機能でデータを分割することが有用である。例えば、出力キーがURLである場合があり、一つのホストのエントリがすべて同じ出力ファイルに収まるようにしたい。このような状況をサポートするために、MapReduceライブラリのユーザーは特別なパーティショニング機能を提供することができる。例えば、"hash(Hostname(urlkey)) mod R "をパーティショニング関数として使用すると、同じホストからのURLがすべて同じ出力ファイルに行き着く。

4.2 オーダー保証

与えられたパーティション内で、中間キー/値のペアがキーの昇順で処理されることを保証する。この順序保証により、パーティションごとにソートされた出力ファイルを生成することが容易になる。これは、出力ファイル形式がキーによる効率的なランダムアクセス検索をサポートする必要がある場合や、出力のユーザーがデータをソートすることが便利である場合に有効である。

4.3 コンバイナー機能

場合によっては、各マップタスクによって生成される中間キーに大きな繰り返しがあり、ユーザが指定したReduce関数は可換かつ連想的である。この良い例は、セクション2.1の単語カウントの例である。単語頻度はZipf分布に従う傾向があるので、各マップタスクは<the, 1>の形の数百から数千のレコードを生成する。これらのカウントはすべてネットワーク上に送信され、1つのReduceタスクに続き、Reduce関数によって1つの数値が追加される。このデータをネットワーク上で送信する前に、ユーザーがオプションでCombiner関数を指定できるようにする。

Combiner関数は、マップタスクを実行する各マシンで実行される。通常、コンバイナーとリデュースの両方の実装に同じコードが使用される。reduce関数とコンバイナー関数の唯一の違いは、MapReduceライブラリが関数の出力をどのように処理するかである。reduce関数の出力は、最終的な出力ファイルに書き込まれる。コンバイナー関数の出力は、reduceタスクに送られる中間ファイルに書き込まれる。

部分結合は、MapReduceの特定のクラスの操作を大幅に高速化する。付録Aはコンバイナーを使った例である。

4.4 入出力タイプ

MapReduceライブラリは、入力データをいくつかの異なるフォーマットで読み込むためのサポートを提供する。

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

例えば、”text ”モードの入力は、各行をキー/値のペアとして扱います:キーはファイル内のオフセット、値は行の内容です別の一般的なサポート形式は、キー/値のペアのシーケンスをキーごとにソートして保存する。各入力タイプの実装は、別々のマップタスクとして処理するために、自分自身を意味のある範囲に分割する方法を知っている(例えば、テキストモードの範囲分割は、範囲分割が行境界でのみ発生することを保証する)。ユーザーは、シンプルなリーダーインターフェースの実装を提供することで、新しい入力タイプのサポートを追加することができますが、ほとんどのユーザーは、あらかじめ定義された少数の入力タイプのいずれかを使用するだけですでした。

読者は必ずしもファイルから読み取ったデータを提供する必要はない。例えば、データベースやメモリにマッピングされたデータ構造からレコードを読み出すリーダーを定義するのは簡単である。
同様の方法で、異なるフォーマットのデータを生成するための出力タイプのセットをサポートし、ユーザーコードが新しい出力タイプのサポートを追加するのは簡単である。

4.5 副作用

MapReduceのユーザーは、マップや/演算子を削減するための追加出力として、補助ファイルを作成するのに便利である場合がある。このような副作用を原子的かつ恒常的にするために、アプリケーションライターに頼っている。通常、アプリケーションは一時的なファイルに書き込み、完全に生成されるとこのファイルをアトミックにリネームする。

1つのタスクで生成される複数の出力ファイルに対する原子的な2段階コミットのサポートは提供しない。したがって、クロスファイル整合性要件を持つ複数の出力ファイルを生成するタスクは、決定論的であるべきである。この制限は、実際には決して問題になっていない。

4.6 悪い記録のスキップ

MapやReduce関数が特定のレコードで決定論的にクラッシュする原因となるバグがユーザーコードに存在することがある。このようなバグは、MapReduceの操作が完了するのを妨げる。通常の動作はバグを修正することであるが、実行不可能な場合もある。バグはサードパーティライブラリにあり、ソースコードが利用できないのかもしれない。また、大規模なデータセットで統計解析を行う場合など、少数の記録を無視することが許容される場合もある。MapReduceライブラリが決定論的なクラッシュの原因となるレコードを検出し、これらのレコードをスキップして前進させるオプションの実行モードを提供する。

各ワーカープロセスは、セグメンテーション違反とバスエラーを捕捉するシグナルハンドラをインストールする。ユーザーMapまたはReduce操作を呼び出す前に、MapReduceライブラリは引数のシーケンス番号をグローバル変数に格納する。

