第9回 AICS公開ソフト講習会
K MapReduce

丸山 直也
松田 元彦
滝澤 真一朗

理化学研究所 計算科学研究機構
プログラム構成モデル研究チーム

Agenda

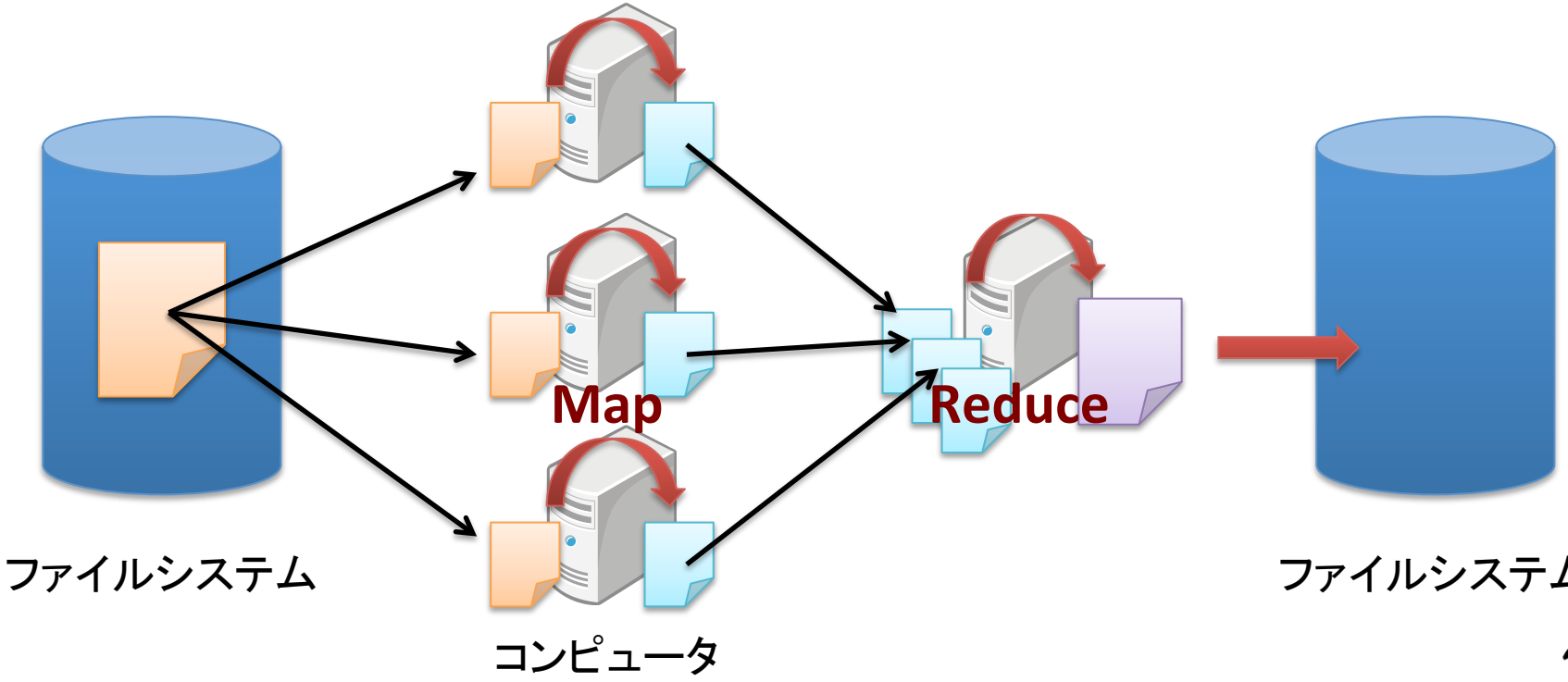
- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- KMR利用事例
  - ゲノム解析
  - レプリカ交換分子動力学法
- まとめ

MapReduceとは

- 多数のコンピュータ上で、大規模なデータを分散処理することを目的に導入されたプログラミングモデル
  - 2004年にGoogleが発表した論文を期に広まる
    - Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04.
- 学術利用だけでなく、企業利用事例も多数ある
  - ログ(アクセスログ、購入記録等)解析
  - ソーシャルグラフ解析
  - ゲノム、顕微鏡等の観測データ解析

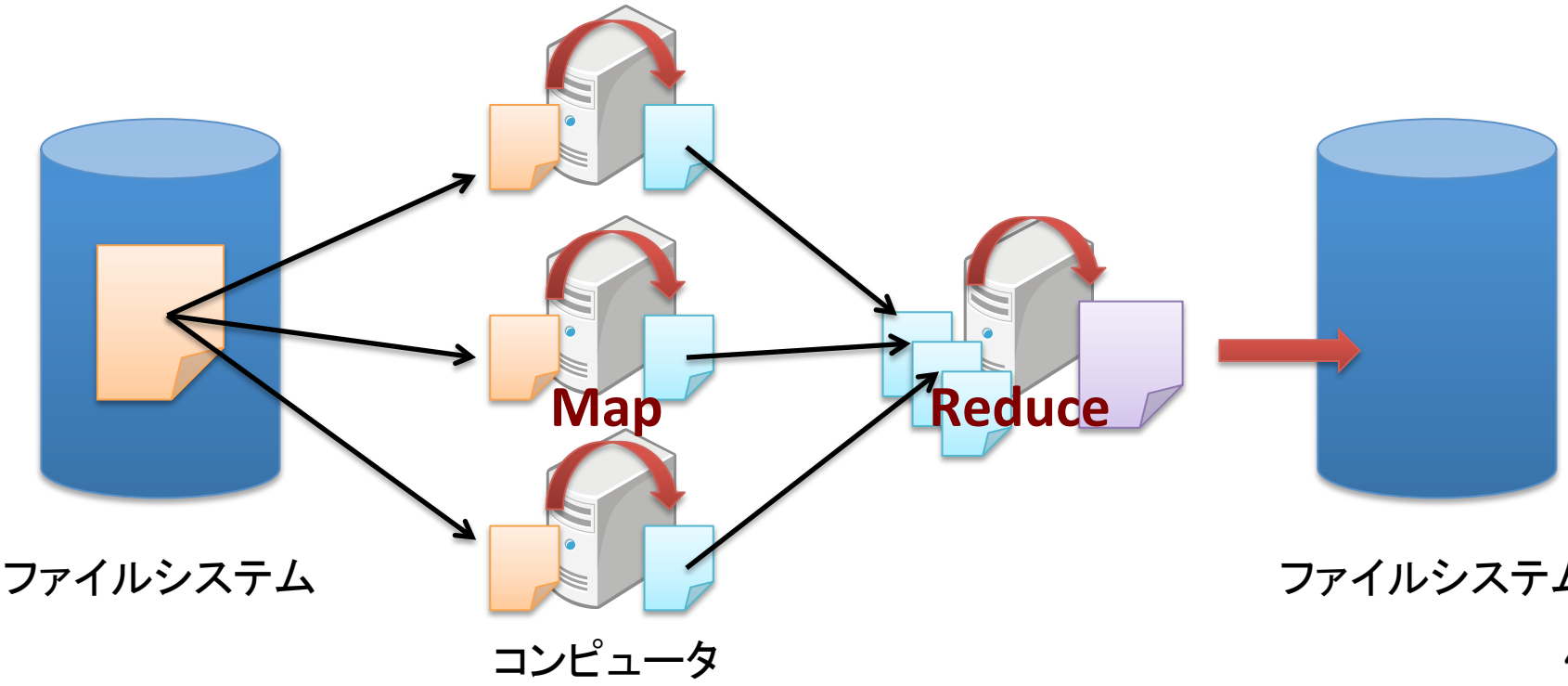
MapReduce動作概要 (1/2)

- データを分割して、多数のコンピュータで並列処理し、結果をまとめる
  - 個々の処理は「互いに独立」が前提



MapReduce動作概要 (2/2)

- データの分割・コンピュータへの分散、並列実行はMapReduce処理系が対応
  - 利用者はMap、Reduceで行う処理を逐次処理として実装するだけで良い => 並列化は処理系が担当



MapReduceの利点

- Map用プログラム、Reduce用プログラムの2つの逐次処理プログラムを実装するだけで良い
  - データの分割、コンピュータへの分散、Map/Reduce処理の並列実行はMapReduce処理系が担当

➡計算ロジックだけに集中してプログラム実装できる

- ただ、従わなければならない制約もあります

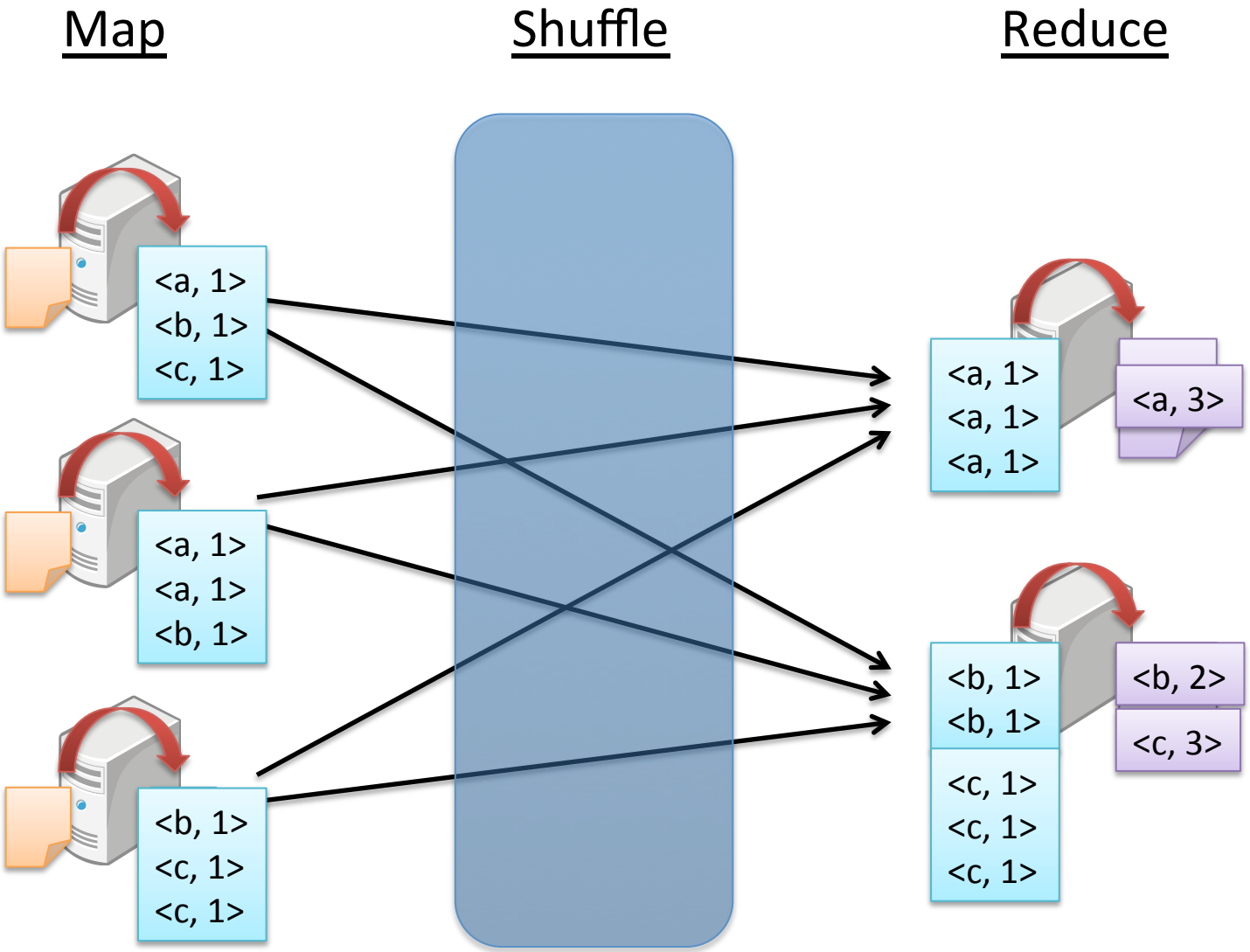
プログラム実装上の制約

- Map用プログラム
  - 異なるデータに対して、同じ処理が行われる
    - 入力毎に異なる計算をしたければ、入力のコンテキストに応じて処理を分岐しなければならない
  - Key-Value (KV)を出力する処理として実装
- Reduce用プログラム
  - 異なるデータに対して、同じ処理が行われる
  - KVを入力・出力とする
    - Map用プログラムが出力したKVがKey毎にまとめられて入力される
    - 同じKeyを持つ複数のValueを結合する処理として実装

