計算機構成論　レポート課題

Google Cluster Architecture

2006-12-15

M06-A04　井上 謙次

大阪工業大学大学院情報科学研究科

1. 緒論

Web 全体の全文検索エンジンは重要な技術である。しかし，Web 上に存在する文書全体が構成する空間は非常に広大で，そのデータ全体を扱う処理は厖大な計算量を必要とする。そのような情報処理を要求される時間内に行うには，単位時間当たりに処理可能な計算量を増やすというハードウェア的な解決案と，処理の特性に応じて必要な計算量自体を減少させるというソフトウェア的・アルゴリズム的な解決案が存在するであろう。

本レポートでは，ハードウェア的な解決案の一つとして，Google の PC クラスタアーキテクチャについて取り上げる。Google は Web に関するリーディングカンパニーであり，並列計算処理を実運用して有益なサービスを展開している。従って，実際の PC クラスタ設計を概観することにより，並列計算環境に関わる様々な知見を得られることが期待できる。

2. アーキテクチャの概要

Google のクラスタアーキテクチャは完全に in-house で，つまり Google 社内の技術者が設計，開発，運用を行っている。そして，そのアーキテクチャは低レベルの階層ではコモディティ PC（一般的に流通している価格の比較的低いパーソナルコンピュータ）やオープンソースソフトウェアを多く活用しながら，その上の高レベルの階層では独自開発したデータベースシステムや並列計算用プログラミングモデルを載せることで全体として柔軟性や拡張性を持ち，かつ効率的なシステムに仕上がっているといえよう。

本レポートでは，おおまかに

- ハードウェア構成
- ネットワーク構成
- ファイルシステム（メモリモデル）
- 並列プログラミングモデル

に分けてアーキテクチャを概観する。なお，本レポートは 2003 年から 2004 年に書かれた論文をもとに記しており，記載する数字等はその当時のものである。また，現在の Google の PC クラスタアーキテクチャは本レポートの内容とは異なっているかもしれないことを注記しておく。

3. ハードウェア

Google の PC クラスタは 10,000 を超える commodity クラスの PC によって，そして基本的にはそれらの commodity クラスの PC のみによって，構成されている。それら commodity クラスの PC とは，例えば 2004 年の時点で典型的に使われていたものであれば，

- 2 つの 2 GHz Intel Xeon プロセッサ
- 2 つの 160GB IDE ドライブ
- 4GB のメインメモリ

- 100M bps もしくは 1G bps の Ethernet

というような，一般的に販売されている安価な PC である。これらの安価なハードウェアの上に，OS としてオープンソースの Linux が動いている。従って，全てベンダに依存しないオープンスタンダードの技術を採用しているといってもよい。

上述したような commodity クラスの PC は，1 台 1 台の PC は信頼性が低い。それを使うのはなぜか。逆に言えば，なぜ信頼性の高いハイエンドのサーバを使わないのか。彼らの主要な評価基準は次の二つである。

- 運用全体のコストパフォーマンス比の最大化
- 処理できるスループットの最大化（ピークパフォーマンスの最大化ではない）

ハイエンドサーバに対する commodity クラス PC のコストパフォーマンス比の高さというのは，マシン性能だけ比較すると単純である。すなわち，信頼性を確保するための費用を支払っていない分，同じ費用で得られるマシン性能は commodity クラスの PC の方が高い。しかし，信頼性の低い PC は壊れやすく，保守するマシンの数も多くなるため，メンテナンスコストが高くなる。結局のところ，Google は commodity クラスの PC を採用した方がコストパフォーマンスが高いと判断し，信頼性の低さが問題とならないクラスタリングシステムを構築したのであるが，その判断基準は Google のサービス内容にも大きく依存している。すなわち，Google においては限られた少数のアプリケーションが重要であり，汎用的なクラスタリングシステムを採用する必要はない。特に，彼らの提供する Web サービスでは，各マシンのピークパフォーマンスを維持する必要はない。すなわち，クエリーに対する応答速度を最大化しなくてもよい。そのような応答速度は，並列化処理を割り当てる PC の数を増やせば調整できるのであって，個々のマシンのピークパフォーマンスというのは重要ではなく，全体のスループットを最大化することが Google のアプリケーションにとって意味を成す。

Google のクラスタアーキテクチャは彼らの問題を効率的に解くためのアーキテクチャである。

4. ネットワーク

詳しい言及はないが，PC クラスタのネットワークアーキテクチャは 2 階層のツリー構造になっているようだ（図 1）。ただし，これはデータ処理（例えばユーザからの検索クエリなど）用の PC クラスタであって，Web サーバやそのロードバランサは別途存在する。