ユーザコードが信号を生成した場合、信号ハンドラはシーケンス番号を含む「最後の呼び出し」UDPパケットをMapReduceマスタに送信する。マスターが特定のレコードで複数の失敗を見た場合、対応するMapまたはReduceタスクの次の再実行を発行したときに、そのレコードをスキップする必要があることを示す。

4.7 ローカル実行

実際の計算は分散システムで行われ、多くの場合数千台のマシンで行われ、作業割り当ての決定はマスターによって動的に行われるため、MapやReduce関数の問題をデバッグするのは厄介である。デバッグ、プロファイリング、小規模なテストを容易にするために、ローカルマシン上でのMapReduce操作のためのすべての作業を順次実行するMapReduceライブラリの代替実装を開発した。制御は、計算を特定のマップタスクに限定できるように、ユーザーに提供される。ユーザは自分のプログラムを特別なフラグで呼び出し、その後、有用と思われるデバッグツールやテストツール(例:gdb)を簡単に使うことができる。

4.8 ステータス情報

マスターは内部のHTTPサーバーを実行し、人間が消費するためにステータスページのセットをエクスポートする。ステータス・ページは、完了したタスクの数、進行中のタスクの数、入力のバイト数、中間データのバイト数、出力のバイト数、処理速度など、計算の進捗を示す。ページには、各タスクによって生成された標準エラーファイルと標準出力ファイルへのリンクも含まれている。ユーザはこのデータを使って、計算にかかる時間や、計算にリソースを追加すべきかどうかを予測することができる。これらのページは、計算が予想よりはるかに遅い場合を把握するためにも使用できる。

さらに、トップレベルのステータス・ページは、どのワーカーが失敗したのか、失敗したときにどの作業をマッピングし、削減しているのかを示している。この情報は、ユーザーコードのバグを診断する際に有用である。

4.9 カウンター

MapReduceライブラリは、様々なイベントの発生をカウントするためのカウンター機能を提供する。例えば、ユーザーコードは、処理された単語の総数やインデックスされたドイツ語文書の数などを数えたいと思うかもしれない。

この機能を使用するために、ユーザーコードは名前付きカウンターオブジェクトを作成し、Map and/or Reduce関数でカウンターを適切にインクリメントする。例えば