Key-Value
ハッシュのような
キーと値のペア

<a, 1>, <b, 2>
<a, 3> など

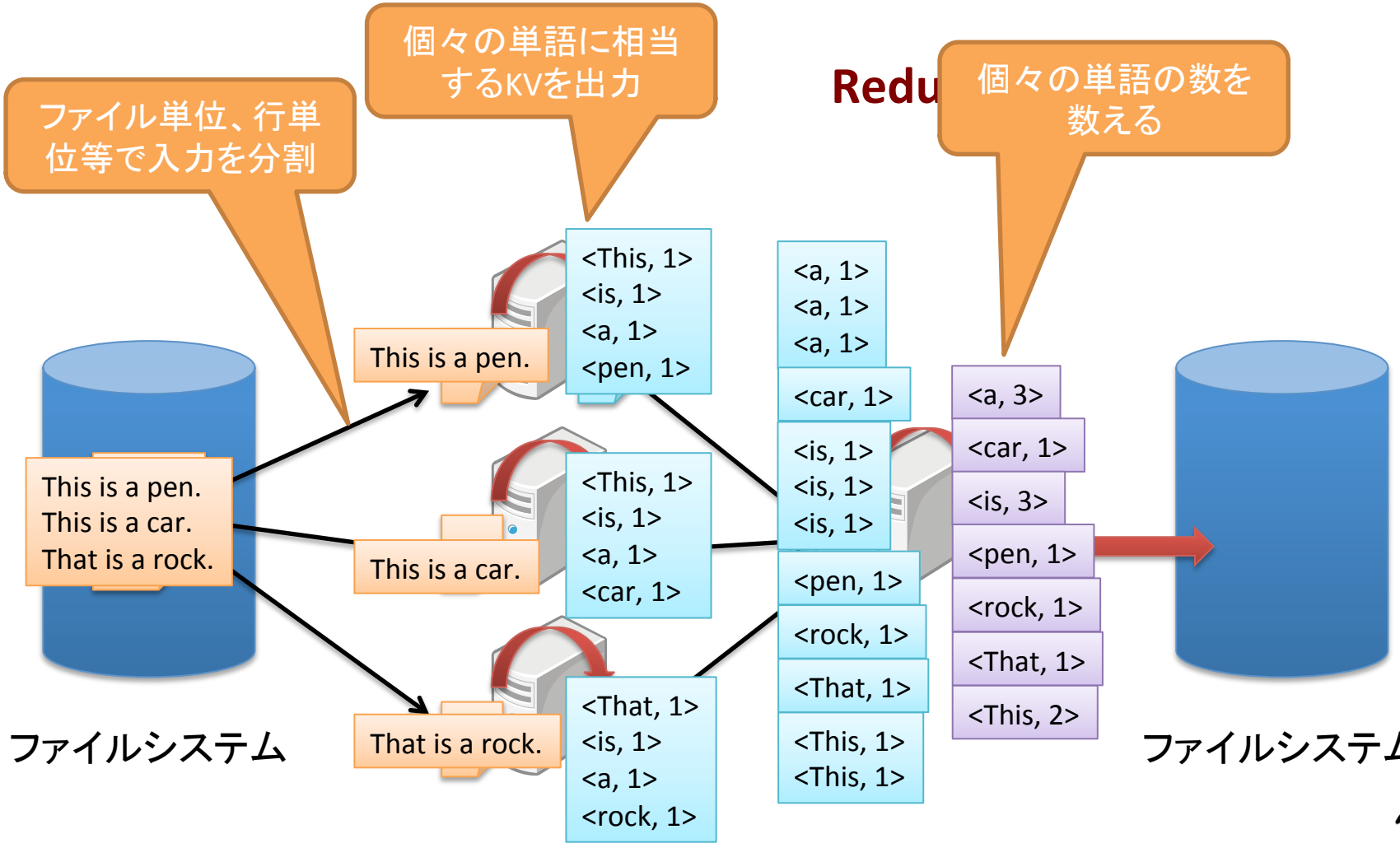
MapReduce動作詳細



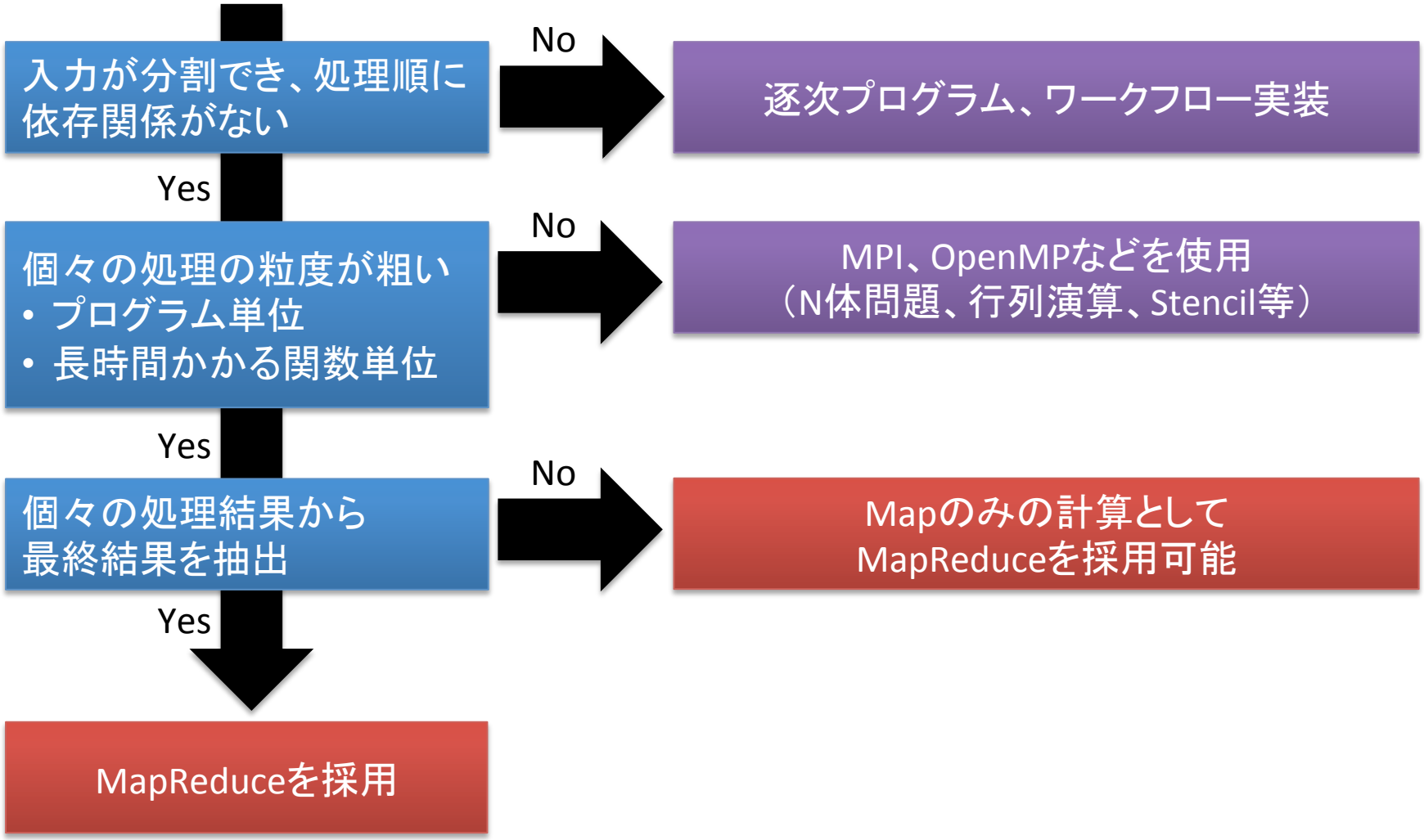
K computer

MapReduceプログラム例：Word Count

- テキストファイル中に出現する単語の数を数える



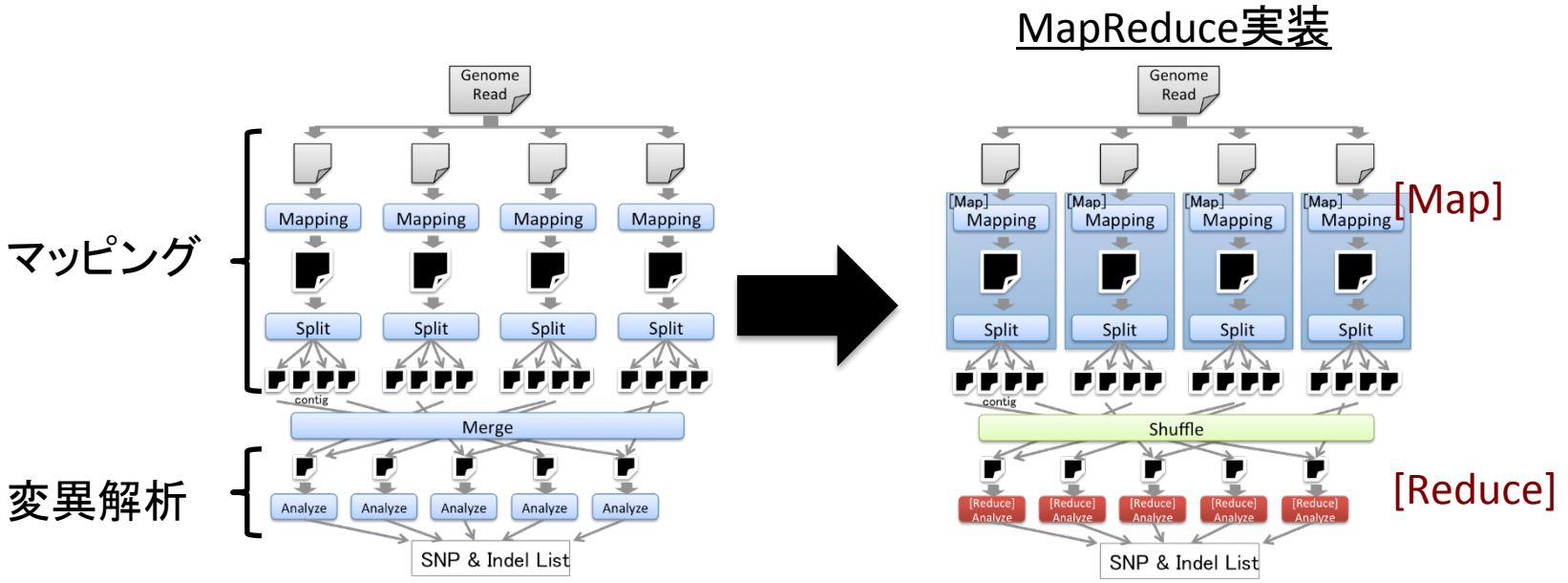
MapReduceモデルが有用な計算パターン



計算科学アプリケーションへの適用事例 (1/4)

- ゲノム解析ワークフローをMapReduceモデルで構築
  - 大規模データ処理を並列実行
  - シークエンサー出力のゲノムデータを分割してマッピング処理、塩基配列パターン毎に変異解析

➡ **典型的なMapReduceパターンで表現可能**



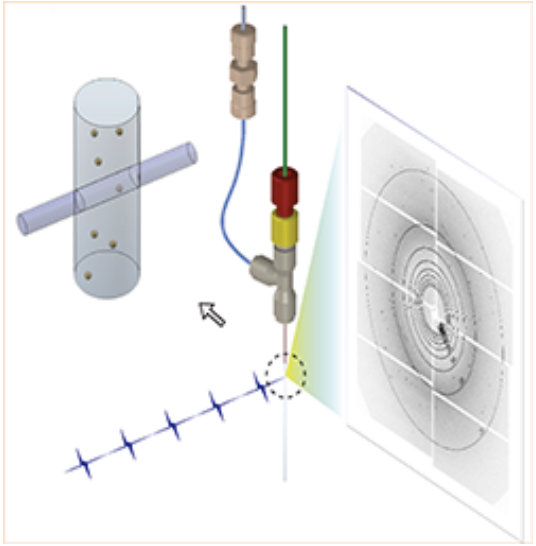
＊詳細は後述

K computer

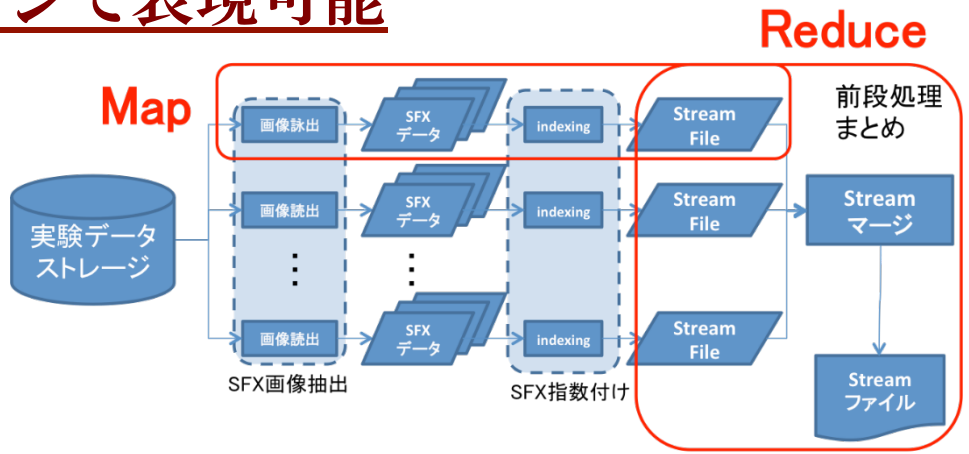
計算科学アプリケーションへの適用事例 (2/4)