各 PC は 100M bps もしくは 1G bps の Ethernet 接続を有し，100M bps の Ethernet スイッチを介して互いに接続されている。また，それら 100M bps の Ethernet スイッチは 1 本から 2 本の Gigabit Ethernet で上流のコアネットワークに接続されている。コアネットワークは Gigabit Ethernet スイッチで接続されている。

おそらく，同じ（下流の）スイッチに接続されている PC 群は，一つのサーバラックに収納されているのではないかと推測される。それら同一スイッチに接続されている PC 間の通信は，当然ながら，そうでない通信に比べて高速である。そこで，並列処理においては，可能な限り近くにある PC 間の通信で済むように PC に処理を割り当てることで locality optimization（局所性最適化）が行われている。

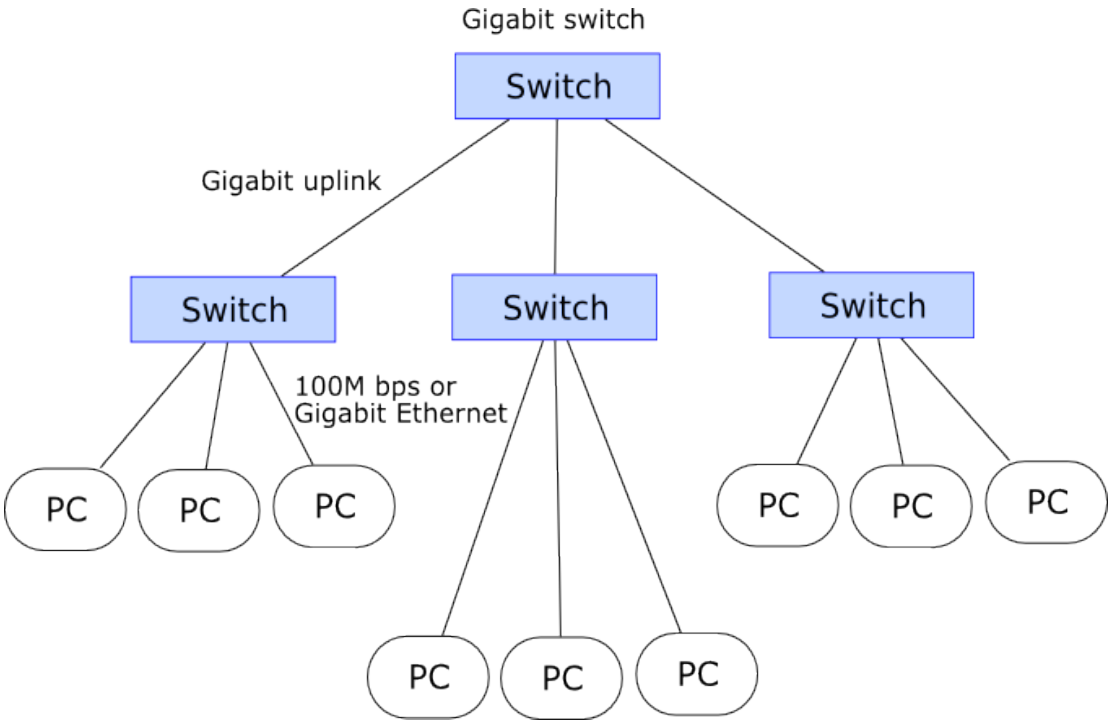

図 1　2 階層のツリー状ネットワーク

5. ファイルシステム

Google の PC クラスタで使われるファイルシステムは Google File System (GFS)と呼ばれる分散ファイルシステムである。3 節で取り上げたように，個々の PC にハードディスクドライブが付属しており，全体のデータはそれら各 PC のハードディスク上に分散して配置されている。

各 PC のハードディスクドライブは IDE が採用されており，RAID されているとしても壊れやすい，信頼性の低いデバイスである。そこで，データには必ず複製(replica)が用意されている（典型的には同じデータは 3 つ存在するようである）。そのような冗長化によってデータが失われないようになっていると共に，前節で述べた並列処理時における locality optimization の効果も上げている。

また，常に故障がないかをチェックし続ける必要があり，ハードディスクの故障の検知・復旧作業は全て自動化されている。10,000 台を超える PC クラスタであるから，おそらく毎日一つの IDE ハードディスクドライブが故障したとしてもおかしくはない。それらの作業はまさに”the norm rather than exception”（例外というよりは標準である）という言葉通りであろう。