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(文字列名、文字列内容):
  for 内容中の各単語w
    if (IsCapitalized(w)):
      uppercase->Increment();
    中間(w, "1");
```

個々のワーカーマシンからのカウンタ値は、定期的にマスターに伝搬されます(pingレスポンスにピギーバックされます)。マスターは、成功したマップとリデュースタスクのカウンタ値を集約し、MapReduce操作が完了したらユーザーコードに返す。現在のカウンタ値もマスターステータスページに表示され、人間がライブ計算の進行状況を確認できるようになっている。カウンタ値を集約する場合、マスターは同じマップの重複実行の影響を排除するか、二重カウントを避けるためにタスクを減らす。(重複実行は、バックアップタスクの使用や、障害によるタスクの再実行から発生する可能性があります)。

MapReduceライブラリによって、入力キー/値のペアの処理数や出力キー/値のペアの生成数など、いくつかのカウンタ値が自動的に保持される。

ユーザは、MapReduce操作の動作をサニティチェックするために、カウンター機能が有用であると判断した。例えば、MapReduceの操作では、ユーザーコードは、生成される出力ペアの数が処理される入力ペアの数と正確に等しいこと、または、処理されるドイツ語文書の割合が、処理される文書の総数のある許容可能な割合の範囲内であることを保証したい場合がある。

5 パフォーマンス

このセクションでは、大規模なマシンクラスタ上で実行される2つの計算に対するMapReduceの性能を測定する。1つの計算で、特定のパターンを探す約1テラバイトのデータを検索することができる。もう一つの計算は、およそ1テラバイトのデータをソートする。

これらの2つのプログラムは、MapReduceのユーザーが書いた実際のプログラムの大きなサブセットを代表するものである。1つのクラスのプログラムは、ある表現から別の表現へデータをシャッフルし、別のクラスは、大きなデータセットから少量の興味深いデータを抽出する。

5.1 クラスター構成

すべてのプログラムは、約1800台のマシンで構成されるクラスタ上で実行された。各マシンには、Hyper Threadingが有効な2GHz Intel Xeonプロセッサ2台と、

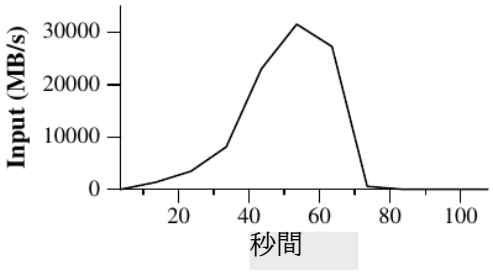


図2:経時的なデータ転送速度

4GBのメモリ、160GBのIDEディスク2台、ギガビットのイーサネットリンクがあった。マシンは、ルートで利用可能な総帯域幅が約100～200Gbpsの2レベルツリー型スイッチドネットワークに配置された。すべてのマシンが同じホスティング施設に設置されていたため、どのペアのマシン間の往復時間も1ミリ秒未満であった。

4GBのメモリのうち、クラスタ上で実行される他のタスクでは、約1-1.5GBが予約されていた。プログラムは、CPU、ディスク、ネットワークがほとんどアイドル状態だった週末の午後に実行された。

5.2 グリープ

grepプログラムは10^{10}個の100バイトレコードをスキャンし、比較的まれな3文字パターンを検索する(パターンは92,337レコードに出現する)。入力は約64MB個(M = 15000)に分割され、出力全体が1つのファイル(R = 1)に配置される。

図2は、計算の経時的な経過を示したものである。Y軸は入力データがスキャンされる速度を示す。このMapReduceの計算に割り当てられるマシンが増えるにつれて、このレートは徐々にピックアップし、1764人のワーカーが割り当てられたときに30GB/s以上でピークに達する。マップタスクが終了すると、レートは下がり始め、計算開始の約80秒後にゼロにヒットする。全計算は開始から終了まで約150秒かかる。これには、約1分間のスタートアップのオーバーヘッドが含まれる。オーバーヘッドは、すべてのワーカーマシンにプログラムを伝播させること、および、1000個の入力ファイルのセットを開き、局所性最適化に必要な情報を得るためにGFSと相互作用する遅延によるものである。

5.3 ソート

ソートプログラムは10^{10}個の100バイトレコード(約1テラバイトのデータ)をソートする。このプログラムはTeraSortベンチマーク[10]をモデルとしている。

ソートプログラムは50行以下のユーザーコードで構成される。3行のMap関数は、テキスト行から10バイトのソートキーを抽出し、キーと元のテキスト行を中間キー/値のペアとして出力する。