- X線自由電子レーザー施設SACLAの利用実験データ解析
  - Serial Femtosecond Crystallography (SFX)実験では、タンパク質微小結晶からの回折パターンを大量に取得し、立体構造を解析
    - 100万枚/日規模の回折パターンを解析
  - SFX用の既存ツールを用いて、個々の回折パターンから特徴量を抽出し、結果を結合

➡ **典型的なMapReduceパターンで表現可能**



SACLA利用によるSFX実験の模式図

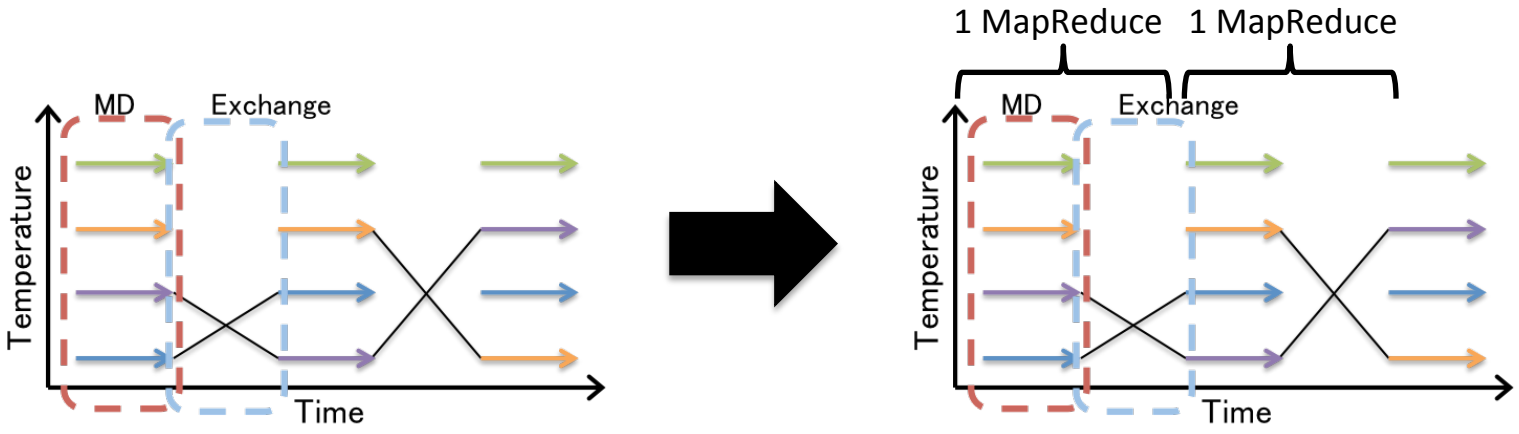


回折パターン1枚1枚にSFXツールを実行し、インデクシングを行う(Map)。
解析結果をストリームファイルとして結合する(Reduce)。

計算科学アプリケーションへの適用事例 (3/4)

- アンサンブル計算をMapReduceモデルで構築
  - 異なるパラメータのシミュレーションを複数行い、結果を統計処理
  - 例)レプリカ交換分子動力学法
    - エネルギーの異なる複数のタンパク質のレプリカについてMD計算
    - MD計算の結果を基に実行パラメータを交換し、MDを繰り返し実行

➡ **繰り返し処理を行うMapReduceパターンで表現可能**

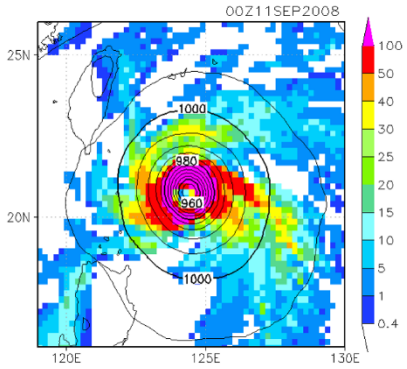


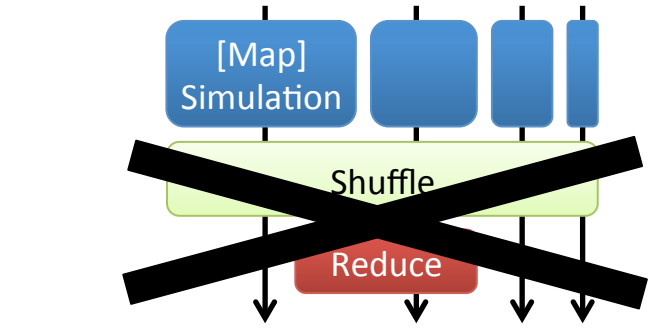
＊詳細は後述

計算科学アプリケーションへの適用事例 (4/4)

- 異なるパラメータで多数のシミュレーションを要求する計算科学アプリケーション
  - データ同化を用いた気象予報：10～10,000ケース
  - 自動車交通流等の社会シミュレーション：10～10,000ケース
- 京コンピュータでは同時受付ジョブ数に制限（15ジョブ）
  - 制限を超えたジョブを投入するためには、利用者がジョブ実行を監視・管理する必要がある

➡ Map実行のみのMapReduceとして、複数ケースを1ジョブ実行





Shuffle, Reduceの無い、MapReduceとして実行

K computer

MapReduceを京で効率よく実行するための要件

- マルチノード・マルチコア環境での高いスケーラビリティ
  - 82,944ノード・663,552コアの有効活用
  - 大量のKVの高速な通信
- 大容量ファイルの効率的な処理
  - 並列ファイルシステムに保存されているファイルへの、多数のノードからの同時アクセスに対応
- プログラムをまとめて実行する機能
  - 京ではジョブ投入数に制限

KMRはこれらを満たすように設計・実装されています

HadoopやMR-MPIなどの既存のMapReduce処理系は、
性能面、スケーラビリティにおいて不十分

K computer

Agenda

- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- KMR利用事例
  - ゲノム解析
  - レプリカ交換分子動力学法
- まとめ

K MapReduce (KMR)

- 大規模並列システム上で、スケーラブルに動作するMapReduce処理系実現を目的に開発された処理系

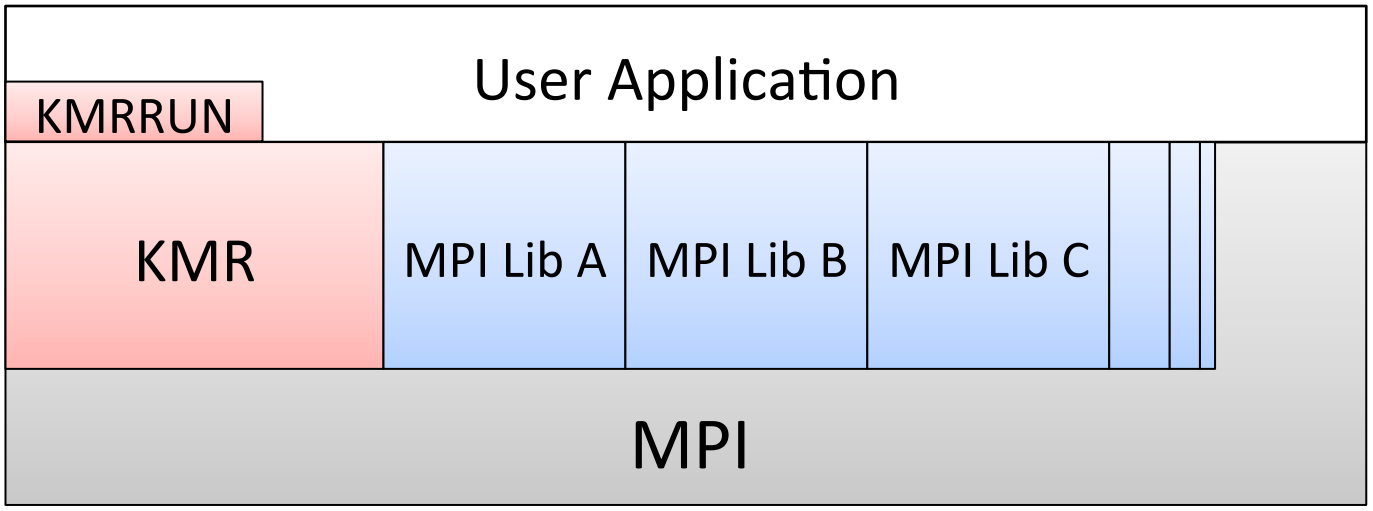
- 開発チーム：AICS プログラム構成モデル研究チーム
  - 丸山直也（プロジェクトリーダー）
  - 松田元彦
  - 滝澤真一朗

- 動作環境
  - 京コンピュータ、富士通FX10
  - Linux/Solaris + gcc/Intel Compiler + OpenMPI
    - コンパイラはc99準拠のこと
    - MPI以外の依存ライブラリはなし

KMR情報源

- プロジェクトホームページ
  - http://mt.aics.riken.jp/kmr/
- 論文
  - Motohiko Matsuda, Naoya Maruyama and Shinichiro Takizawa: K MapReduce: A Scalable Tool for Data-Processing and Search/Ensemble Applications on Large-Scale Supercomputers, IEEE Cluster 2013 Conference (2013).
    - KMRの実装と性能評価
  - 滝澤真一朗, 松田元彦, 丸山直也: MapReduceによる計算科学アプリケーションのワークフロー実行支援, 情報処理学会 HPCS2014 (2014).
    - KMRによる計算科学アプリケーションの実装と評価

KMRの実装位置づけ



- MPIの1ライブラリとしてC言語にて実装
- MPIや他のMPIライブラリと組み合わせたプログラムの実装が可能
- コマンドとしての利用も可能(KMRRUN)

KMRの特徴

- ノード間・ノード内にて並列実行するハイブリッド並列
  - ノード内： OpenMP
  - ノード間： MPI
- オンメモリ処理
  - ファイルを介さないので高速、一方で耐故障性には劣る
- 京コンピュータのネットワーク・ストレージ構成を意識した性能最適化
  - KVデータサイズに応じたShuffle通信手法の切り替え
  - 通信とIOを組み合わせた、高速な集団ファイル読み込み
  - ファイルの位置を考慮した、最適タスク配置
- 対応言語
  - ライブラリ使用時：C/C++, Fortran
  - コマンド使用時：ノードがサポートする任意の言語

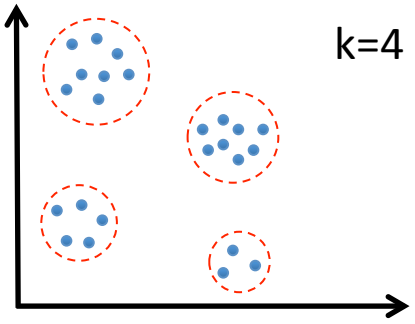
KMRのメリット

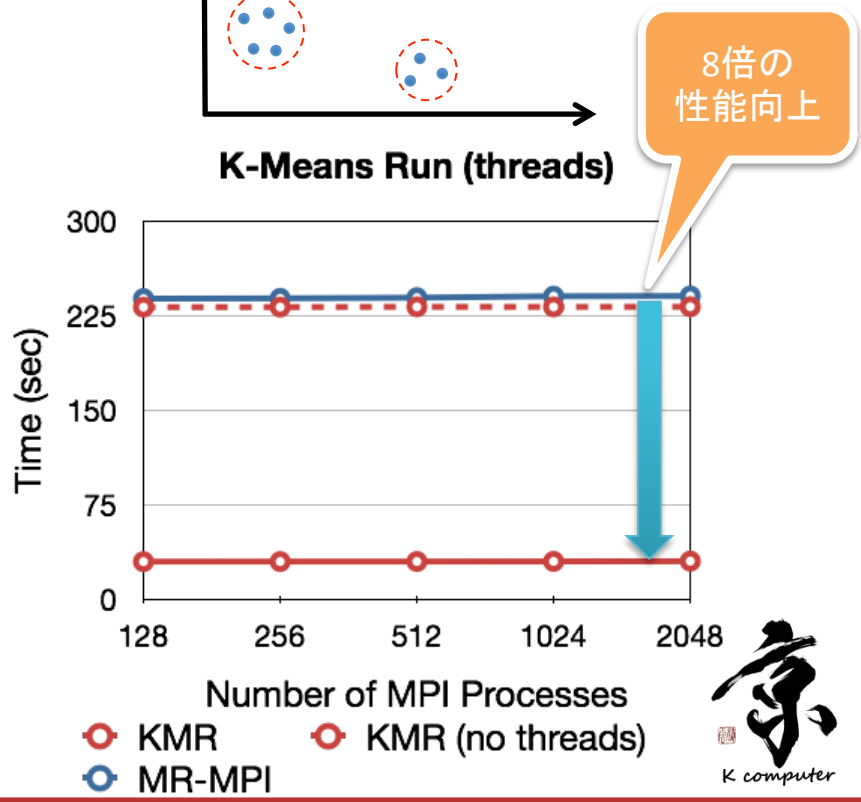
- MapReduceモデルを用いた、大規模並列システムでの容易な並列プログラミングモデル
  - 逐次処理のみの実装で、大規模並列実行可能
- 並列プログラミングを簡易化
  - Master-Workerタイプ計算の実行
  - パラメータ設定の簡易な集団通信
- 京コンピュータ利用時の複雑さを隠蔽
  - ネットワークアーキテクチャを考慮し、メッセージサイズに応じて集団通信アルゴリズムを自動切替
  - ストレージアーキテクチャを考慮し、IOノードへの負荷を削減するファイル読み込みを実施
  - MPI_Comm_spawnの動的プロセス終了の待ち時間調整

