Silkによる分散データ処理のストリーミング

斉藤　太郎†

† 東京大学大学院 情報生命科学専攻
E-mail: †leo@xerial.org

あらまし　Silk は並列・分散計算とそのパイプライン処理 (ストリーミング) の融合を目指したフレームワークである。Silk ではプログラム中のデータ（変数）と分散演算 (map, reduce, join など) の依存関係をプログラムの AST から解析し、クラスタの空き状況に応じて実行プランの最適化やデータの分割を行う。また、プログラムの部分実行により、GNU Make のように障害からの回復やプログラムの変更後のパイプラインの再計算が可能であり、変数名を指定した中間データの検索（ワークフロークエリ）を実現できるのが特徴である。本研究では Silk のプログラミングモデルとシステムの設計・実装について紹介し、その応用について議論する。

キーワード　分散データベース、クラウドコンピューティング

1. はじめに

Silk は並列分散計算とそのパイプライン処理（ストリーミング）の融合を目指したフレームワークであり、Scala のライブラリとして実装されている。以下に Silk でのプログラミング例を示す。

```
val in = loadFile("input.sam")
val size = in.lines
     .map(_.split("\t"))
     .filter(_(2) == "chr1")
     .size
```

このコードは `input.sam` というテキスト形式のファイルを開き (`loadFile`)、各行を読んで (`lines`) タブで区切り (`split`)、2 列目の要素が chr1 となるものを取り出して (`filter`) その数をカウントする (`size`) 計算を表している。ゲノム情報処理ではこのような簡単なデータ処理を日常的に行うことが多い。

小さなテキストファイルでは問題にはならないが、これが 100GB を超えるとしたらどうであろう。秀でたプログラマはマルチコアで計算するためのスレッド処理を加えるであろうし、分散計算に詳しい技術者なら Hadoop の MapReduce プログラムに置き換えるかもしれない。データベース技術者なら RDBMS 用にテキストを整形し、SQL 文で問い合わせを記述するであろう。

しかし、実行方式は違えど、行いたい計算は既に上記の小さなプログラムに記述されている。Silk でのチャレンジは、計算環境（マルチコア、クラスタ環境、メモリ容量、共有ファイルシステムの有無など）に合わせて、コードはそのままに実行方式のみを切り替えることにある。Silk は分散計算を利用するプログラマ・データサイエンティストの負担を軽減することを念頭にデザインされたシステムで、実装とその開発はオープンソースで GitHub 上 (https://github.com/xerial/silk) で行われている。

2. ゲノムサイエンスにおけるデータ処理

Silk のようなフレームワークが必要になった背景を、ゲノムサイエンスの分野における例を題材に説明していく。ゲノムデータの処理では、データベース化される前の大量のテキストデータを扱う必要がある。例えば、次世代シーケンサー (Illumina, PacBio 社が開発しているもの）と呼ばれる配列解読器の進化により A, C, G, T からなる DNA 配列（ゲノム）を低コストで大量に読むことができるようになり、病気の原因などを探る材料として広く活用されている。シーケンサーから出力される情報は扱いやすいテキスト形式（あるいはそれを圧縮したもの）が用いられている。ゲノムサイエンスでは各種データ解析のためのプログラムも非常に多く開発されている（興味がある読者は Bioinformatics 誌、Nature Methods 誌などを参照すると良い）が、これらのプログラムもテキストベースのフォーマット (FASTA, FASTQ, GFF, BED, SAM, WIG, VCF など) を入出力に採用している。これは、汎用的な書式でデータのスキーマを定義し、研究者間のデータのやり取りを円滑にする文化がコミュニティにあるためである。これにより、データ解析に関わる人材のバックグラウンドの多様さ（プログラミングスキル、言語の好みなど）に広く対応できるようになっている。

一方、既存のデータベースシステム側で扱えるデータ形式は非常に狭い。RDBMS、XML DB, NoSQL など種々のシステムがあるが、現状扱えるデータはリレーショナルモデルかドキュメント型（XML、JSON、raw text など) にほぼ絞られており、ゲノムサイエンスのデータを扱うためにはフォーマットの変換が必須となっている。

2.1 巨大なデータの処理

データを DBMS で扱える形式にする際、まずデータの大きさが障害となる。ヒトゲノム解析では一人分のゲノムデータを 50x で読む場合（ヒトゲノムのサイズは約 3GB で、エラー補正のため通常この 50 倍程度の配列データを読む必要がある)、FASTQ（ゲノム配列＋そのクオリティの値）のテキストデータで約 500GB にもなる（gzip 圧縮するとこの 1/5 程度のサイズになる)。その後、ヒトゲノムにアラインメントする作業で 1.5 倍の SAM データ（テキストで 750GB) が出力される。ETL (extract - trasform - load) でデータベースにデータを取り込

んで解析を行なうにも、まず分散処理が必須なのである。

2.2 多様なデータ解析

ゲノムサイエンスに置ける解析は他種多様で、SQL の枠に収まらない計算も多く必要となる。上手くデータを RDBMS に取り込めたとしても、研究に必要な計算（例えばゲノム配列へのアラインメント (参照)、ゲノムアセンブリなど (参照)）は、通常大規模なメモリを必要とし DBMS 側の機能だけでは十分に実装ができない。また、UDF として DBMS 側に新しい関数を追加することも考えられるが、実装の複雑さに比べてわざわざ DBMS に組み込む利点が少ない。そのため、DBMS を使わず済ますよう独自のデータフォーマットの開発が進んでいる。例えば SAM フォーマット（ゲノムのアラインメント情報のためのデファクトスタンダード）のデータを扱う samtools はデータのバイナリフォーマット（BAM) への変換、ソーティング、インデキシング、区間クエリの実装を独自に行い、各種言語への移植 (picard ライブラリ, bamtools など) も盛んに行なわれている。このようなデータ処理は本来 DBMS が得意とするはずの機能であるが、多様な解析とデータ処理がうまく融合できていないために DBMS を活用できていない事例である。