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

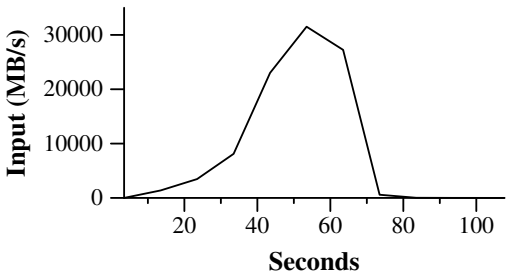


Figure 2: Data transfer rate over time

5.2 Grep

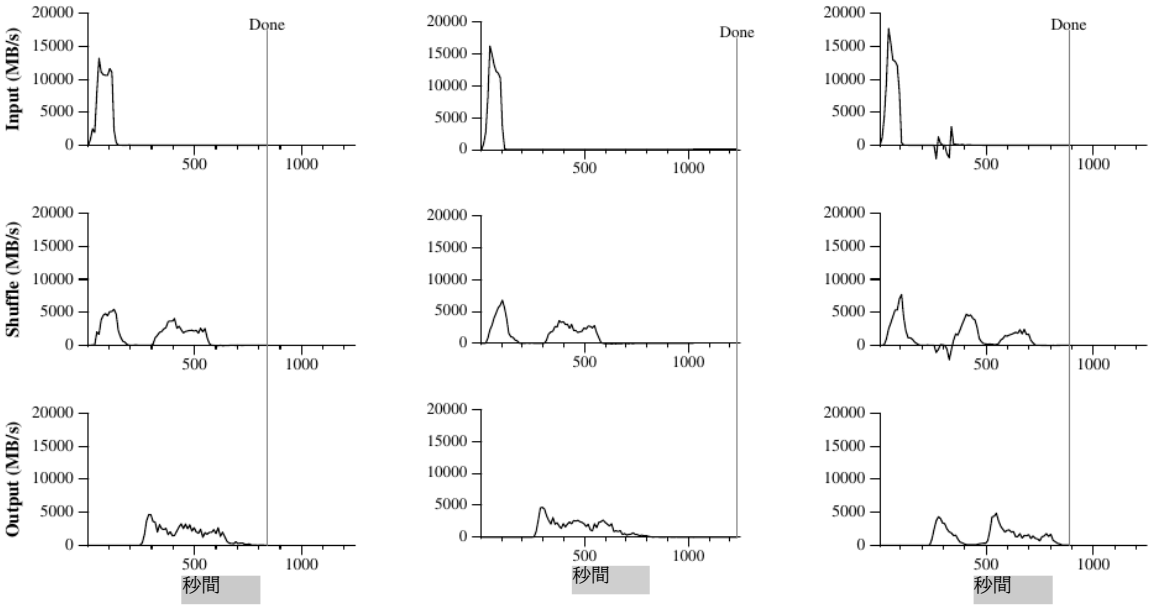
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) 通常の実行　(b)予備タスクなし　(c) 200タスクの死亡

図3:ソートプログラムの異なる実行における経時的なデータ転送率

Reduce演算子として、内蔵のIdentity関数を使用した。この関数は、中間キー/値のペアを出力キー/値のペアとして変更せずに渡す。最終的にソートされた出力は、2ウェイ複製されたGFSファイルのセットに書き込まれます(つまり、2テラバイトがプログラムの出力として書き込まれます)。

先ほどと同様に、入力データは64MB個に分割される(M = 15000)。ソートされた出力を4000ファイル(R = 4000)に分割する。パーティショニング関数は、キーの初期バイトを使用して、R個のピースのいずれかに分離する。

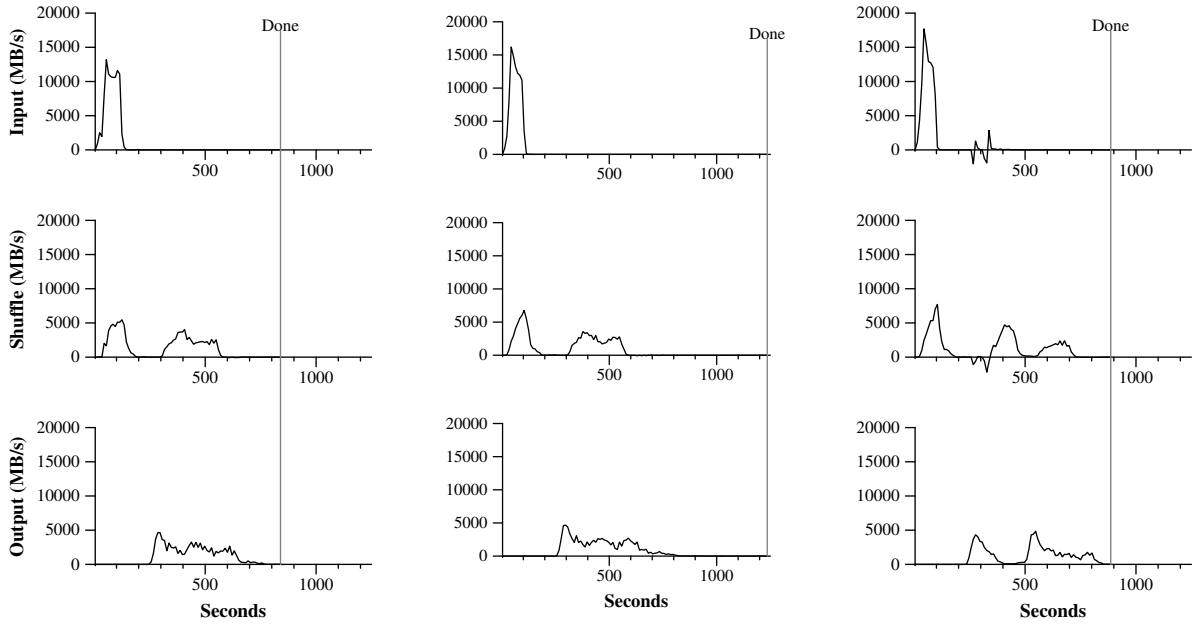
このベンチマークに対する我々のパーティショニング関数は、キーの分布に関する知識を内蔵している。一般的なソートプログラムでは、キーのサンプルを収集し、サンプリングされたキーの分布を使用して、最終的なソートパスの分割点を計算するプリパスMapReduce操作を追加する。

図3(a)はソートプログラムの通常の実行の進行状況を示している。左上のグラフは、入力が読み込まれる速度を示している。このレートは約13GB/sでピークに達し、200秒が経過する前にすべてのマップタスクが終了するため、かなり早く消滅する。入力レートはgrepより小さいことに注意してください。これは、ソートマップタスクが約半分の時間を費やし、I/O帯域幅がローカルディスクに中間出力を書き込むためである。grepに対応する中間出力は無視できるサイズであった。

左中段のグラフは、地図タスクから縮小タスクへのデータ送信の割合を示している。このシャッフルは、最初のマップタスクが完了するとすぐに開始される。グラフの最初のこぶは、約1700のリデュースタスクの最初のバッチです(MapReduce全体には約1700台のマシンが割り当てられ、各マシンは一度に最大1台のリデュースタスクを実行する)。計算のおよそ300秒後、これらの最初のバッチのリデュースタスクの一部が終了し、残りのリデュースタスクのデータのシャッフルを開始する。シャッフルはすべて計算開始から約600秒後に行われる。

左下のグラフは、reduceタスクによってソートされたデータが最終的な出力ファイルに書き込まれる割合を示している。最初のシャッフル期間の終わりから書き込み期間の開始までの間に、機械が中間データのソートに忙しくなるため、遅延が発生する。ライティングはしばらくの間、約2-4GB/sの速度で続行される。すべての書き込みは、計算の約850秒後に終了する。起動オーバヘッドを含めると、全体の計算時間は891秒である。これは、TeraSortベンチマーク[18]で報告されている現在の最高結果である1057秒と同様である。

いくつかの注意点:入力レートはシャッフルレートや出力レートよりも高いが、これは局所性最適化のためである - ほとんどのデータはローカルディスクから読み取られ、比較的帯域幅の制約を受けたネットワークをバイパスしている。出力フェーズがソートされたデータの2つのコピーを書き込むため、シャッフル率は出力率より高くなります(信頼性と可用性の理由から、出力の2つの複製を作成します)。というのも、これが我々の基礎となるファイルシステムによって提供される信頼性と可用性のメカニズムだからである。データを書き込むために必要なネットワーク帯域幅は、基礎となるファイルシステムが複製ではなく消去符号化[14]を使用する場合、削減される。



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 バックバックタスクの効果

図3(b)に、バックアップタスクを無効にしたソートプログラムの実行を示す。実行の流れは図3(a)と同様であるが、書き込みアクティビティがほとんど発生しない非常に長いテールがある。960秒後、5つのリデュースタスクを除くすべてのタスクが完了する。しかし、この最後の数人のストラグラーは300秒後まで終了しない。計算全体には1283秒かかり、経過時間は44%増加した。