KMRのメリット – 計算

- マルチノード・マルチコアを用いた自動並列
  - KMRではKVを自動的にコア単位で複数ノードに割り振り、Map/Reduce処理を並列実行

- 例）k-meansクラスタリング
  - データをk個のグループに分割
  - MR-MPIでは、ノード間並列は行うが、コア間並列は行わない
    - MR-MPI：MPI実装されたMapReduce



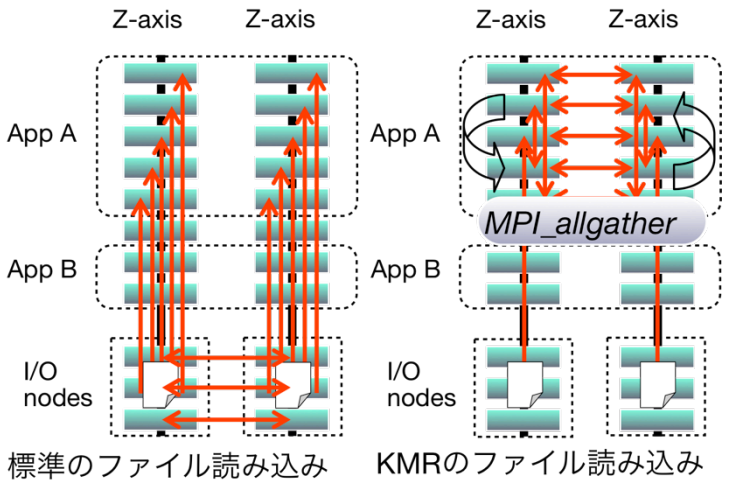


実行環境
- 京コンピュータ (8CPU/Node)
- 1 MPI Proc/Node

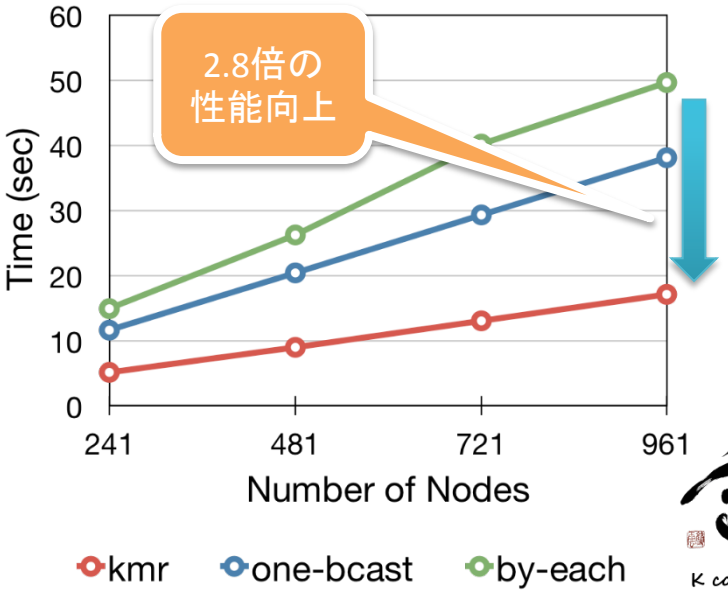
実行設定
- Points: 100,000/Proc
- Clusters: 10,000
- 4次元座標
- 繰り返し回数：10

KMRのメリット – ファイルアクセス

- IOアクセス競合を避け、ファイル読み込み性能を向上
  - 計算ノードを、グループ単位でIO数を制限することによりIOノードへのアクセスを削減
  - 計算ノード間で集団通信によるファイルの共有



- 1GBゲノムDBファイル読み込み
  - 京の各ノードから、一斉にゲノムファイルを読み込む
  - by-each：すべてのノードが個別に読み込む
  - one-bcast：1ノードが読み込み、他ノードにbcast転送



Agenda

- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- KMR利用事例
  - ゲノム解析
  - レプリカ交換分子動力学法
- まとめ

利用方法

- KMRRUNによる簡易MapReduce実行
  - 一連のMapReduce計算を実行
  - 計算ノードがサポートする任意の言語で実装された逐次プログラムを実行可能

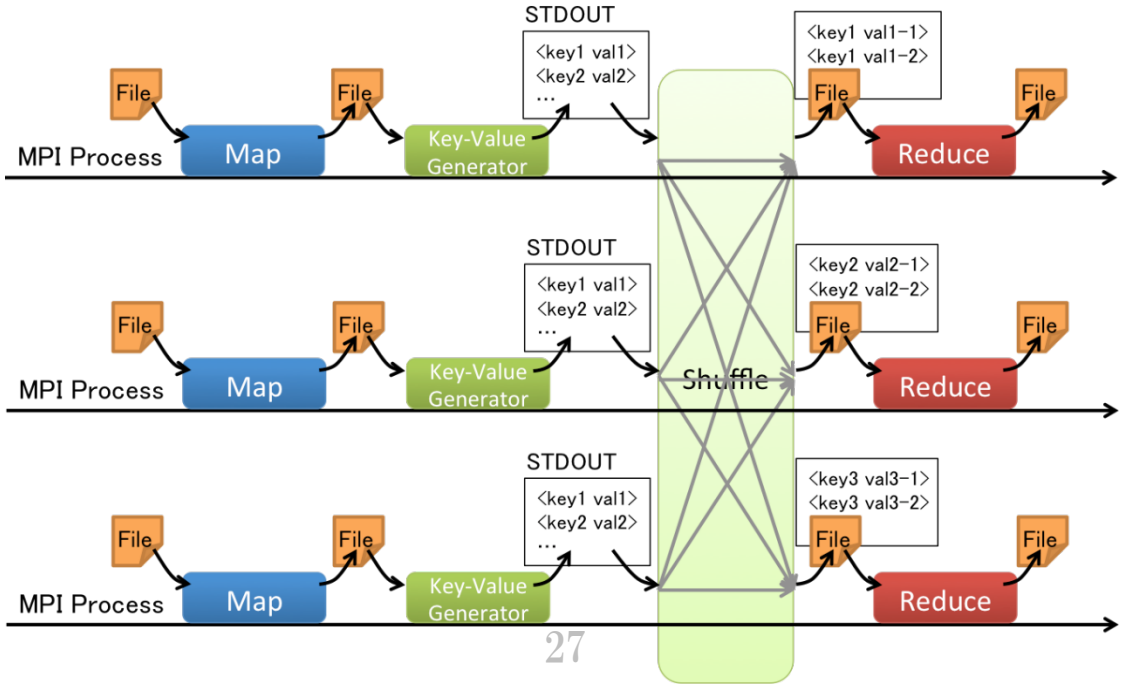
- KMRライブラリを用いたプログラミング
  - MapReduceモデルに従った任意のワークフローを実装可能
  - KMRの全機能を利用可能
  - C/C++, Fortranにてプログラムを実装

Agenda

- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- KMR利用事例
  - ゲノム解析
  - レプリカ交換分子動力学法
- まとめ

KMRRUN

- MapReduceワークフローを実行
- Mapper/Reducerとして、MPIプログラム、任意の言語で実装された逐次プログラム(ノード内並列対応)を実行可能
  - Mapperの出力からKVを生成し、標準出力に書き出すKey-Value Generator プログラムも必要



K computer

KMRRUNコマンド

- 実行コマンド

```
$ mpirun MPIOPT ./kmrrun -n procs -m mapper ¥
-k kvgen -r reducer ./input
```

- コマンドの意味

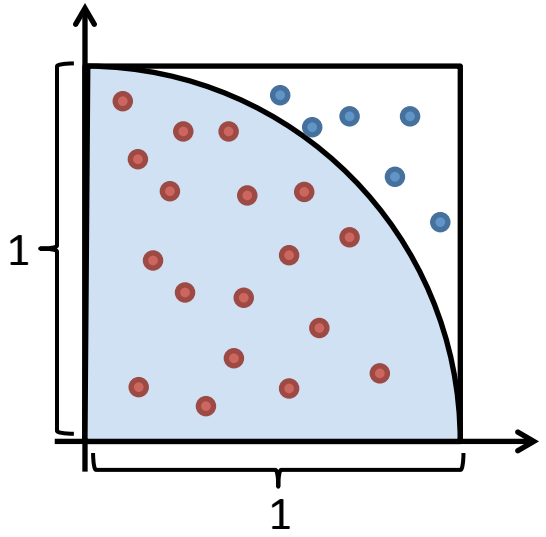
kmrrun	KMRRUNプログラム本体。 KMR_INST/lib/kmrrunにインストールされている。
-n procs	1回のMapper/Reducer実行で使用するプロセス数を指定。 「m_procs:r_procs」フォーマットで指定すれば、Mapper/Reducerで異なるプロセス数で実行可能。デフォルトは1。 [省略可能]
-m mapper	Mapperプログラム
-k kvgen	KV Generatorプログラム [省略可能]
-r reducer	Reducerプログラム [省略可能]
./input	Mapperの入力ファイル、またはディレクトリ。 全MPIプロセスからアクセスできる、共有ディレクトリ上におくこと。

Mapper/Reducer/KV Generator仕様

	Mapper	KV Generator	Reducer
実装言語	任意	任意	任意
並列実行	MPI/OpenMP	OpenMP	MPI/OpenMP
入力	• ファイル読み込み • ファイル名は最後の引数として渡される	• ファイル読み込み • Mapperの入力ファイル名が最後の引数として渡される • Mapperの出力を読み込む場合は、ファイル名を推測して作成	• ファイル読み込み • ファイル名は最後の引数として渡される • 1 KV/行フォーマット • KeyとValueはスペース1つで区切られる
出力	• ファイル書き出し • ファイル名は、入力ファイル名から類推できる名前とする	• 標準出力に出力 • 1 KV/行フォーマット • KeyとValueはスペース1つで区切られる	• ファイル書き出し

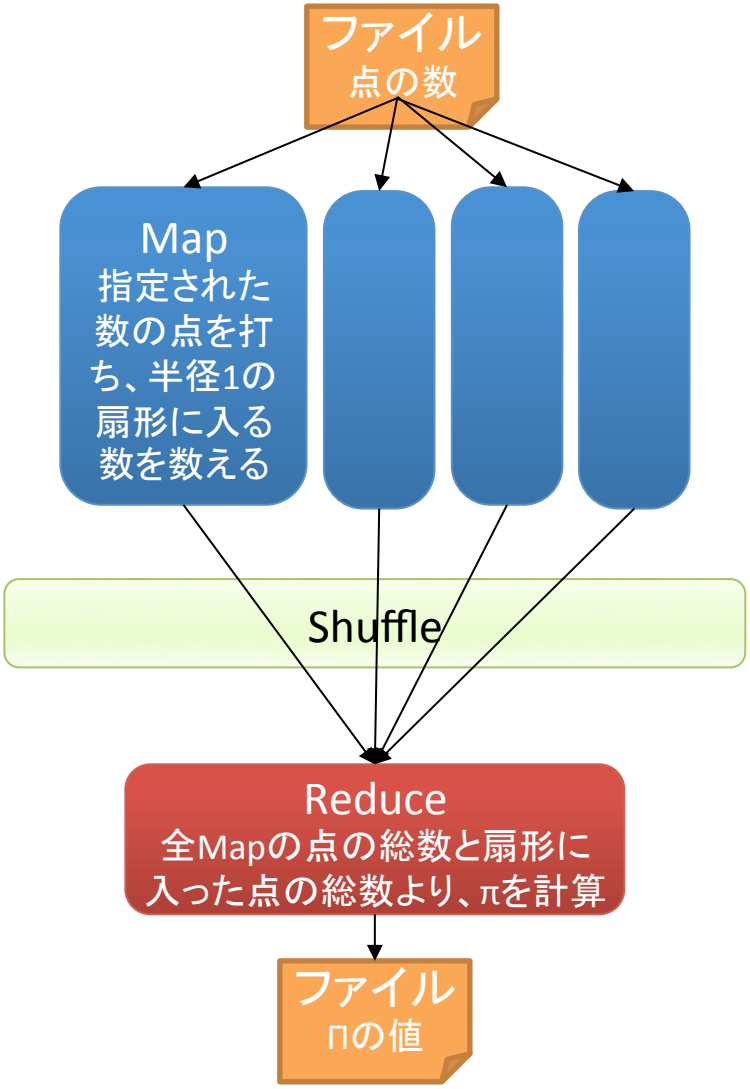
例：MapReduceによるPI計算

- モンテカルロ法によるPI計算
  - 1x1のサイズの正方形内にランダムに点を打つ
  - 点の総数（N）と、1x1の扇形内に入った点の数（M）を数える
  - 点の比率（M/N）は正方形の面積（1）と扇形の面積（π/4）の比率（π/4）に等しい
    - π/4 = M/N ➡ π = 4M/N