2.3 データのパイプライン処理

ヒトの疾患関連変異を見つけるための解析では、通常１サンプルでは不十分で、数百〜数万規模のサンプルデータを処理し、その中での共通変異や特異的な変異を見いだし、病気の原因を探る解析が必要になる。このような計算は１パスで処理しきることは難しく、クラスタの障害（ノードの障害、ファイルシステムの不具合、プログラムのバグ、メモリ不足）などが起った場合でも、再実行できるようにデータ処理をパイプライン化しておく必要がある。

また病気の種類により追加で必要な解析も変わってくる。遺伝性の疾患の場合、疾患関連変異を持っていても必ずしも発症する訳ではなく（浸透率)、弧発性疾患の場合は家系を調べても病気の原因がわからない。ゲノムだけではなく、エピゲノム (ゲノム上に乗るデータ全般）のデータ（メチル化、ヒストン修飾など）の状態を次世代シーケンサーを使って調べることもあり、そのような場合にはさらに解析パイプラインが複雑になる。解析結果を見て結果が思わしくなければ、パイプラインの変更、プログラムのパラメータのチューニングなどの処理を施すことも多く、パイプライン処理といっても定型処理ではなく、研究のニーズに応じて拡張されていくものになる。

2.4 解析プログラムのモジュール化

ゲノムサイエンスに限らずデータサイエンスではプログラミングとデータ処理が密結合している。しかし、実際のデータ解析のパイプラインを構築する現場ではコードをモジュール化し疎結合にするアプローチが取られてきた。例えば、UNIX のコマンド群を組み合わせて Makefile のようなパイプラインを書くのがその一例である。タスク間の依存関係はモジュール間の依存関係で表されるが、データサイエンスでは一度しか使わないコードも多く書かれ、プログラム毎にモジュール化するコストの方が大きくなってしまう。さらに、モジュール間の接続（例えば受け渡しするファイル名）やデータの保存形式を間違えると誤った結果が生成されることになるため、モジュール化を支援する仕組みも必要である。

2.5 並列・分散計算習得の難しさ

並列・分散計算のプログラムを書くためには、スレッド、プロセスの知識、ロックによる同期、データのメモリ間共有、プロセス間共有、ソケットなどネットワーク周りの知識などコンピューターサイエンスの学部・大学院レベルの多様な知識が要求される。分散・並列計算の実行方式でも様々な実装がある。C/C++なら OpenMP や MPI、Java では Hadoop 関連のエコシステムのプロジェクト群があるが、専門家以外には運用すら難しくなっている。MPI では通信周りの最適化は施されているが、プログラムの書きやすさの面では、例えば一般のオブジェクトを通信用のデータ構造に使うだけでもオブジェクトのパッケージングが必要となるなど、とたんに記述が難しくなってしまう。IBM が開発している X10 [3] は比較的新しいフレームワークでオブジェクトの通信は大分書きやすくはなっているが、障害対応、パイプライン処理の実行という点は十分にサポートされていない。高度な知識を学んでも手軽に使えるフレームワークが見当たらないのが現状となっている。

分散計算のニーズがゲノムサイエンスのように必ずしもコンピューターサイエンスを専門に学んだ人以外にも広がっている現在では、コンピューターサイエンスと同等の大学教育を分散計算を行う全ての人に習得してもらうのは現実的ではない。むしろこれらの知識を学んだ人が、フレームワークの開発に力を注ぎ、分散計算が手軽になるように尽力すべきだと考える。データサイエンスを行う人の関心はやはり大きな「謎」を解明したり「発見」する側の「サイエンス」にあり、手段としての分散計算（これも「コンピューターサイエンス」として大きなテーマであることに変わりないが）に費やす労力をできるだけ軽減することに価値があるだろう。

3. Silk フレームワーク

前説までで述べたように、データ処理のプログラムには以下に挙げる課題がある

- 巨大なデータの処理
- 多様なデータ解析プログラムの融合
- 解析のモジュール化・パイプライン化
- 障害・拡張への対応
- 習得のしやすさ

Silk ではプログラムモジュール化の手間を軽減するため、light-weigt なモジュールとして、プログラミング言語中の関数をそのままモジュールとして使えるように設計されている。モジュールの依存関係は、Silk では関数の依存関係として表される。プログラムのモジュールも関数も入力を受け取って出力を返すというパラダイム上では同じものである。Scala の簡潔なシンタックスを利用することにより習得も容易になる。Silk のデータ処理の関数を用いるだけなら Scala の複雑な機能まで全て学習する必要はない。