5.5 機械の故障

図3(c)では、1746個のワーカープロセスのうち200個を意図的に計算中に殺したソートプログラムの実行を示している。基礎となるクラスタスケジューラは、これらのマシン上で直ちに新しいワーカープロセスを再開した(プロセスのみが死亡するため、マシンはまだ正常に機能していた)。
作業員の死亡は、以前に完了した地図作業の一部が消滅し(対応する地図作業員が死亡しているため)、負の入力率として現れ、やり直す必要がある。このマップワークの再実行は比較的早く行われる。起動のオーバーヘッドを含め、全計算は933秒で終了します(通常の実行時間で5%の増加だけです)。

6 経験値

我々は2003年2月にMapReduceライブラリの最初のバージョンを書き、2003年8月に局所性の最適化、ワーカーマシン間のタスク実行の動的負荷分散など、MapReduceライブラリを大幅に強化した。それ以来、我々はMapReduceライブラリが我々が取り組む種類の問題に対して、いかに広く適用可能であったかに快く驚いている。Google 内では、以下のような幅広いドメインで使用されています:

- 大規模な機械学習問題
- Google NewsとFroogle製品のクラスタリング問題、
- 人気のあるクエリ(例:Google Zeitgeist)のレポートを作成するために使用されるデータの抽出、
- 新しい実験や製品のためのウェブページの特性の抽出(例:局所的な検索のためのウェブページの大規模なコーパスからの地理的な位置の抽出)、および
- 大規模なグラフ計算を行う。

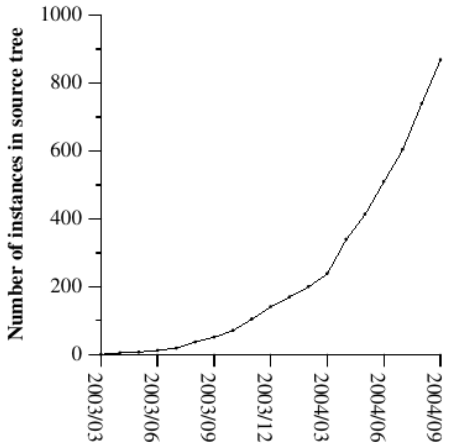


図4:MapReduceインスタンスの経時変化

ジョブの数	29,423
平均ジョブ完了時間	634 secs
使用機械日数	79,186 days
入力データ読み出し	3,288 TB
中間データの作成	758 TB
出力データ	193 TB
ジョブあたりの平均ワーカーマシン数	157
労働者の1人当たりの平均死亡者数	1.2
ジョブあたりの平均マップタスク数	3,351
ジョブあたりの平均削減タスク数	55
ユニークなマップの実装	395
ユニークなreduceの実装	269
ユニークマップ/リデュースの組み合わせ	426

表1:2004年8月に実行されたMapReduceジョブ

図4は、2003年初頭の0件から2004年9月下旬の約900件まで、時間の経過とともに、我々の一次ソースコード管理システムにチェックインされた個別のMapReduceプログラムの数が大幅に増加していることを示している。MapReduceがこれほど成功したのは、簡単なプログラムを書き、30分以内に1000台のマシンで効率的に実行できるようになり、開発とプロトタイピングのサイクルが大幅にスピードアップしたからである。さらに、分散システムや並列システムの経験がないプログラマが、大量のリソースを簡単に利用できるようにする。
各ジョブの終了時に、MapReduceライブラリは、そのジョブが使用する計算リソースに関する統計情報を記録する。表1では、2004年8月にGoogleで実行されたMapReduceジョブのサブセットの統計情報を示している。

6.1 大規模インデックス作成

MapReduceのこれまでの最も重要な用途の1つは、Googleウェブ検索サービスに使用されるデータ構造を生成するプロダクション・インデックス・システムの完全な書き換えである。

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

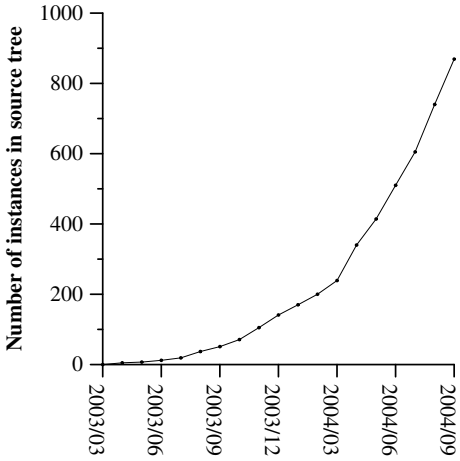


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

インデックス作成システムは、クローリングシステムによって取得された、GFSファイルのセットとして格納された大規模な文書セットを入力とする。これらの文書の生コンテンツは20テラバイト以上のデータである。インデックス作成プロセスは、5～10回のMapReduce操作のシーケンスとして実行される。MapReduceを使用することで、(インデックス作成システムの前バージョンのアドホック分散パスの代わりに)いくつかの利点が得られます:

- インデックス作成コードは、フォールトトレランス、ディストリビューション、並列化を扱うコードがMapReduceライブラリに隠されているため、よりシンプルで、より小さく、理解しやすい。例えば、MapReduceで表現すると、計算の1フェーズのサイズが約3800行のC++コードから約700行に減少した。