- MapReduceでの実装
  - Map：ランダムに点を打ち、NとMを数える
  - Reduce：すべてMapのNとMを集計し、πを計算

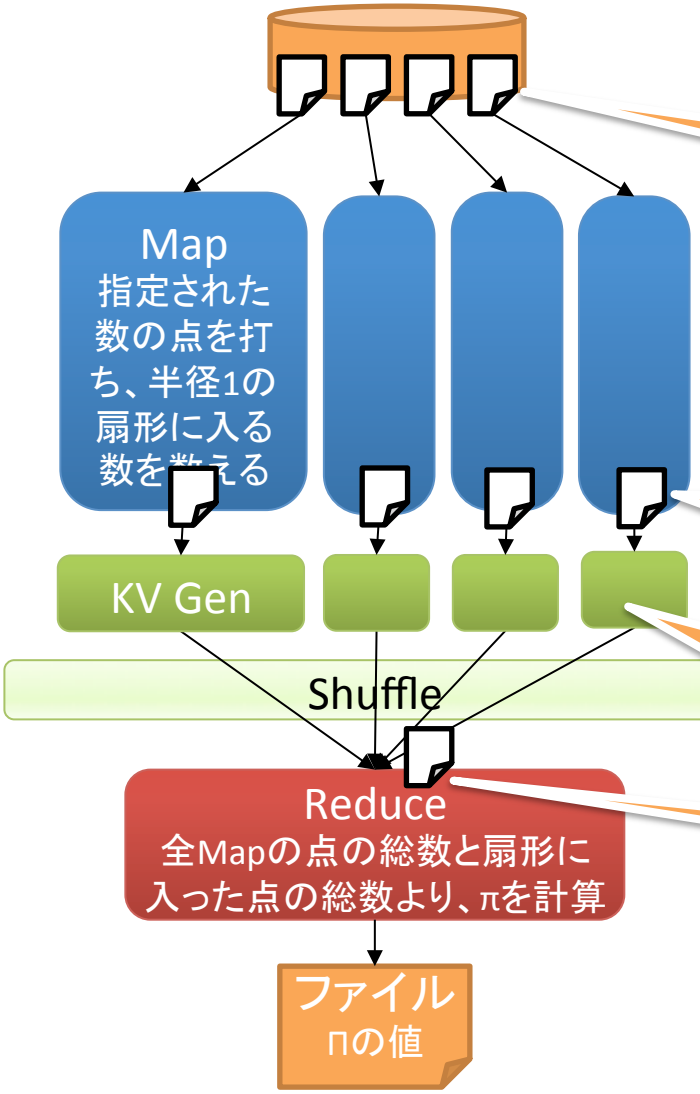
PI計算のMapReduce実行モデル



- Map
  - 複数プロセスが並行して点のプロット・カウント
  - Reduceする際に1プロセスに全結果を集めるため、KV <0, M/N>を出力
    - Key：全Mapで同じKey
    - Val：Mapごとの点の数
- Reduce
  - 全MapプロセスのMとNを合計し、πを計算

KMRRUNによるPI計算の実装

- MapとKV GenとReduceを実装



Mapプロセスごとに異なるファイルを読み込む
入力ファイル数がMap数

結果はファイルに書き出す

Mapの出力ファイルを読み込み、
標準出力にKV <0, M/N>を出力
Key数がReduce数

全MapのValueがファイル名「0」(Key名)ファイルに書き込まれる

PI計算サンプルプログラム

- PI計算のサンプルプログラムを2種類用意
  - 逐次プログラム版：KMR_SRC/kmrrun/
    - Mapper：pi.mapper.c
    - KV Generator：pi.kvgen.sh
    - Reducer：pi.reducer.c
  - MPIプログラム版：KMR_SRC/kmrrun/
    - Mapper：mpi_pi.mapper.c
    - KV Generator：mpi_pi.kvgen.sh
    - Reducer：mpi_pi.reducer.c
- 入力データの配置

```
inp/                内容
  ├── 000    (10000)
  ├── 001    (10000)
  └── 002    (10000)
```

PI計算の実行例

1. ファイル配置

```
$ ls
inp/  kmrrun  machines  pi.kvgen.sh  pi.mapper  pi.reducer
```

2. 実行

```
$ mpirun -machinefile machines -np 2 ./kmrrun ¥
-m ./pi.mapper -k ./pi.kvgen.sh -r ./pi.reducer ./inp
3.135600
$ ls
0.out    kmrrun      pi.kvgen.sh    pi.reducer
inp/     machines    pi.mapper
$ cat 0.out
3.135600
```

マシンファイルでは6ノード指定し、-npオプションでは2ノードを指定した場合

⇒ kmrrun, KV Generatorは2ノードで動作
⇒ Mapper, Reducerは残りの4ノードで動作
Mapperの入力ファイルは4個あるので、4 Mapperが同時実行可能

K computer

KMRRUN コツと注意点 (1/3)

- Mapperだけを実行したい
  - 結果をReduceする必要なく、異なるパラメータで複数の計算を実行したい場合など、Reducer実行が不要な場合があります。その時にはKV GeneratorやReducerは省略できます。

```
$ mpirun -np 4 ./kmrrun -n 8 -m ./mapper ./input
```

複数の計算を1つのジョブとしてまとめて実行したいときに有効

京のように、同時投入ジョブ数に制限がある場合に便利

KMRRUN コツと注意点 (2/3)

- 障害等で実行が中断した時に、実行を再開したい
  - KMRRUNでは実行状態の保存と再開を実現する機能 Checkpoint/Restart を提供します。再開時にはノード数を減らしての再開もサポートします。

```
$ mpirun -np 4 ./kmrrun -n 8 -m mapper -k kvgen.sh ¥
-r reducer --ckpt ./input
```

  - 中断時には、カレントディレクトリにチェックポイントファイル（ckptdirXXXXX, XXXXXはランク）を生成
  - チェックポイントファイルが存在する時に --ckpt オプションを指定すると、保存された状態から処理を再開

KMRRUN コツと注意点 (3/3)

- オプション「-n」(1回のMapper/Reducer実行時の使用プロセス数)の指定方法
  - Mapper/Reducerが逐次プログラムの場合 => 1
    - 1が指定された場合は、逐次プログラムと自動判定します
  - Mapper/ReducerがMPIプログラムの場合 => 2以上
    - MPIプログラムは並列度1では実行できません

- MPIで動作しているので、1プロセスで障害が起こると、全体の実行が中断する

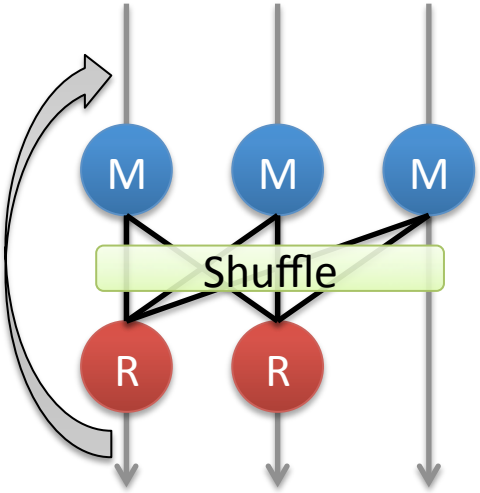
Agenda

- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- KMR利用事例
  - ゲノム解析
  - レプリカ交換分子動力学法
- まとめ

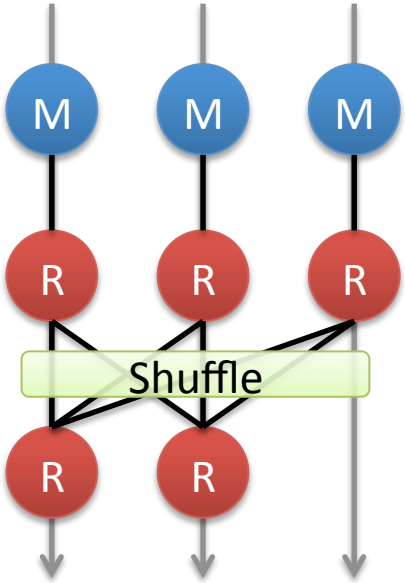
KMRライブラリ

- MapReduceプログラミングモデルに従った任意のワークフローを実装可能
  - MapReduceの繰り返し実行
  - Combiner（Shuffle前にReduce）実行

Iterative MapReduce



Combiner



- C/C++, Fortranにてプログラムを実装

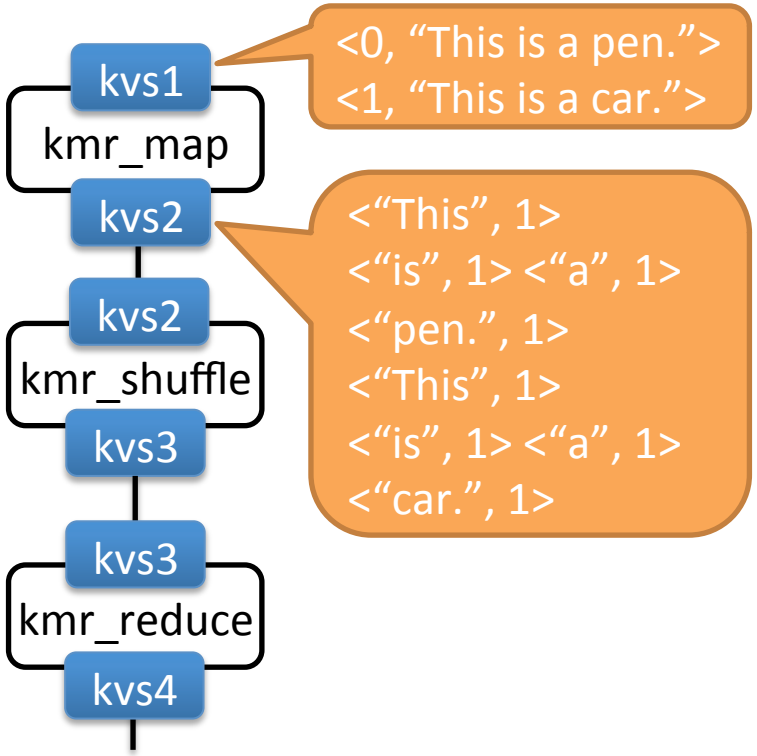
K computer

実装するもの

- Mapper関数
  - Map処理として実行される関数
  - 入力：1つのKV、またはファイル、メモリ上の変数
  - 出力：1つ以上のKV
- Reducer関数
  - Reduce処理として実行される関数
  - 入力：同じKeyを持つ、複数のKV
  - 出力：1つ以上のKV
- ドライバ関数（main関数）
  - KMR実行の初期化、終了処理
  - KMR関数群を呼び出し、MapReduce計算処理を実装

KMRの処理の流れ

- Key-Value Store (KVS)とは
  - 0個以上のKVを保持する メモリ上のデータ構造
  - KVSを単位にMapper、 Reducerを実行
    - 1つのKVSを入力とし、 個々のKVにMapper/Reducer 実行し、結果を出力KVSに 書き込む



- KMRライブラリを用いたプログラミングは、 KVSの変化の流れを記述すること
  - ドライバ関数に記述

K computer

ドライバ関数の実装

- 一連のMapReduce処理を行うプログラム

```
#include <mpi.h>
#include “kmr.h”

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    kmr_init();
    KMR *mr = kmr_create_context(MPI_COMM_WORLD, MPI_INFO_NULL, 0);

    KMR_KVS *kvs0 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs1 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs2 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs3 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    kmr_map(kvs0, kvs1, 0, kmr_noopt, mapfn);
    kmr_shuffle(kvs1, kvs2, kmr_noopt);
    kmr_reduce(kvs2, kvs3, 0, kmr_noopt, redfn);

    kmr_free_context(mr);
    kmr_fin();
    MPI_Finalize();
    return 0;
}
```

ドライバ関数の実装

- 一連のMapReduce処理を行うプログラム

```
#include <mpi.h>
#include "kmr.h"

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    kmr_init();
    KMR *mr = kmr_create_context(MPI_COMM_WORLD, MPI_INFO_NULL, 0);

    KMR_KVS *kvs0 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs1 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs2 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs3 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    kmr_map(kvs0, kvs1, 0, kmr_noopt, mapfn);
    kmr_shuffle(kvs1, kvs2, kmr_noopt);
    kmr_reduce(kvs2, kvs3, 0, kmr_noopt, redfn);

    kmr_free_context(mr);
    kmr_fin();
    MPI_Finalize();
    return 0;
}
```

MPIの1ライブラリなので、MPIのヘッダファイル読み込み、MPI初期化・終了処理を実装

K computer

ドライバ関数の実装

- 一連のMapReduce処理を行うプログラム

```
#include <mpi.h>
#include "kmr.h"

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    kmr_init();
    KMR *mr = kmr_create_context(MPI_COMM_WORLD, MPI_INFO_NULL, 0);

    KMR_KVS *kvs0 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs1 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs2 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs3 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    kmr_map(kvs0, kvs1, 0, kmr_noopt, mapfn);
    kmr_shuffle(kvs1, kvs2, kmr_noopt);
    kmr_reduce(kvs2, kvs3, 0, kmr_noopt, redfn);

    kmr_free_context(mr);
    kmr_fin();
    MPI_Finalize();
    return 0;
}
```