Scala は Java 仮想マシン（JVM) 上で動く言語で、強力なコレクションライブラリ（list, set, map などの基本データ構

関数	演算オブジェクト	結果の型
`in.map[B](f:A=>B)`	`MapOp(in:SilkSeq[A], f:A=>B)`	`SilkSeq[B]`
`in.flatMap[B](f:A=>SilkSeq[B])`	`FlatMapOp(in:SilkSeq[A], f:A=>SilkSeq[B])`	`SilkSeq[B]`
`in.filter(f:A=>Boolean)`	`FilterOp(in:SilkSeq[A], f:A=>Boolean)`	`SilkSeq[A]`
`in.sort(implicit ord:Ordering[A])`	`SortOp(in:SilkSeq[A], ord:Ordering[A])`	`SilkSeq[A]`
`in.reduce(f:(A,A)=>A)`	`ReduceOp(in:SilkSeq[A], f:(A,A)=>A)`	`SilkSingle[A]`
`in.size`	`SizeOp(in:SilkSeq[A])`	`SilkSingle[Long]`
`left.join[B,K](right:SilkSeq[B], lk:A=>K, rk:B=>K)`	`JoinOp(left:SilkSeq[A], right:SilkSeq[B], lk:A=>K, rk:B=>K)`	`SilkSeq[(A, B)]`
`in.groupBy[K](k:A=>K)`	`GroupByOp(in:SilkSeq[A], k:A=>K)`	`SilkSeq[(K, SilkSeq[A])]`
`in.mapWith[B,R](resource:Silk[R])(f:A=>B)`	`MapWithOp(in:SilkSeq[A], resource:Silk[R], f:A=>B)`	`SilkSeq[B]`
`in.split`	`SplitOp(in:SilkSeq[A])`	`SilkSeq[SilkSeq[A]]`
`in.concat`	`ConcatOp(in:SilkSeq[SilkSeq[A]])`	`SilkSeq[A]`
`in.shuffle[K](f:A=>K)`	`ShuffleOp(in:SilkSeq[A], f:A=>K)`	`SilkSeq[(K, SilkSeq[A])]`
`c"(unix command)"`	`CmdOp(cmd:String, inputs:Array[Silk[_]])`	`SilkSingle[Any]`
`c"(unix command)".lines`	`CmdOutputLines(cmd:StringContext, inputs:Array[Silk[_]])`	`SilkSeq[String]`
`c"(unix command)".file(name)`	`CmdOutputFile(cmd:StringContext, file:String, inputs:Array[Silk[_]])`	`SilkSingle[File]`

表 1　Silk の演算オブジェクト（抜粋）。入力データ `in` に対して関数を呼び出した結果生成される演算オブジェクトとその型を示している。全ての演算オブジェクトにはその結果が代入される変数名や定義されているクラス名、行番号などの情報 (`fcontext`) が含まれているが表記上は省略している

造）を備えているため、データ処理分野への応用が進んでいる。Scala にバージョン 2.9 から標準で搭載されている並列コレクション [15] は、マルチコア環境での並列データ処理をスレッドプログラミングの知識無しに利用できる。分散環境への対応では、Spark の分散データセット (RDD) [19] が提案されておりコミュニティによる開発も盛んである。Twitter 社も Hadoop のジョブを簡易に記述するための Scala API である Scalding を公開している。

Scala のシンタックスは簡潔になりやすいため、同等のデータ処理コードを Java で書くのに比べてコード量が大幅に短くなる傾向がある。これはプログラミングの専門家以外にとってラーニングコストを下げる上で利点となる。また、関数型言語の特徴である高階関数 (関数に関数を渡す) の記述が自然にサポートされているため、MapReduce のように関数をリモートに配備し分散で実行させるスタイルによく合致している。

Silk が目標にするのは、Scala で記述されたパイプラインの拡張をしながら、障害の発生時には、既に計算が済んだ部分の結果は再利用しつつパイプラインの残りの処理を実行する仕組みである。これにはプログラムの途中結果を保存する仕組みと、プログラムの部分実行をサポートする必要がある。次節からこれらの機能をどう実現していくかについて解説する。

3.1　コードへのマーキング

プログラム中の変数名を使って途中結果を保存できれば、変数名を使ってプログラムの途中結果に対するクエリを実行することができ、さらに、変数名を指定して再実行したいプログラム中の箇所を選択できるようになる。ここでいかにプログラム中の変数名を取得するかが問題になる。変数名の情報は通常コンパイル時には失われてしまい、実行時のプログラムは知ることができない。

計算結果がどの変数に代入されたかの情報を得るために、コンパイル時に得られる情報を活用するアプローチに至った。2013 年に Scala に導入されたマクロ [1] の機能を用いると、関数呼び出しの記述をコンパイル時に書き換えることが可能になる。その際、関数呼び出し周辺の構文木（AST, Abstract Syntax Tree) を取得することができる。この機能を用いると、コード中に map, filter, join などの分散演算が現れた箇所周辺の AST を探索し、各々の演算結果がどの変数に代入されているかを見つけることができる。例えば、

```
val a = input.map(f)
```

という Silk の演算は、マクロでコンパイル時に以下のような演算オブジェクトに置き換えられる。

```
val a = MapOp(in:input, out:"a", op:f)
```

ここで `MapOp` は map を表す演算オブジェクト、`in` は入力データ、`out` は結果の出力先の変数名の文字列、`op` は `in` の各要素に対して適用される関数である。

実際のシステムでは、変数名だけでなく、変数が定義されている関数名、クラス名、ソースコード中の行番号などより詳細な情報が AST から取得されており、これをプログラム中のマーカーとして用いている。演算オブジェクトの実行結果を、このマーカー名を用いてキャッシュやディスクに保存することで、プログラム中の変数名を用いてワークフローを流れるデータの検索や参照が可能になる。

3.2　ネストした演算

前説の変数 `a` の結果から、10 より大きい要素を取り出す式を追加すると、以下のようになる。

```
val b = a.filter(_ > 10)
```

ここで変数 `a`、`b` に代入される演算オブジェクトは、

```
val a = MapOp(in:input, out:"a", op:f)
val b = FilterOp(in:a, out:"b", op:{_ > 10})
```