- MapReduceライブラリの性能は十分であり、概念的に無関係な計算を、データに対する余分なパスを避けるために、それらを混ぜ合わせるのではなく、分離したままにしておくことができる。これにより、インデックス作成プロセスの変更が容易になる。例えば、私たちの古いインデックス作成システムで数ヶ月を要した変更は、新しいシステムで実装するのに数日しかかからなかった。
- 機械の故障、マシンの遅さ、ネットワークのしゃっくりなどによる問題のほとんどは、オペレータの介入なしに MapReduce ライブラリによって自動的に処理されるため、インデックス作成プロセスはより簡単に運用できるようになった。さらに、インデックスクラスタに新しいマシンを追加することで、インデックス作成プロセスの性能を向上させることが容易である。

7 関連研究

多くのシステムが制限付きプログラミングモデルを提供し、その制限を利用して自動的に計算を並列化している。例えば、連想関数は、並列プレフィックス計算を使用して、N個のプロセッサ上で対数N時間でN個の要素配列のすべてのプレフィックスに対して計算することができます[6, 9, 13]。MapReduceは、大規模な実世界計算の経験に基づいて、これらのモデルのいくつかを単純化し、蒸留したものと考えることができる。さらに重要なことは、数千のプロセッサに対応するフォールトトレラントな実装を提供することである。これに対して、並列処理システムの多くは、より小さなスケールでしか実装されておらず、機械の故障を処理する詳細はプログラマに委ねられている。

バルク同期プログラミング[17]といくつかのMPIプリミティブ[11]は、プログラマが並列プログラムを書きやすくする高次の抽象化を提供する。これらのシステムとMapReduceの重要な違いは、MapReduceが制限されたプログラミングモデルを利用して、ユーザープログラムを自動的に並列化し、透過的なフォールトトレランスを提供することである。

我々の局所性最適化は、I/Oサブシステムやネットワークに送信されるデータ量を減らすために、計算をローカルディスクに近い処理要素に押し込むアクティブディスク[12, 15]などの技術からヒントを得ている。ディスクコントローラプロセッサで直接実行する代わりに、少数のディスクを直接接続するコモディティプロセッサで実行するが、一般的なアプローチは同様である。

我々のバックアップタスク機構は、シャーロットシステム[3]で採用されているイーガースケジューリング機構に類似している。単純なイーガースケジューリングの欠点の1つは、与えられたタスクが繰り返し失敗を引き起こす場合、計算全体が完了しないことである。この問題のいくつかの例を、悪いレコードをスキップする我々のメカニズムで修正する。

MapReduceの実装は、大規模な共有マシンのコレクション上でユーザータスクを配布・実行する責任を負う社内のクラスタ管理システムに依存している。本稿の焦点ではないが、クラスター管理システムは、Condor [16]のような他のシステムと精神的に似ている。

MapReduceライブラリの一部であるソート機能は、NOW-Sort [1]と同様の操作である。ソースマシン(マップワーカー)は、ソートされるデータを分割し、R個のリデュースワーカーのいずれかに送る。各リデュースワーカーはローカルにデータをソートする(可能であればメモリに)。もちろん、NOW-Sortには、我々のライブラリを広く適用できるような、ユーザー定義のMap関数とReduce関数がない。

River [2]は、分散キューにデータを送信することで、プロセスが互いに通信するプログラミングモデルを提供している。MapReduceのように、リバーシステムは、異種ハードウェアやシステム摂動によってもたらされる非一様な存在下でも、良好な平均ケース性能を提供しようとする。Riverは、円盤とネットワークの転送を注意深くスケジューリングすることで、バランスの取れた完了時間を達成する。MapReduceは異なるアプローチを持っている。プログラミングモデルを制限することで、MapReduceフレームワークは問題を多数の細かいタスクに分割することができる。これらのタスクは、利用可能なワーカーに動的にスケジューリングされるため、より速いワーカーはより多くのタスクを処理する。制限付きプログラミングモデルはまた、仕事の終わり近くにタスクの冗長な実行をスケジュールすることを可能にし、非一様性(遅い作業員やスタックした作業員など)が存在する場合の完了時間を大幅に短縮する。

BAD-FS[5]はMapReduceとは全く異なるプログラミングモデルを持ち、

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