KMRの初期化・終了処理

ドライバ関数の実装

- 一連のMapReduce処理を行うプログラム

KVSの作成とMapReduce処理

```
#include <mpi.h>
#include “kmr.h”

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    kmr_init();
    KMR *mr = kmr_create_context(MPI_COMM_WORLD, MPI_INFO_NULL, 0);

    KMR_KVS *kvs0 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs1 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs2 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    KMR_KVS *kvs3 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_OPAQUE);
    kmr_map(kvs0, kvs1, 0, kmr_noopt, mapfn);
    kmr_shuffle(kvs1, kvs2, kmr_noopt);
    kmr_reduce(kvs2, kvs3, 0, kmr_noopt, redfn);

    kmr_free_context(mr);
    kmr_fin();
    MPI_Finalize();
    return 0;
}
```

K computer

KVSの作成

- 空のKVS (KMR_KVS型)を作成

```
KMR_KVS *kmr_create_kvs(KMR *mr, enum kmr_kv_field k,
                        enum kmr_kv_field v);
```

  - 第2引数でKeyの型、第3引数でValueの型を指定

KMR_KV_INTEGER	整数型(int)
KMR_KV_FLOAT8	浮動小数点型(double)
KMR_KV_OPAQUE	バイト列

- KVSを解放

```
int kmr_free_kvs(KMR_KVS *kvs);
```

  - KMR関数(kmr_map, kmr_reduce等)の入力となったKVSは自動で解放される

Mapper実行関数の利用

- 入力KVS中の個々のKVに対して(利用者定義)Mapper関数を実行

```
int kmr_map(KMR_KVS *kvi, KMR_KVS *kvo, void *arg,
            struct kmr_option opt, kmr_mapfn_t m);
```

KMR_KVS *kvi	入力KVS
KMR_KVS *kvo	出力KVS
void *arg	KVS以外の入出力データがあれば、 この変数にポインタとして指定 例)入力ファイル名
struct kmr_option opt	関数実行時のオプション 通常は「kmr_noopt」で可
kmr_mapfn_t m	利用者定義のMapper関数へのポインタ

- 関数終了後、入力KVSは解放される

Mapper関数の実装

- Mapper関数型

```
int (*kmr_mapfn_t)(const struct kmr_kv_box kv,
                   const KMR_KVS *kvi, KMR_KVS *kvo,
                   void *arg, const long index);
```

struct kmr_kv_box kv	Mapperの処理対象KV
KMR_KVS *kvi	入力KVS(参照のみ)
KMR_KVS *kvo	出力KVS
void *arg	Mapper実行関数の第3引数として渡されたポインタ
long index	KVS中のKVのインデックス

– 個々のKVへの処理を定義する

Mapper関数の実装 – 例

- k-means法(KMR_SRC/ex/kmeans-kmr.c)
  – ある点1つが、どこの集合に属す

```
int calc_cluster(const struct kmr_kv_box kv,
                 const KMR_KVS *kvi, KMR_KVS *kvo
  int i;
  kmeans_t *kmeans = (kmeans_t *)p;
  int dim    = kmeans->dim;
  int *means = kmeans->means;
  int n_means = kmeans->n_means;
  int *point = (int *)kv.v.p;
  int min_idx = 0;
  unsigned int min_dist = calc_sq_dist(point, &means[
  for (i = 1; i < n_means; i++) {
    unsigned int dist = calc_sq_dist(point, &means[i
    if (dist < min_dist) {
      min_idx  = i;
      min_dist = dist;
    }
  }
  struct kmr_kv_box nkv = { .klen = sizeof(long),
                            .vlen = dim * sizeof(int),
                            .k.i  = min_idx,
                            .v.p  = (void *)point };
  kmr_add_kv(kvo, nkv);
  return MPI_SUCCESS;
}
```

ポインタpにはk-means実行条件を保存

Valueに点の座標が保存されている

対象の点と、全ての集合との距離を計算し、最小の集合を選択

Keyに集合のID、Valueに点の座標、となるKVを生成し、出力KVSに追加

K computer

他の代表的なMapper実行関数

- ファイル等からKVを作成し、KVSに保存

```
int kmr_map_once(KMR_KVS *kvo, void *arg,
                 struct kmr_option opt, _Bool rank_zero_only,
                 kmr_mapfn_t m)
```

  - Mapper関数(m)にて、ポインタ(*arg)を参照してKVを作成
  - Mapper関数(m)は1度しか実行されないため、1度の実行で必要なKVを全て作成すること
- MPIプログラムを実行し、その出力ファイルを元に作成したKVを出力KVSに保存

```
int kmr_map_via_spawn(KMR_KVS *kvi, KMR_KVS *kvo, void *arg,
                      MPI_Info info, struct kmr_spawn_option opt,
                      kmr_mapfn_t mapfn);
```

  - 入力KVS(kvi)中のKVのValueにMPIプログラムを指定
  - Mapper関数(mapfn)では、MPIプログラムの出力ファイルを読み込み、その内容に応じたKVを作成するよう実装

（参考：KMR_SRC/src/testf.f90、KMR_SRC/kmrrun/kmrrun.c）

Reducer実行関数の利用

- 入力KVS中の個々のKVに対して(利用者定義)Reducer関数を実行

```
int kmr_reduce(KMR_KVS *kvi, KMR_KVS *kvo, void *arg,
               struct kmr_option opt, kmr_redfn_t r);
```

KMR_KVS *kvi	入力KVS
KMR_KVS *kvo	出力KVS
void *arg	KVS以外の入出力データがあれば、 この変数にポインタとして指定 例)入力ファイル名
struct kmr_option opt	関数実行時のオプション 通常は「kmr_noopt」で可
kmr_redfn_t r	利用者定義のReducer関数へのポインタ

- 関数終了後、入力KVSは解放される

Reducer関数の実装

- Reducer関数型

```
int (*kmr_redfn_t)(const struct kmr_kv_box kv[], const long n,
                   const KMR_KVS *kvi, KMR_KVS *kvo,
                   void *arg);
```

`struct kmr_kv_box kv[]`	Reducerの処理対象KV 全てのKVのKeyは等しい
`long n`	KVの数
`KMR_KVS *kvi`	入力KVS(参照のみ)
`KMR_KVS *kvo`	出力KVS
`void *arg`	Reducer実行関数の第3引数として渡されたポインタ

– Keyの等しい、複数のKVへの処理を定義する

Reducer関数の実装 – 例

- k-means法(KMR_SRC/ex/kmeans-kmr.c)
  – 集合の中心座標の更新

```
int update_cluster(const struct kmr_kv_box kv[], const
                   const KMR_KVS *kvi, KMR_KVS
  int i, j;
  int cid = (int)kv[0].k.i;
  kmeans_t *kmeans = (kmeans_t *)p;
  int dim = kmeans->dim;
  int average[dim];

  for (i = 0; i < dim; i++)
    average[i] = 0;
  for (i = 0; i < n; i++) {
    int *point = (int *)kv[i].v.p;
    for (j = 0; j < dim; j++) {
      average[j] += point[j];
    }
  }
  for (i = 0; i < dim; i++)
    average[i] /= n;
  struct kmr_kv_box nkv = { .klen = sizeof(long),
                            .vlen = dim * sizeof(int),
                            .k.i  = cid,
                            .v.p  = (void *)average };
  kmr_add_kv(kvo, nkv);
  return MPI_SUCCESS;
}
```

Keyは集合のID、Valueはその集合に属す点の座標

ポインタpにはk-means実行条件を保存

集合に含まれる全点の座標から、集合の中心座標を計算

Keyに集合のID、Valueに集合の中心座標、となるKVを生成し、出力KVSに追加

K computer

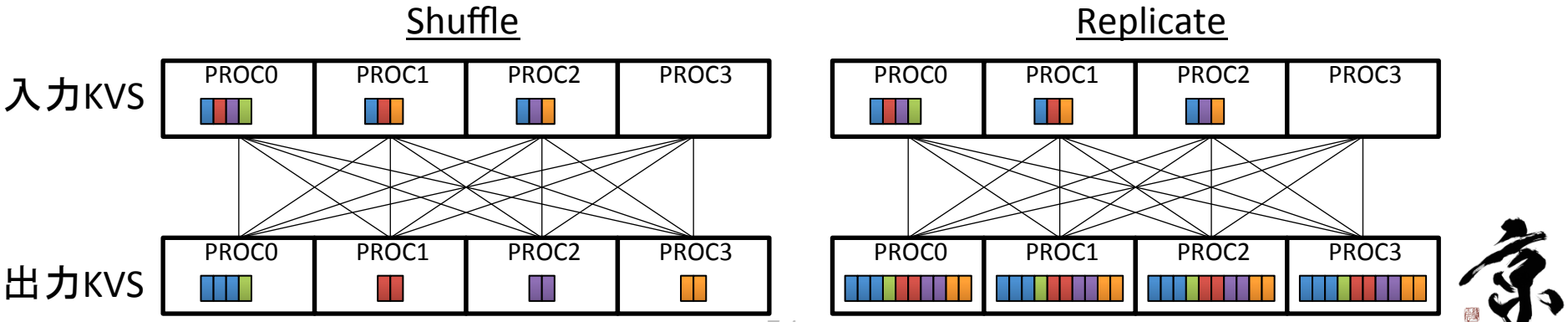
通信関数の利用

- ShuffleとReplicate

```
int kmr_shuffle(KMR_KVS *kvi, KMR_KVS *kvo, struct kmr_option opt);
int kmr_replicate(KMR_KVS *kvi, KMR_KVS *kvo, struct kmr_option opt);
```

KMR_KVS *kvi	入力KVS
KMR_KVS *kvo	出力KVS
struct kmr_option opt	関数実行時のオプション 通常は「kmr_noopt」で可

- ReplicateはMapReduce実行結果のKVSを全プロセスで共有する時に有用



K computer

他の有用な関数

- KVSの中身をソート

```
int kmr_sort(KMR_KVS *kvi, KMR_KVS *kvo, struct kmr_option opt);
```

  - 入力KVS(kvi)内のKVをKey順にソートし、出力KVS(kvo)に保存
  - kmr_reduceは結果をソートしないため、ソートした結果を必要とする場合には、Reduce後、この関数を実行

- マルチノード高速ファイル読み込み

```
int kmr_read_file_by_segments(KMR *mr, char *file, int color,
                              void **buffer, off_t *size);
```

  - ファイル(file)を読み込み、内容をメモリ(buffer)上に展開

コンパイルと実行

- MPIコンパイラでコンパイル
  - ヘッダファイル検索ディレクトリの指定、ライブラリとリンク
  - 京、FX10

```
$ mpifccpx -Kopenmp,fast -I KMR_INST/include ¥
SOURCE_CODE.c KMR_INST/lib/libkmr.a
```

  - Linux/Solaris + gcc + OpenMPI

```
$ mpicc -O3 -I KMR_INST/include SOURCE_CODE.c ¥
KMR_INST/lib/libkmr.a
```

- 実行

```
$ mpiexec -np 4 ./a.out
```

  - 環境変数設定

```
$ cat kmrrc
verbosity=9
$ KMROPTION=kmrrc mpiexec -np 4 ./a.out
```

環境変数

verbosity	1~9を指定（デフォルトは5）。数値を小さくすると、デバッグ用メッセージを表示しなくなる
trace_map_spawn	0/1を指定（デフォルトは0）。kmr_map_via_spawn()の実行をトレース

サンプルプログラム

- KMR_SRC/ex 以下にいくつか
  - 付属のMakefileにMakeターゲットが定義されています
  - 例)k-means法のサンプルプログラム(kmeans-kmr.c)のコンパイル

```
$ cd KMR_SRC/ex
$ make kmeans
```

KMRライブラリ 注意点

- Mapper/Reducerの計算量を考慮して並列度(-np)を指定する
  - Mapper/Reducer共に同じ並列度で動作します
  - 入力ファイル(Mapperへの入力)がたくさんあるが、出力するKeyの数が少ない場合、高い並列度を指定すると、Reduce時に資源の無駄が生じ得ます(逆もしかり)
    - 例) 入力ファイルは1,000個、Map結果の出力Keyは10個
      => 並列度1,000で実行すると、Reduceにて990の無駄
- Reduce結果はソートされない
  - HadoopではReduce結果はソートされて出力されるが、KMRではソートしません
- MPIで動作しているので、1プロセスで障害が起こると、全体の実行が中断する

Agenda

- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- **KMR利用事例**
  - **ゲノム解析**
  - **レプリカ交換分子動力学法**
- まとめ