であり、実際にこの変数 `b` の中身を展開すると以下のようにネストした演算オブジェクトなっている。

```
FilterOp(
  in:MapOp(in:input, out:"a", op:f),
  out:"b",
  op:{_ > 10})
```

Silk の全ての演算は、このネストを許した演算オブジェクトで表される。**表 1** に Silk で定義される演算オブジェクトの一覧を示す。これは関係代数 (relational algebra) における操作

命令 (σ(selection), π(projection) など) に該当する。演算オブジェクトは `Silk[A]` という型を持つ。これは A というオブジェクト型を生成する演算オブジェクトであることを表し、単一のオブジェクトを生成する演算には `SilkSingle[A]` 型、リスト型を生成する演算には `SilkSeq[A]` 型が付けられている。Join 演算など入力が 2 つ以上の場合もあり一般には DAG(directed acyclic graph) の形状を取る。将来的に SQL と同等の演算オブジェクトを Silk に導入することも検討している。

3.3 UNIX コマンドの利用

Silk では UNIX コマンドのパイプラインを作成することもできる。以下の例は、カレントディレクトリ内の*.txt ファイルを ls コマンドで列挙し、各々に対して wc(word count) コマンドを実行し、単語数（のリスト）を計算する例である。

```
val inputFiles = c"ls -1 *.txt".lines
val wordCounts =
  for(file <- inputFiles)
    yield c"wc $file".lines.head
```

c で始まる文字列 `c "(UNIX command)"`は UNIX コマンドの演算オブジェクトを表し、この文字列内には変数や演算などの任意の式を`$expr` の形式で埋め込める。for 文は Scala では for-comprehension と呼ばれ、`inputFiles` の各要素 `file` に対し、yield 以下の式を実行した結果を連結したものを返す。for-comprehension は `map` 演算の syntax sugar であり、上記の for 文のコードは

```
inputFiles.map(file => c"wc $file".lines.head)
```

と書くのと等しい。この例の場合は、各ファイル毎の wc コマンドの実行結果の先頭行のリストになる。

UNIX コマンドの実行結果が標準出力に返される場合は `.lines` で各行のデータを取得できる。ファイルに結果を書き込む場合は、`.file(name)` でファイル名を明示的に返すことで、コマンド実行結果のファイルを受け取ることができる。

3.4 演算オブジェクトの拡張

Silk では関数を再利用できるため、上記の基本演算に加えて独自の実装を追加することもできるようになっている。例えば、Silk での hash-join の実装は、**表 1** の演算の組み合わせ (map, shuffle, reduce など) と、データがメモリに十分収まるサイズになったときにインメモリで join を実行する記述になっている。大規模データのソーティングでは、データから数%程度のサンプルを取得しヒストグラムを作成し、ノード間の偏りが無いようにデータを分散してソートしているが、このヒストグラムの作成も Silk のパイプラインで記述されている。

3.5 Weaving: 演算オブジェクトの評価

`Silk[A]` の演算オブジェクトは `weave` 関数を呼び出すと、スケジュールの評価が始まる。これは、素材 (silk) を編んで (weave) 織物 (product) を生成するのに例えている。Silk の設計では、Silk[A] の構築と、weaving の過程を明確に切り分けている。そうすることで、パイプラインの記述は変えずに、実際に使う計算機環境に合わせて実行方式だけを切り替えられるようにできる。これはパイプラインのテスト用と本番環境での実行の切り替えにも役立つ。

Silk には演算オブジェクトを評価する数種類の weaver が提供されている。メモリ中で計算を済ませる `InMemoryWeaver`, クラスタ環境で分散してデータ処理をする `ClusterWeaver`, GNU Make の用に結果を逐次ファイルに書き出す `MakeWeaver` がある。この他にも後から独自の weaver を追加することも可能なように設計している。

3.6 静的な最適化

`Silk[A]` の DAG スケジュールはパターンマッチによりグラフを組み替えることで、ルールベースの最適化を施すことができる。この静的な最適化の例としては、関数の合成ルール

```
map(f).map(g) => map(f andThen g)
```

や、map と filtering の合成により生成される中間オブジェクトの量を減らすルール

```
map(f).filter(c) => mapWithFilter(f, c)
```

などがある。他にも Silk 内で使われるストレージの種類により pushing-down selection、columnar ストレージへのアクセス、オブジェクトの部分的な projection などの適用が考えられる。

4. クラスタでの実行

クラスタで分散演算を実行するための ClusterWeaver の概要を**図 1** に示す。以降の節では、ClusterWeaver の各コンポーネントについて紹介し、どのように Silk が分散演算を実行しているかについて解説する。

4.1 ClassBox: リモートでのコード実行

Silk は Scala 上で動くコードであり、Java プログラムと同様 JVM 上で動く。リモートに立ち上げた SilkClient(JVM で動作）でローカル JVM と同じようにコードを実行するためには、まず classpath に含まれる全ての class ファイルと jar ファイルをリモートノードにも配備する必要がある。Hadoop や Spark では必要な jar をコード中に指定する必要があるが、Silk では、ローカル環境中の classpath と jar のエントリを列挙し ClassBox と呼ばれるオブジェクトを生成する。ClassBox には classpath エントリと jar ファイルのパスとその hash 値の情報が含まれている。ClassBox をリモートに転送（実際にはリモートの SilkClient が jar ファイルを pull する）することでローカルと同じ環境をリモートノードで再現できる。

UNIX コマンドを各ノードで同じように実行するには、VM の利用や、docker などの軽量 UNIX コンテナなどを利用するより汎用性の高い方法が考えられるが、ここでは簡単のため全ノードで同じ UNIX コマンドが使えることを想定している。