MapReduceとは異なり、広域ネットワーク全体のジョブの実行を対象としている。しかし、根本的な類似点は2つある。(1) 両システムとも、障害によるデータ損失から回復するために冗長な実行を使用する。(2)両者とも、混雑したネットワークリンクを横切るデータ送信量を減らすために、局所性を意識したスケジューリングを使用する。

TACC [7]は、高度に利用可能なネットワークサービスの構築を簡素化するために設計されたシステムである。MapReduceと同様に、フォールトトレランスを実装するメカニズムとして再実行に依存する。

8 結論

MapReduceプログラミングモデルは、Googleで様々な目的で成功裏に使用されている。この成功はいくつかの理由によるものである。第一に、並列化、フォールトトレランス、局所性最適化、負荷分散などの詳細が隠されているため、並列分散システムの経験がないプログラマーでも、このモデルは使いやすい。第二に、多種多様な問題はMapReduce計算として容易に表現できる。例えば、MapReduceは、Googleのプロダクションウェブ検索サービス、ソート、データマイニング、機械学習、その他多くのシステムのデータ生成に使用されている。第三に、数千台のマシンからなるマシンの大規模なクラスタに対応する MapReduce の実装を開発した。この実装は、これらのマシンリソースを効率的に利用するため、Googleで遭遇する多くの大規模な計算問題に使用するのに適している。

この研究からいくつかのことを学んだ。まず、プログラミングモデルを制限することで、計算の並列化や配布が容易になり、そのような計算をフォールトトレラントにする。第二に、ネットワーク帯域幅は希少な資源である。したがって、我々のシステムにおける多くの最適化は、ネットワーク全体で送信されるデータ量を減らすことを目標としている。局所性最適化は、ローカルディスクからデータを読み取ることを可能にし、中間データの1コピーをローカルディスクに書き込むことで、ネットワーク帯域幅を節約する。第三に、冗長実行は、低速マシンの影響を低減し、マシンの故障やデータ損失を処理するために使用することができる。

謝辞を述べる

Josh Levenbergは、MapReduceの使用経験や他の人々の強化提案に基づき、ユーザーレベルのMapReduce APIを多くの新しい機能で修正・拡張するのに役立っている。MapReduceは、その入力を読み、その出力をGoogleファイルシステム[8]に書き込む。モヒト・アロン、ハワード・ゴビオフ、マルクス・グッチケに感謝したい、David Kramer、Shun-Tak Leung、Josh Redstone。また、MapReduceで使用されているクラスタ管理システムの開発に携わったPercy LiangとOlcan Sercinogluに感謝したい。Mike Burrows、Wilson Hsieh、Josh Levenberg、Sharon Perl、Rob Pike、Debby Wallachは、本論文の初期の草稿に対して有益なコメントを寄せてくれた。匿名のOSDI査読者と我々の羊飼いであるEric Brewerは、論文を改善できる分野について多くの有益な示唆を与えてくれた。最後に、有益なフィードバック、提案、バグレポートを提供してくれたGoogleのエンジニアリング組織内のMapReduceの全ユーザーに感謝する。

参考文献

[1] アンドレア・C・アルパチ=デュソー、レムジ・H・アルパチ=デュソー、デビッド・E・カラー、ジョセフ・M・ヘラースタイン、デビッド・A・パターソン。ワークステーションのネットワークにおける高性能ソート。1997年ACM SIGMODデータ管理国際会議予稿集、アリゾナ州ツーソン、1997年5月。

[2] レムジ・H・アルパシ=デュソー、エリック・アンダーソン、ノア・トレアフト、デビッド・E・カラー、ジョセフ・M・ヘラースタイン、デビッド・パターソン、キャシー・エリク。I/Oを川とクラスター化する: 高速ケースを共通にする。第6回並列分散システムにおける入力/出力に関するワークショップ(IOPADS '99)予稿集、10-22ページ、ジョージア州アトランタ、1999年5月。

[3] アラシュ・バラトロ、メメット・カラウル、ズヴィ・ケデム、ピーター・ウィコフ。シャーロット:ウェブ上のメタコンピューティング第9回並列分散コンピューティングシステム国際会議講演論文集, 1996.

[4] ルイズ・A・バローゾ、ジェフリー・ディーン、ウルス・ヘルツル。惑星のウェブ検索 Googleクラスタアーキテクチャ。IEEE Micro, 23(2):22-28, 2003年4月.