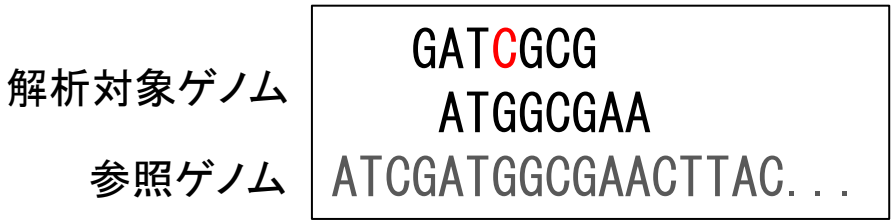
KMR利用事例

- 計算科学アプリケーションにKMRを適用した事例を紹介
  - アプリケーションのワークフロー（処理の流れ）をMapReduceモデルにて記述

- ゲノム解析
- レプリカ交換分子動力学法

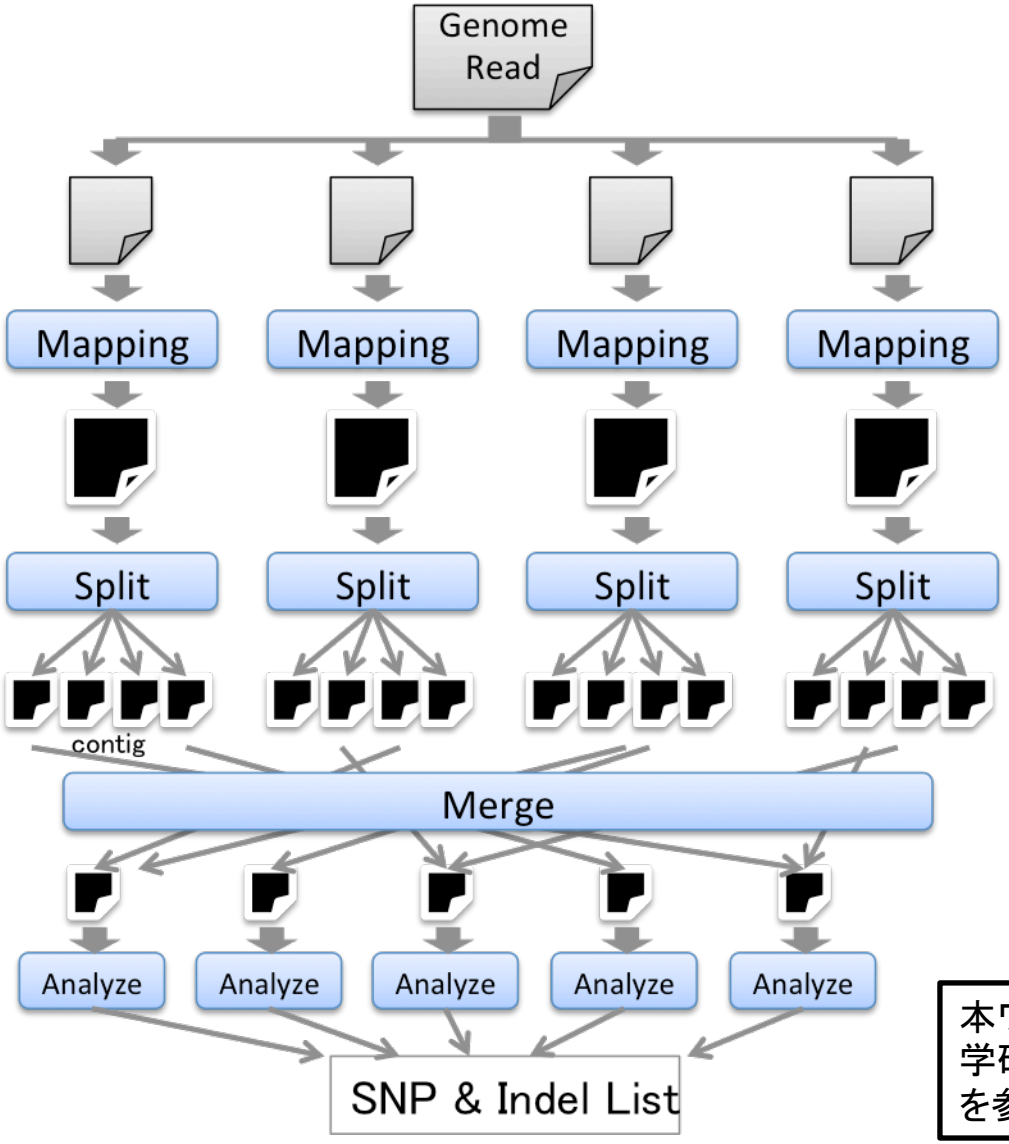
ゲノム解析

- ゲノム解析
  - DNAシークエンサーにより読み取られた膨大なゲノムデータと、参照ゲノムデータ(既知のゲノム配列)の差異を解析



- パイプライン実装
  - 一連の解析を行うために、種々のツール群を組み合わせて実装
    - シーケンスマッピングツール
    - データフォーマットツール
    - 解析ツール

ゲノム解析：ワークフロー



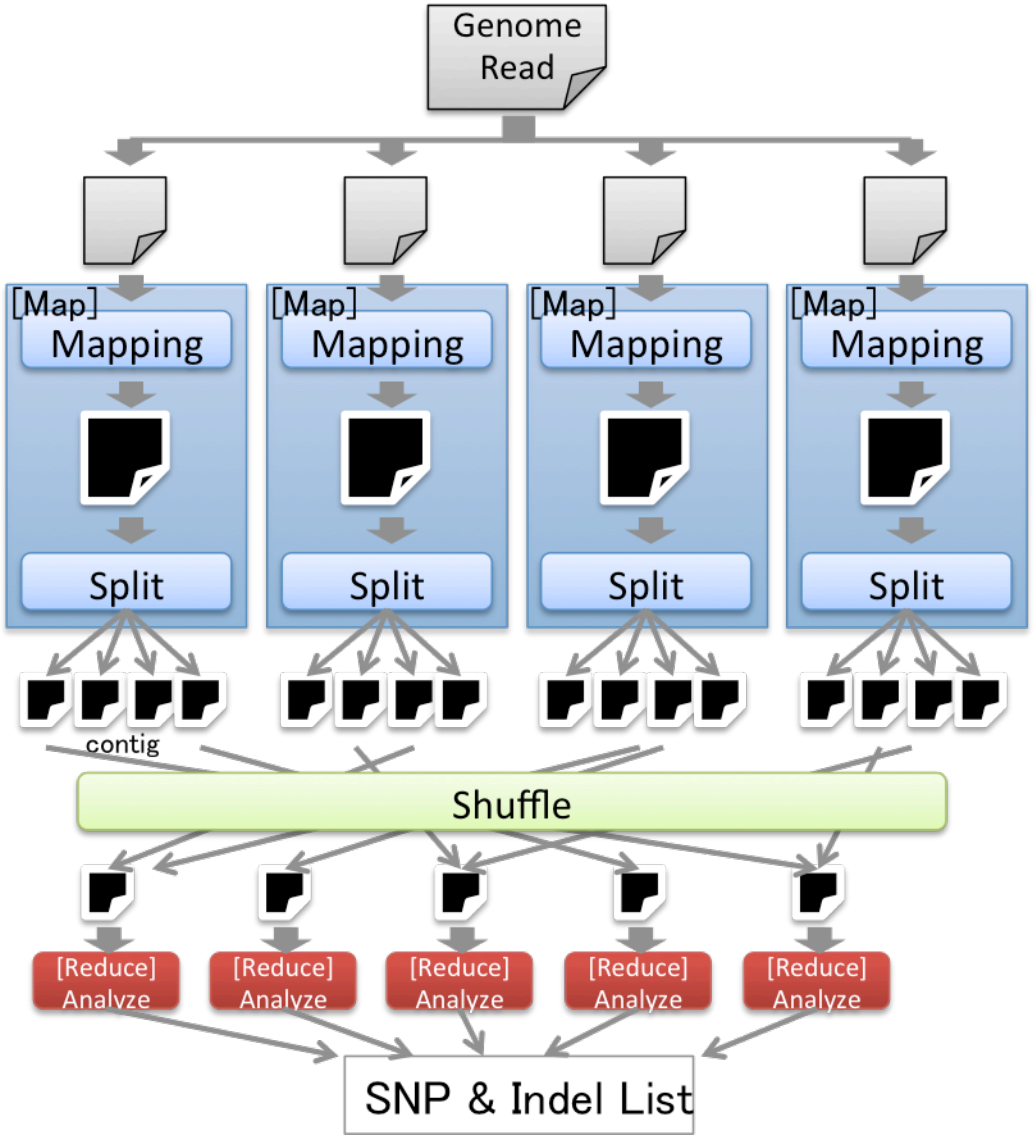
Mapping	リードを参照ゲノムにマッピング
Splits	contig（連続する塩基配列パターン）毎にMapping結果を分割
Merge	contig毎に結果をマージ
Analyze	contig単位に変異解析

- 各タスク間のデータの受け渡しはファイルベース
- Mergeは共有ファイルシステム上で行われる

本ワークフローは、理化学研究所 統合生命医科学研究センターにて開発されているNGS Analyzerを参考にしている

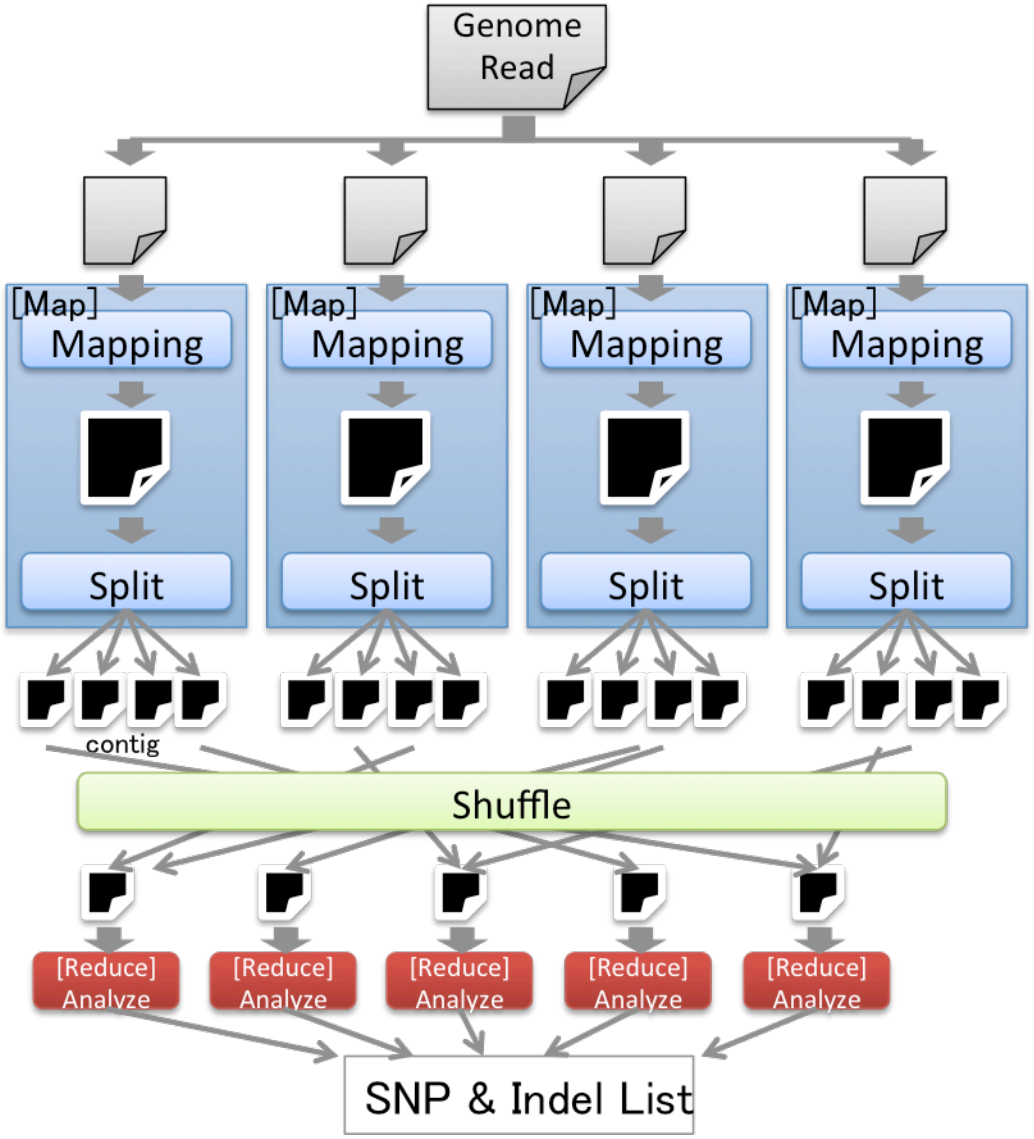
K computer

ゲノム解析：MapReduce実装 (1/2)



- Map処理
  - MappingとSplitを実行
  - KV <contig, マッピング結果> を出力
- Reduce処理
  - KV <contig, マッピング結果> を入力
  - Anayzeを実行
- MergeはShuffleで置き換え
  - IOを伴わない、オンメモリ転送

ゲノム解析：MapReduce実装 (2/2)



- KMRRUNにて実装
  - Mapper/Reducerともに Pythonにて逐次プログラムとして実装
    - ワークフロー実行管理は KMRRUNが行う
  - 合計111行
    - NGS Analyzer: 748行
      - ワークフロー実行管理含む