4.2 クロージャのクリーンアップ

MapOp の演算オブジェクトではリモートで関数 `f:A=>B`（A の型のデータを引数に受け取り B の型の結果を返す関数）を実行する必要がある。そのために関数 `f` の内部で参照されている自由変数 (free variable. 関数の引数で定義されていない変数) すなわち、関数の外で定義されている外部変数 (outer variable) のデータを提供する必要がある。以下の例では、変数 `N` が map に渡される関数内で定義されていないので外部変数となる。

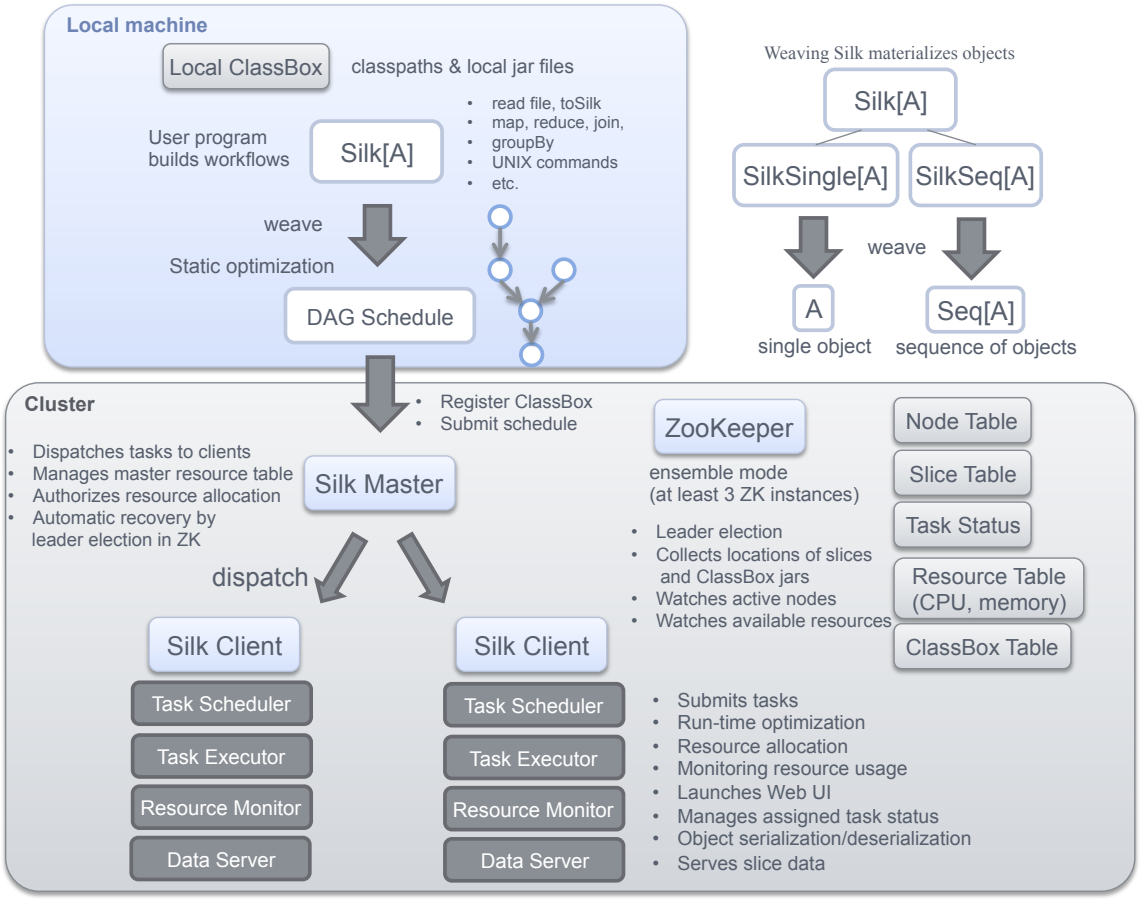

図 1　Silk の概要図. Silk では手元のローカルマシンで Silk を使った演算を記述する。Silk[A] のデータはそのままプログラムの DAG を構成している。ルールベースの最適化により DAG スケジュールを生成し、クラスタ中で動作している SilkMaster にスケジュールを送り Silk の結果を評価するタスクを実行する。Silk クラスタの維持に重要な情報はレプリカ構成で 3 ノード以上で動作する ZooKeeper に保存されており、SilkMaster が落ちた場合には他の SilkClient が SilkMaster に昇格する。SilkClient は個々のタスクの実行管理、スケジューリング、リソース確保の役割を担う。メモリやローカルディスク上のデータを配信するための DataServer も各 SilkClient で立ち上がっており HTTP ベースで各ノードとデータのやり取りができるようになっている。

```
val N = 100
val result = in.map(_ * N)  // N は外部変数
```

この N の情報と関数の情報同時に送らないとリモートで正しい計算が実行できない。この外部変数の環境と関数本体を合わせたものをクロージャーという。

例えば Hadoop では DistributedCache(バージョン 2.x 系列では JobContext) に外部変数の情報を登録しておき、全ノードからアクセスできるようにしているが、クロージャー内の外部変数の出現を把握できればこのプロセスは自動化できる。

Scala ではクロージャーは 1 つのクラスとして定義されているため、ClassBox が転送済みであれば、外部変数の情報を補うことでクロージャーをリモートでも実行できる。しかしそのクロージャー内で参照されているクラスのうち 1 つのフィールドしかアクセスされない場合、クロージャーのシリアライズ時にリモートに送るバイナリサイズが巨大になってしまう問題がある。