GFS のアーキテクチャは，中央の司令塔となる master が一つと，実際にデータを保存する多くの chunkserver から構成される。master はファイルシステムのメタデータを全て扱い，またシステム全体に渡る操作のコントロールやガーベージコレクションなども扱う。しかし，実際のデータの転送は master を介さずに行われる。chunkserver は Linux の通常のファイルとしてデータを保存する（chunkserver の扱うデータの chunk サイズ，すなわちブロックサイズは 64MB に固定されている）。

データ処理の効率性の点から，データのメモリ内保持や Linux のキャッシュシステムなどを含め，広範な考察が行われてアーキテクチャが決定されている。また，当然データの consistency（一貫性）については保証されている。それらについて本レポートでは詳しく述べない。

特筆すべき点としては，サービスを提供するのに必要なアルゴリズムや並列計算処理も含めて，全体のアーキテクチャの中で「ネットワークを流れる通信量を減らす」という意図を持ってファイルシステムが設計されている点であろう。そのキーとなるアイデアが Locality optimization であり，各 PC のローカルディスク上にデータが保存されている点が重要である。すなわち，データベースサーバからデータを取ってきてクライアントがそのデータを処理する，というアーキテクチャではない。処理が必要なデータを持っている PC がその処理を任せられるのである（正確には，データを保持しているその PC が busy な場合は可能な限りその近くの PC が処理を割り当てられる）。データベースにプロセッサが付いているともいえるし，プロセッサにデータベースが付いているとも表現できよう。それら 2 つの要素を分離しないことによって，分散ファイルシステムとしての利点を活かし，必要な通信量を大きく削減することに成功している。

6. 並列処理

並列処理の基本は，ある問題を互いに独立な部分問題に分けて解き，最後にその結果を合わせることである。通常，そのような部分問題への分割は人間の判断によって行われる。並列計算用にアルゴリズムを最適化し，注意深く実装しなければならない。並列計算アルゴリズムは一般的にはアプリケーション依存であり，それぞれのアルゴリズムを考案し実装するまでにかかる時間は大きい。

その制約は Google の PC クラスタアーキテクチャにおいても同様である。しかし，Google は MapReduce と呼ばれる並列処理用のプログラミングモデルを開発することで，並列処理アルゴリズムの実装を非常に簡略化することに成功した。Google では MapReduce を用いた並列処理だけが行われているわけではないが，非常に興味深い事例であるので，本レポートでは MapReduce のみを取り上げることにする。

MapReduce のアイデアは，プログラミングモデルを「自動的に並列処理化が可能な要素のみ」に制約することである。具体的には，そのような要素とは map 関数と reduce 関数の 2 つであり，それが MapReduce の名称の由来となっている。それらの要素は自動的に並列処理化ができるのであるから，そのような並列処理化を自動で行うライブラリを一つ実装しさえすれば，MapReduce プログラミングモデルに従うアルゴリズムは全て効率的に並列処理として実行できる。そのような制約されたプログラミングモデルでは当然あらゆる処理を書けるわけではない。しかし，もし解こうとしている問題がそのようなプログラミングモデルで記述できるのであれば，最も時間のかかるアルゴリズムの並列処理化を考案して実装する必要はなく，開発に必要な時間が大幅に短縮される。

MapReduce では map 関数と reduce 関数のみが並列計算用に使用可能である（実際には他にも存在するが，本質的な差はないので説明は省く）。並列計算用に使用可能であるとは，map や reduce が一つの単位として並列処理化されるという意味であり，それらの関数内では（引数として与えられたデータ内で完結する以上は）自由にプログラムを記述してもよい。

map や reduce は多くのプログラミング言語に採用されている基本的なリスト処理演算用の高階関数であるので，ここでは説明を省く。ユーザが記述するのはそれらの高階関数に引数として渡す関数である（従って，ユーザが記述する関数には全体のデータ集合のうちの一つがイテレーティブに渡される）。map 関数は「キーと値のペア」を入力とし，中間「キーと値のペア」値の集合を出力とする。reduce 関数は同じ中間キーに関連付けられた値を全てマージする。map 関数と reduce 関数の型は次のようになっている。

```
map        (key1, value1)        →   list(key2, value2)
reduce     (key2, list(value2)   →   list(value2)
```

このプログラミングモデルで処理が可能な例として，以下のものが挙げられている。

- 単語の出現数の数え上げ (word occurrences)
- 逆インデックス (inverted index)
- Web の逆リンクグラフ (reverse web-link graph)
- 分散 grep (distributed grep)
- 分散ソート (distributed sort)

理解の促進のため，“MapReduce: Simplified Data Processing on Large Clusters”に記載がある「単語の出現数の数え上げ」を行う例（擬似言語によるコーディング）を以下に掲載する[3, p.2]。

```
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));
```