K computer

ゲノム解析：性能

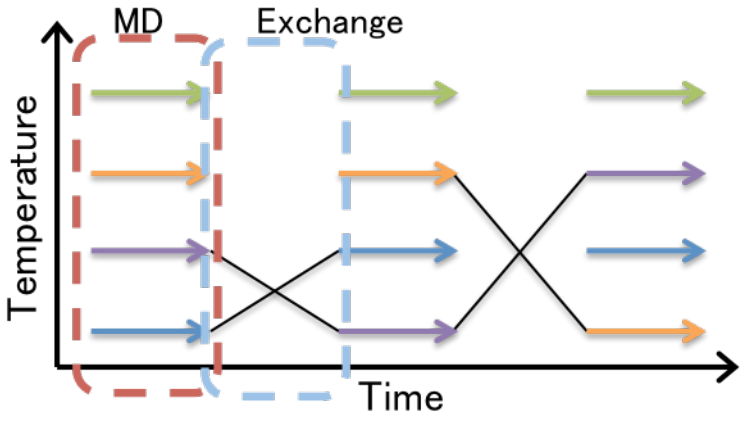
- 日本人全ゲノム解析
  - ヒト一人全体で490GB
    - 25万配列毎にファイル分割、Mapperの入力へ
  - 参照ゲノム：6.3GB
- 実行環境：京コンピュータ
  - 1 MPIプロセス / ノード、16GBメモリ、ノード内並列なし
  - 1Mapper / ノードのタスク割り当て

データサイズ	118 MB / 12 Nodes	87.9 GB / 512 Nodes	490 GB / 4160 Nodes
NGS Analyzer シェル	357秒	4,985秒	22,848秒
KMR実装	353秒	3,691秒	14,593秒

NGS Analyzerシェル：
NGS Analyzerのワークフローを忠実に1つのシェルスクリプトとして実装（124行）

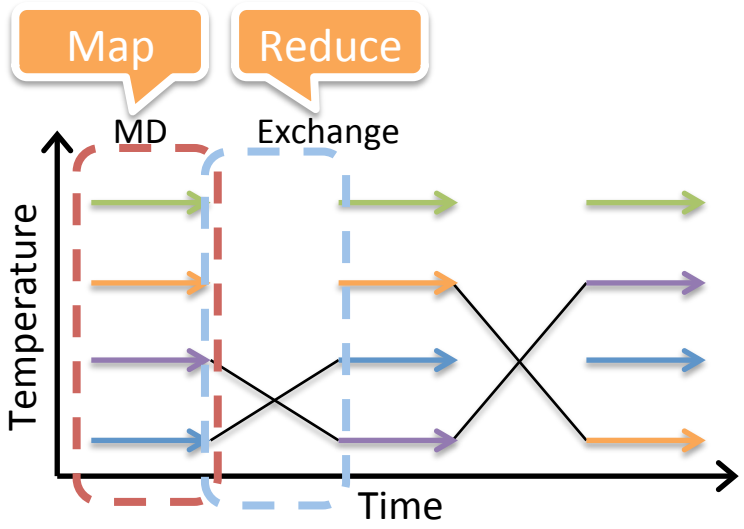
レプリカ交換分子動力学法

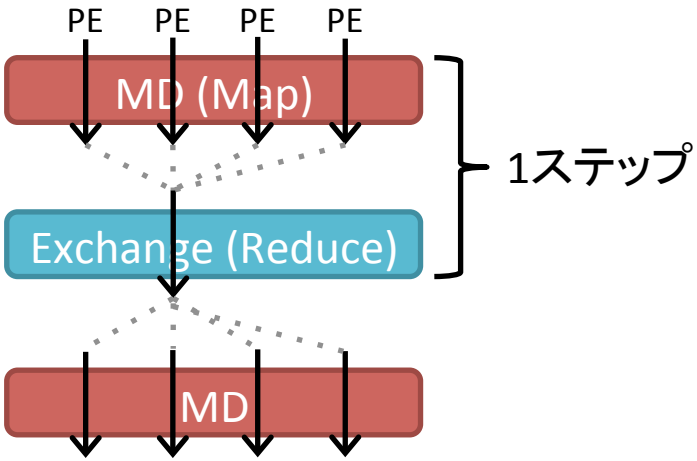
- レプリカ交換分子動力学法(REMD)
  - 創薬等で、タンパク質の構造解析に用いられる1手法
- ワークフロー
  - 分子構造の複数のレプリカを用意し、それぞれに異なる温度を割り振り、構造サンプリング(MD)を行う
  - MD実行後、レプリカの温度パラメータを交換(Exchange)
  - 以上を繰り返し実行する、アンサンブル計算



K computer

REMD：MapReduce設計





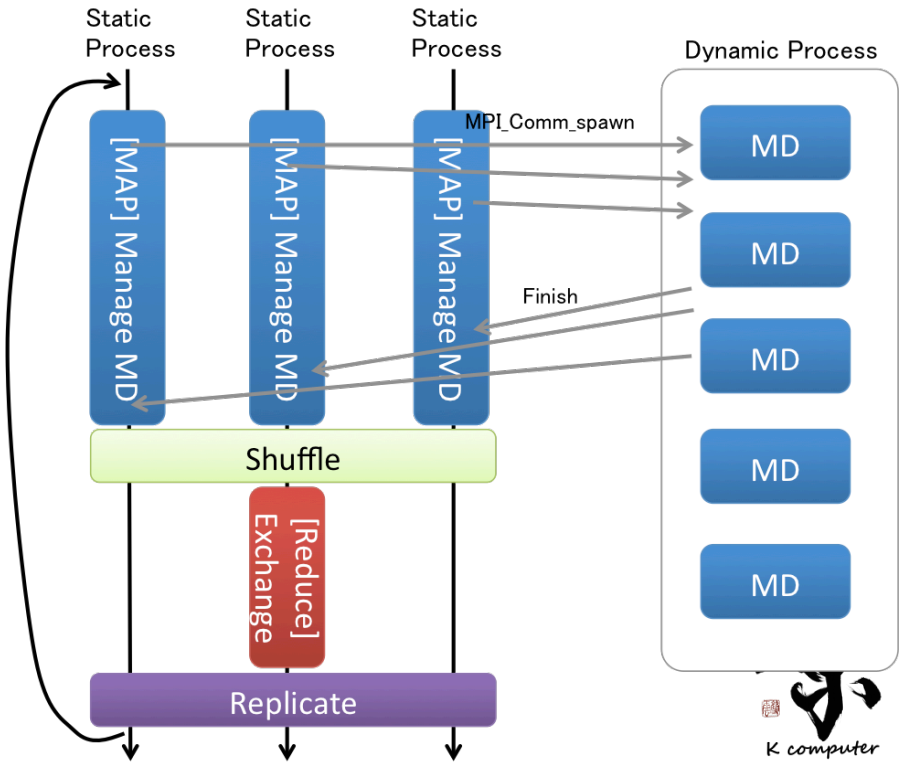
- REMDの1ステップを1 MapReduceとして実装
  - [Map] 複数プロセスで並列にMD計算
  - [Reduce] 1プロセスにMD結果を集約し、交換条件を計算
- REMDステップを繰り返し実行
  - Iterative MapReduce
  - 次のステップ開始前に、温度交換結果を全プロセスで共有

K computer

REMD: KMR実装 (1/2)

- KMRライブラリを用いてFortranで実装
- Map処理
  - MD実行条件を設定し、MDを実行
    - MDには NAMD2 を使用 => kmr_map_via_spawn()にて起動
  - 入力KV：<レプリカID, 入力ファイルパス>
  - 出力KV：<0, レプリカIDとエネルギー>
- Reduce処理
  - 温度交換の計算
  - 入力KV：<0, レプリカIDとエネルギー>
  - 出力KV：<結果の種類を表す文字列, Exchange結果>
- 結果共有
  - Reduce完了後、kmr_replicate()にて全プロセスで結果を共有

本実装は、理化学研究所 計算科学研究機構 粒子系生物物理研究チームにて開発されているREINを参考にしている。



REMD: KMR実装 (2/2)

- 実際にはMap処理、Reduce処理双方にて、多数のMapper/Reducerを実行
- Map処理

MD入力ファイルの作成
- [入力] Key: Replica ID, Val: Replica ID
- [出力] Key: Replica ID, Val: MDコマンド

MD Rescaling
- [入力] Key: Replica ID, Val: MDコマンド
- [出力] Key: Replica ID, Val: MDコマンド

MD実行
- [入力] Key: Replica ID, Val: MDコマンド
- [出力] Key: Replica ID, Val: Replica ID

MD出力結果の取得
- [入力] Key: Replica ID, Val: Replica ID
- [出力] Key: Replica ID, Val: Replica ID

Shuffle用KVSの作成
- [入力] Key: Replica ID, Val: Replica ID
- [出力] Key: 0, Val: Replica ID & エネルギー

- Reduce処理

構造型へのレプリカ情報の書き込み
- [入力] Key: 0, Val: Replica ID & エネルギー
- [出力] Key: 0, Val: Replica ID

エネルギートラジェクトリファイルの作成
- [入力] Key: 0, Val: Replica ID
- [出力] Key: 0, Val: Replica ID

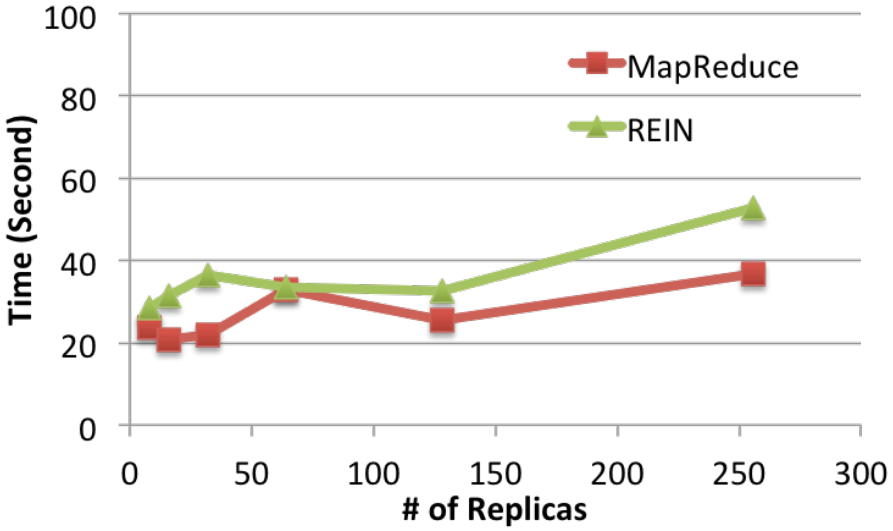
Exchange実行
- [入力] Key: 0, Val: Replica ID
- [出力] Key: 0, Val: Replica ID

リスタートファイルの作成
- [入力] Key: 0, Val: Replica ID
- [出力] Key: 0, Val: Replica ID

Replicate用KVSの作成
- [入力] Key: 0, Val: Replica ID
- [出力] Key: 文字列, Val: Exchange結果

REMD: 性能

- 実行設定
  - データ:REIN付属のサンプルデータ
    - 1次元REMD、10 Iteration
    - レプリカ数:8, 16, 32, 64, 128, 256
- 実行環境: 京コンピュータ
  - 使用ノード数:9, 18, 36, 72, 144, 288
    - レプリカ数に応じて設定
  - 1 MD実行で使用するMPIプロセス数:8(1ノード)



K computer

Agenda

- MapReduceプログラミングモデル
- K MapReduce (KMR)
  - 概要・特徴
  - 利用方法
    - KMRRUNによる簡易MapReduce実行
    - KMRライブラリを用いたプログラミング
- KMR利用事例
  - ゲノム解析
  - レプリカ交換分子動力学法
- まとめ

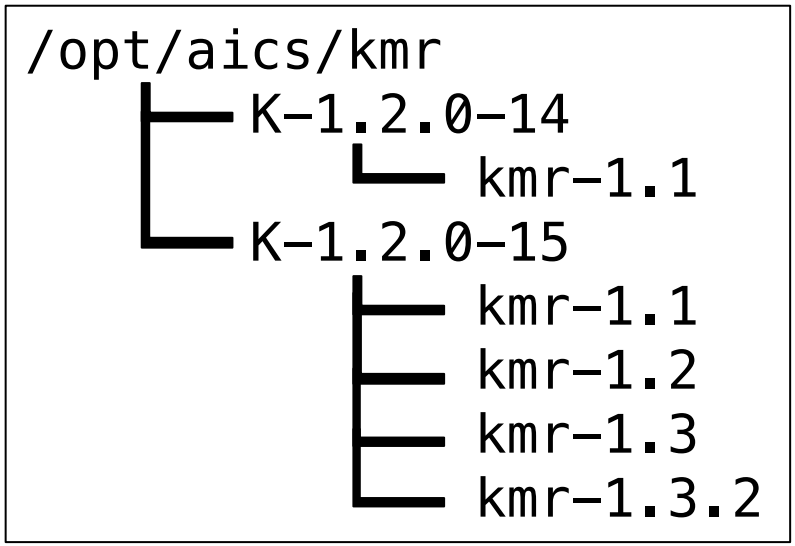
まとめ

- MapReduceプログラミングモデルの紹介
- KMRを用いたMapReduceプログラムの実行
  - KMRRUNによる簡易実行
  - KMRライブラリを用いたプログラム実装

- KMRを用いた、MapReduceモデルによる計算科学アプリケーションの事例紹介
  - ゲノム解析
  - レプリカ交換分子動力学法

京での利用

- /opt/aics/kmrにインストール済み
- ディレクトリ構成



– KMR更新時、または言語環境更新時にその時々のバージョンにあったKMRをインストールしていきます

おわり

ご質問・お問い合わせ

丸山: nmaruyama
松田: m-matsuda
滝澤: shinichiro.takizawa @riken.jp

K computer