実際に関数中で使われる最小限の外部変数の情報を得るために、Silk では関数 `f` のバイトコードをスキャンしており、その操作のために ASM (Java の bytecode を操作するためのライブラリ) を用いている。関数の動作を JVM のスタックへの操作も含めてシミュレートすることで、関数 `f` からどのような関数が呼び出されるかを追跡でき（データフロー解析）、コンテキスト内で定義されていない変数にアクセスした際にそれを外部変数への参照として記録する。この操作で各関数（クラス名）と、その内部で使われる外部変数の対応表ができる。この対応表の作成は、関数 1 つにつき 1 回実行するだけで良いため重い計算ではない。この情報を元に、使われないフィールドを null 値でクリアすることで、Java の標準のシリアライザーを使って

クロージャーをシリアライズした場合でもバイナリサイズを必要最小限に納められるようになった。

また、Silk では大きな外部変数を必要があって使う場合は、外部変数として参照するではなく、`mapWithResource` 関数

```
in.mapWithResource(resource:R)(f:A=>B)
```

の利用を推奨している。`mapWithResource` では、外部変数の代わりに f の内部で使用できるリソース R の情報を渡すことで、リモートに送るデータサイズを小さくしており、リソースのあるノードに近い場所でなるべく計算を行うなどの工夫ができるようになっている。

4.3 クラスタのセットアップ

Silk でのクラスタのセットアップはできるだけ簡単になるように実装されている。ユーザーが最低限設定に必要なのは、$HOME/.silk/hosts ファイルにクラスタとして使うノードのホスト名を一行毎に列挙するだけで良い。`silk cluster start` コマンドを実行すると、`SilkClient` を指定されたホストに ssh ログインして立ち上げてくれる。hosts ファイルがない場合は単一ノードで動く standalone モードで Silk が動作する。

4.4 ZooKeeper

Silk ではノードの死活管理や分散データのインデキシング、タスクの実行状態の記録などの用途に ZooKeeper [7] を用いている。ZooKeeper はクラスター計算機のためのコーディネーションシステムで、ノード間の同期を取るためのプリミティブな操作（分散ロックなど）が提供されており、比較的小さなデータの共有データストレージとしての役割も果たす。Silk ではクラスタの起動時にノードのリストから 3 台を選び、ZooKeeper を ensemble 構成で自動的に起動するようにしている。これによりユーザーは ZooKeeper をセットするアップ手間を省けるようになるとともに、ノードの障害時に ZooKeeper 内のデータ喪失も防ぐこともできる。

各ノードで起動された `SilkClient` は、ZooKeeper の ephemeral mode（ノードへの接続が維持されている間だけデータが存在する）を用いて自身の情報（ホスト名、アドレス、ポート番号、プロセス ID など）を記録しており、クラスタ中でアクティブなノードのリストが ZooKeeper 上で管理されるようになっている。

ZooKeeper は leader election の機能も果たしており、`SilkClient` の中から 1 台を `SilkMaster` に選出するために使われている。`SilkMaster` が何らかの理由でクラッシュした場合は、`SilkClient` の中から新しい `SilkMaster` が選出され、ZooKeeper などに保存された情報から自動的に復帰するように設計されている。

4.5 SilkMaster

`SilkMaster` はタスクの `SilkClient` への配分 (dispatch)、リソーステーブルのマスターの保持などの役割を担う。`Silk[A]` を ClusterWeaver 上で実行する際にはまず `SilkMaster` に DAG スケジュールを送信することになる。`SilkMaster` が DAG スケジュールを受け取ると、アクティブな `SilkClient` の中から 1 台を選んでスケジューリングを委譲する分散スケジューリングのアプローチを取っている。マルチスケジュール最適化などを行う際には、単一のマスターがリソースの共有に基づいた最適化を行う方が都合が良いが、それと同時にマスターの負荷が重くなりすぎインタラクティブな解析に弱くなる欠点がある。Silk ではより細粒度のタスクが多いことを想定し分散スケジューラーのモデルを採用している。

4.6 SilkClient

`SilkClient` は各ノードでデーモンして立ち上げられており、タスクのスケジューリング、実行、ノードの CPU 使用率、メモリ使用量の管理を行っている。

4.7 Data Server

Silk では `SilkSeq[A]` 型の大きなデータを Slice という単位に分割し、各ノードにばらまくことで分散計算を実現している。その Slice のデータを保持するのが `DataServer` である。`DataServer` は各 `SilkClient` 毎に配備され、Slice がどの `DataServer` に保存されているかの情報は ZooKeeper に記録されている。インメモリ計算の場合は DataServer にはオブジェクトデータがそのまま保存されており、ローカルノードからのリクエスト時にはそのままオブジェクトの参照を渡すが、リモートからデータがリクエストされた場合には、オブジェクトをシリアライズして転送する。

4.8 Resource Monitor

Silk では、クラスタ中のあらゆるジョブが単一スケジューラーで管理されることを想定しておらず、他のジョブ実行エンジンと共存することを前提にしている。これは、共用クラスタでは、他のジョブがノードの CPU やメモリを埋め尽くしている場合もあり、この状態を把握するために、単一スケジューラーで全てを動かす（例えば Mesos [5] や、YARN [17] のアプローチ）に移行するよりは、各ノードでリソースモニターを動かし、実際に使用されている CPU の load average や空きメモリ量を定期的に ZooKeeper に記録し、その情報を活用するようにしている。