[5] ジョン・ベント、ダグラス・タイエン、アンドレア・C・アルパシ=デュソー、レムジ・H・アルパシ=デュソー、ミロン・リブニー。バッチ対応分散ファイルシステムにおける明示的な制御。第1回USENIXシンポジウム「ネットワークシステムの設計と実装」NSDI予稿集、2004年3月。

[6] Guy E. Blelloch. 原始的な並列操作としてのスキャン。IEEE Transactions on Computers, C-38(11), 1989年11月.

[7] アルマン・フォックス、スティーブン・D・グリブル、ヤティン・チャワテ、エリック・A・ブリュワー、ポール・ゴーティエ。クラスターベースのスケーラブルなネットワークサービス第16回ACMオペレーティングシステム原理シンポジウム予稿集、78-91ページ、フランス、サンマロ、1997年。

[8] サンジャイ・ゲマワット、ハワード・ゴビオフ、シュン・タク・レオン。Googleファイルシステム。第19回オペレーティングシステム原理シンポジウム、29-43ページ、ニューヨーク州ジョージ湖、2003年。

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. スキャンと他のリスト同型の体系的な効率的並列化。L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par'96. 並列処理、コンピュータサイエンスのレクチャーノート1124、ページ401-408。Springer-Verlag, 1996.

[10] Jim Gray. ベンチマークホームページをソートする。http://research.microsoft.com/barc/SortBenchmark/.

[11] ウィリアム・グロップ、ユーイング・ラスク、アンソニー・スケルムMPIの使用: メッセージパッシングインターフェースによるポータブル並列プログラミング。MITプレス、ケンブリッジ、マサチューセッツ州、1999年。

[12] L. (1)言語処理、(2)言語処理、(3)言語処理、(4)言語処理、(5)言語処理、(6)言語処理。ダイヤモンド: 対話型検索における早期廃棄のためのストレージアーキテクチャ。2004年USENIXファイル&ストレージ技術FAST会議予稿集, 2004年4月.

[13] リチャード・E・ラドナー、マイケル・J・フィッシャー並列接頭辞計算. ACM, 27(4):831-838, 1980.

[14] マイケル・O・ラビン. セキュリティ、負荷分散、フォールトトレランスのための効率的な情報分散。ACM, 36(2):335-348, 1989.

[15] エリック・リーデル、クリストス・ファルアウトソス、ガース・A・ギブソン、デビッド・ナグル。大規模データ処理のためのアクティブディスクIEEEコンピュータ、68-74ページ、2001年6月。

[16] ダグラス・タイン、トッド・タネンバウム、ミロン・リヴニー。分散コンピューティングの実践: コンドルの経験。同時実行と計算: Practice and Experience, 2004.

[17] L. G. Valiant. 並列計算のための橋渡しモデル.ACM通信, 33(8):103-111, 1997.

[18] Jim Wyllie. Spsort: テラバイトを素早くソートする方法。http://almel.almaden.ibm.com/cs/spsort.pdf.

A 単語頻度

このセクションには、コマンドラインで指定された入力ファイルのセット中の各ユニークな単語の出現回数をカウントするプログラムが含まれている。

```
#include "mapreduce/mapreduce.h"

// ユーザのマップ関数
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// ユーザーの削減機能
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduce仕様仕様仕様

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  オプション:ネットワーク帯域幅を節約するために、マップ内の部分和を行う // タスク out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_メガバイト(100);

  // 実行
  MapReduce結果;
  (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```