上記の例では，map 関数によって各文書内にある単語（キー）を取り出してその各々にカウント 1（値）を出力し，reduce 関数によって同じ単語（キー）のカウント値（値）を全て足し合わせることで，それぞれの単語に対する総出現数が求められている。

次に，この MapReduce モデルで記述されたプログラムの実行がどのように制御されるかについて完結に説明する。MapReduce プログラムの実行は一つの master と，master によって任命される複数の worker によって行われる。master は中央司令塔の役割を果たし，それぞれの worker との調整（通信）を行うだけで，master それ自身はデータを処理しない。master は必要に応じて map 処理と reduce 処理を busy 状態ではない PC に割り当てる（それらをそれぞれ map worker，reduce worker と呼ぶ）。

まず，master は入力となる全データ（テラバイトのオーダーに及ぶ）を 64MB 程度のブロックに分け，分割されたそれぞれの入力ファイルに対して map worker を割り当てる。次に map worker は GFS から実際の入力データを読み取って map 処理を適用し，自身のローカルディスクに中間データを書き出す。なお，locality optimization により，可能であれば入力データを持つ PC 自身が map worker として割り当てられることになるので，その場合はこの処理によって入出力データ自体の転送は全く起こらないことに注意を喚起されたい。次に，中間データが生成され次第，master は reduce worker を順次割り当てていき，reduce worker は中間データを読み取って reduce 処理を適用し，最終結果を出力する。この並列計算処理は，GFS というファイルシステムに強く依存する

ことで，locality optimization を行ってネットワークに流れる転送量を少なくし，効率的な処理を行っている。

並列計算処理には様々な工夫がなされているが，ここでは worker PC の故障，そしてバックアップタスクについてのみ簡単に説明する。もし worker PC に障害が発生した場合，master は自動的にそれを検知し（一定間隔で ping を行う），その worker に割り当てていた処理を自動的に他の worker に割り当てるようにする。なお，map タスクにおいては，処理の結果は障害が発生した PC のローカルディスクに格納されていて利用できないため，処理の進捗度に関わらず全ての処理を再実行しなければならない。このアーキテクチャは，数十台や百台程度の PC が同時に利用不可能となっても処理が完了できるほどの耐性を持つ。なお，master PC に障害が発生した場合，自動的に復旧せずに単に全ての処理を中断する。これは，1 台しかない master が故障する確率は非常に小さいため，それで問題がないためである。

また，処理自体は正常に行えているにも関わらず，他の worker に比べ非常に遅い worker が存在しうる。例えば，ハードディスクが損傷しており，エラーチェックをしながら読み込むためにデータの読み込み速度が数十倍も遅くなってしまうようなケースが存在する。そのような場合，非常に少数の worker の処理が完了するまで全体としての処理が終了しない（そのような最後まで残った worker は”straggler”と呼ばれる）。全体の処理の終了時間が最後まで残った少数の straggler に依存してしまう結果となる。そこで，処理が残り少なくなった際に，同じタスクを複数の worker に割り当て，それらのうち最も早く完了した結果を採用する，ということを行う。そのような 2 番目以降の同じタスクをバックアップタスクという。バックアップタスクのメカニズムを採用することで，worker 間に一種の競争原理が働き，少数の straggler によって処理全体の終了時間が影響を受けにくくなる。例として，バックアップタスクを利用することにより，利用しない場合に比べ 44%早く処理（1 テラバイトのソート）が終了したことが挙げられている。

7. 結論

今まで見てきたように，Google の PC クラスタアーキテクチャは，ハードウェア，ネットワーク構成，ファイルシステム，ソフトウェア，アルゴリズム，プログラミングモデルといった各要素が，安価な commodity クラス PC とオープンソースの Linux 上に，全体として効率的に機能するように構成されている。

その設計には，コストパフォーマンス比の最大化，故障や障害への対応，効率的なファイルシステム，並列計算専用のプログラミングモデルの構築など，多岐に渡るアイデアやトレードオフが存在する。

それらの個々のアイデアは素晴らしいが，実際にそれらのアイデアが効率的に動くシステムとなって構成されているのは，「自分たちの問題を効率的に解く」という方向付けによって着実にまとめられているからであろう。

参考文献

以下の論文は `http://labs.google.com/papers/` **から入手可能である。**

[1] Web Search for a Planet: The Google Cluster Architecture (IEEE, 2003)

[2] The Google File System (ACM, 2003)

[3] MapReduce: Simplified Data Processing on Large Clusters (OSDI, 2004)