4.9 Task Scheduler

タスクのスケジューリングは各 `SilkClient` が独立に行っている。これは Google の Omega スケジューラー [16] に倣った設計で、リソースの使用状況のテーブルを、クラスタ中の全ノードが共有できる前提になっている。

リソースの使い方の判断は、Spark で使用されている *delay scheduling* [18] と同様に、データの locality を考慮して、計算に必要なデータを持っているノードを優先的に使用する。もし何回かの試行（閾値とタイムアウトを設定）でノードが利用可能にならなければ、他の利用可能なノードにデータを転送して計算を実行される割り当てを採用する。

リソースの使い方を決めた後は SilkMaster に問い合わせをし承認を得る。実際のリソースの空き状況と大きくかけ離れていなければ許可を与える設計になっており、楽観的な判断を行っている。これは CPU を 1 つ使うと宣言したタスクで合っても、CPU を必ずしも 100%使用するわけではないことへの対応である。厳密なリソース割り当てを行うと、12 コアのノードでは高々12 個のタスクしか行えないが、現実にはその 1.5 倍程

度のタスクを与えてもうまく処理できることが多い。タスクの割り当て過ぎへの対処には、各ノードから集計された実際のリソース使用量の情報をもとにつじつまを合わせ、タスクの配分の許可を調整するようにしている。

5. 今後の展望と課題

Silk の開発は現在も進んでおり、今後、現場での活用、大規模な性能評価を実施することで性能・機能の向上につながることを期待している。ここでは Silk の活用法について議論する。

5.1 データ解析のサイクルの効率化

「ワークフローのプログラミング」-「分散計算の実行」-「結果の確認」-「コードの修正・拡張」-「再計算」、このサイクルの効率を向上するためには全ての処理をパイプライン化する必要がある。まずデータ処理のための数行のコードを Silk で記述してクラスタ環境にコードをデプロイし、実行をスタートさせる。結果を確認するためのコードを書くのに一番手軽な方法は、ソースコード中にコードを追加することであろう。Silk では変数名を参照して変数の内容を確認するなど任意の操作を追加できるため、データが出力されたファイル名を指定するなどの手間がない。俗に printf デバッグと呼ばれる方式であるが、Java 界隈でも slf4j など文字列を出力する API しかない logger が広く使われている。breakpoint などを使用するデバッガと違いプログラムの動作を手動で止める必要がないという利便さもある。

5.2 ワークフロークエリ

実際にパイプラインで解析中の途中結果に対してクエリを投げることで、どのようなデータが生成されているかをモニタリングすることも可能になる。データの様子を確認することで、次にどのような解析をすれば良いかのアイデアも生まれるようになり、次の解析のためのコードがパイプラインに追加されるようになる。Silk では、変数名を使った中間結果へのマーキングを行うことで、計算済みの結果へのアクセス方法を提供しており、ワークフローに対して変数名を使ったクエリを記述する方向性も期待できる。

5.3 Makefile の代替としての利用

このようなデータ解析のワークフローを実現するのに同等のものとして Makefile が挙げられる。Makefile では UNIX コマンドのパイプラインの記述ができ、通常ソースコードのコンパイル方法の指示などに使われる。Makefile の成果物はファイルであり、必要なファイルが存在しなければそのファイルを生成するためのルールを Makefile 中の記述から見つけ出し、生成に必要な UNIX コマンドを実行する仕組みである。

Makefile はデータサイエンスに置けるパイプラインの記述にも応用することもできるが、Makefile におけるタスクの実行単位は UNIX コマンド毎にタスク間の並列・分散実行 (例えば Sun grid engine や GXP などを用いる) は可能であるが、タスク内並列化はできない。また入出力のデータがファイルになるため、インメモリコンピューティングによる計算の高速化に応用することが難しい。

Makefile を使う利点はタスク間の依存関係を記述できることである。この利点のみを活かし、タスク間 (inter-task)、タスク内 (intra-task) 並列・分散実行を行えるような枠組みを Silk では採用している。

以下に Makefile と Silk の機能の対比を示す。

機能	Makefile	Silk
タスク間の依存関係	ファイルの引数	Silk 型の変数の参照
タスクの実行	コマンドの実行	Silk 型の変数の評価
タスクの細分化	ファイルの分割	SilkSeq の分割

表 2 Makefile と Silk の機能の対比

5.4 Iterative computing への対応

PageRank, K-means の計算などはループによる処理を含み、iterative computing と呼ばれる。現在のところ Silk ではループ毎に weave 処理を繰り返すことで iterative computing を実現しているが、itertive computing の代数的枠組みへの導入も鬼塚らにより提案されており [14]、Silk への導入も検討している。

5.5 LArray: Off-Heap ストレージ

分散インメモリ計算ではメモリストレージが JVM の OutOfMemoryException の発生や、GC 処理の負荷のために応答が停止してしまうという問題がある。この問題を解決するには、JVM が管理するヒープ外の領域 (off-heap) でデータを確保し、データが不要になったら即座にメモリから解放できる仕組みが必要となる。Java の標準にはこのようなメモリ管理を実現できる仕組みがないため、Off-heap メモリを扱うライブラリ LArray (https://github.com/xerial/larray) を開発しリリースしている。

5.6 圧縮オブジェクトストレージ

Silk の分散演算では大量のオブジェクトが生成できるようになる。このオブジェクトデータをスライス毎に効率よく圧縮し、ディスクなどに保存して永続化するストレージの実装も進めている。オブジェクトを列分解し、gzip, snappy などで圧縮することでデータサイズを小さくすることができるが、列分解するコストや、オブジェクトへの復元の高速化が課題となっている。

6. 関 連 研 究

巨大なデータの処理には、データをクラスタ内に分散配置して並列処理する MapReduce やその実装である Hadoop が広く用いられるようになった。一般のデータ処理では単一の MapReduce だけでは十分ではなく、データ処理のパイプラインを作成するために、Pig [13]、Oozie、Twitter Summingbird などのオープンソースプロジェクトが登場している。SQL 言語を Hadoop 上で実現するために Hive [6], 分散ストレージへのアクセスをプラグインにし SQL の演算部分のみを切り出した分散クエリ処理エンジンである Facebook の Presto などのプロジェクトも登場している。

Pig [13] では複数の MapReduce 処理を組み合わせたパイプラインを独自のスクリプト言語を用いて記述する。FlumeJava [2] では Java 言語で MapReduce のパイプラインを手軽に記述するためのプリミティブ演算を提供するライブラリであるが、Scala を用いるほどの簡潔さはなく、Hadoop と密結合された実装で

あるため実行エンジンの切り替えが難しい。DryadLINQ [4] はプログラミング言語中で SQL ライクな構文を使えるようにすることで、計算とデータベースへのアクセスを融合させている。

Nova [12] は Hadoop のジョブをパイプライン化したものに対し、入力データに更新が起った場合、再計算が必要な部分のみの差分を逐次更新するためのフレームワークである。Microsoft の Niad ではこれを differential workflow [11] と定義し、同等の問題を扱っている。

分散計算の iterative computing への対応では、Spark [19] が各ノードのメモリ中データをキャッシュしておくことで処理の効率を挙げている。Spark は Silk と似ている部分があるが、プログラム全体の DAG スケジュールを構築するのではなく、map などの関数ごとにスケジュールを構築する逐次評価のアプローチが取られている。このため、プログラム全体の最適化、実行方式の切り替えを行うにはプロジェクトの大規模な書き換えが必要になるため、Silk では独自の実装を用いる判断に至った。

低レイテンシでのクエリの実行には Google の Dremel [10] や、Muppet [9]、Cloudera の Impala などがある。Dremel はカラム指向ストレージを活用し、アクセスするデータ量の削減、データの集約演算の高速化を実現している。

分散環境でのジョブスケジューラーには、データの局所性を考慮するものが提案されている。Dryad の Quincy [8] ではグラフを用いた最適化が行われ、Spark では delay scheduling [18] が用いられている。これらのアプローチは HDFS など shared-nothing 型の構成でデータを転送するネットワークのコストが重い場合に重要であるが、近年価格が下がってきた Infiniband では 10Gbps を超える速度でデータを転送できる。このようなネットワークが普及してくると、ボトルネックはデータの局所性ではなく、オブジェクトのシリアライズなどによる CPU に推移すると考えられ、スケジューリングアルゴリズムも再考が必要になるだろう。

クラスタで多数のサービスを同時に動かしている場合、そのリソース管理には Mesos [5] や Apache の YARN [17] などのリソースマネージャーの活用が進んでいる。Mesos では単一のマスターが各ノードに対してリソースを提供 (offer) する方式を取っており、Hadoop や Spark [19] などのフレームワーク側が、それらのリソースをどのように使用するかを決定できる。ただし、これらのリソースマネージャーではクラスタ全体のリソース状況をフレームワーク側が知ることができないため、Google では Omega [16] スケジューラーを開発し、リソーステーブルを全ノードが共有するアプローチを採用し、サービス型、タスク型とリソースの利用時間が異なるジョブを共存できるようにしている。

7. おわりに

Silk は分散データ処理を書きやすくし、パイプラインの最適化・拡張・部分計算を実現するためにデザインされたシステムである。Silk のターゲットは、データを使ったサイエンス・分析を行う人すべてであり、ゲノムサイエンスなど応用できる場面は多岐に渡る。分散計算への敷居や学習コストを下げ、なおかつ、運用の手間を減らし、多様なマシン構成に対応できるようにするのはチャレンジングであるが、取り組む価値のある問題である。

文　献

[1] E. Burmako and M. Odersky. Scala Macros, a Technical Report. *EPFL*, 2012.

[2] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. *ACM SIGPLAN Notices*, 45(6):363–375, 2010.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[4] Y. Fetterly, M. Budiu, Ú. Erlingsson, and e. al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *Proc. LSDS-IR*, 2009.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *proc. of NSDI*, pages 22–22, 2011.

[6] Apache Hive. http://hive.apache.org.

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. *USENIX ATC*, 10, 2010.

[8] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *proc. of SOSP '09*, Oct. 2009.

[9] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style processing of fast data. *proc. of VLDB*, 5(12):1814–1825, Aug. 2012.

[10] S. Melnik, A. Gubarev, and e. al. Dremel: Interactive Analysis of Web-Scale Datasets. *In proc. of VLDB*, 2010.

[11] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: recursive, delta-based data-centric computation. In *proc. of VLDB*, July 2012.

[12] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous Pig/Hadoop workflows. In *SIGMOD '11*, June 2011.

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *proc. of SIGMOD*, pages 1099–1110, 2008.

[14] M. Onizuka, H. Kato, S. Hidaka, K. Nakano, and Z. Hu. Optimization for iterative queries on mapreduce. *PVLDB*, 7(4), 2013.

[15] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Euro-Par*. 2011.

[16] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *proc. of EuroSys*, pages 351–364, Prague, Czech Republic, 2013.

[17] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: Yet another resource negotiator. In *proc. of SOCC '13*, 2013.

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*, Apr. 2010.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, Apr. 2012.