並列トランザクショナルアプリケーションのためのプログラミングフレームワーク

水野 謙[†] 菅沼俊夫[†] 石崎一明[†]
古関 聰[†] 上田陽平[†] 小松秀昭[†]

インターネットの普及などによって，アプリケーションが扱うべきデータの量は爆発的に増加している．このような状況に対応するために，計算ノードの追加によって性能の向上を実現するスケールアウトという考え方がある．たとえば Web 検索の分野では，Google の MapReduce やオープンソースプロジェクトの Hadoop といったスケルトン並列プログラミングの考え方に基づいたシステムがこれを実現している．スケールアウトが求められている分野の 1 つに，バッチ処理がある．バッチ処理ではデータの保持のために主にデータベースを利用している．そのため，単純に並列化すると，データアクセスのための通信や特定のノードへのアクセスの集中により，十分なスケーラビリティを得ることができない．そこで我々は，データベースを利用するアプリケーションのスケールアウトを実現するためのシステムを作成した．本システムでは，データをユーザが指定する方法によって分散させ，Owner-Computes Rule に従って計算を行う．ユーザは，Java を拡張した言語を用いて通信や同期など並行処理の詳細について意識することなく，高い抽象度でプログラムを記述することができる．また，耐故障性を備えており，特別なハードウェアを利用しなくても高い信頼性を実現できる．本論文では，本システムで利用する言語の仕様について説明する．

A Programming Framework for Parallel Transactional Applications

Ken Mizuno,[†] Toshio Suganuma,[†] Kazuaki Ishizaki,[†]
Akira Koseki,[†] Yohei Ueda[†] and Hideaki Komatsu[†]

Recent advances in Internet-technology-based services are causing an explosion of transactions in many business operations. To respond in such business environments, systems have to be able to deal with the ever-increasing workload requirements in a scale-out fashion. Google's MapReduce system and Apache Hadoop are examples that successfully provide programming environments for developing Web search applications that can scale out. Batch processing systems also need to scale out. They use databases to store the data and update the database frequently. Thus, simple parallelization causes conflicts over data access and prevents the system from scaling. We designed a system to develop scale-out applications that use databases. In this system, users specify the data distribution and the computations are based on the owner-computes rule. Users develop their programs using a Java-based language without considering the complex details of parallel programming. It is a fault-tolerant system and provides high availability without special hardware. This paper describes the specification of the language used in this system.

1. はじめに

インターネットの発達により，アプリケーションが扱うデータの量は急激に増加している．このような，増加し続ける膨大なデータを扱うシステムを実現するための方法として，クラスタなどを用いてマシンの増強にともない比例した性能を実現する，スケールアウトの考え方がある．Web 検索の分野では，すでに Google の MapReduce [6] や，オープンソースプロジェクトの Hadoop [1] などがあり，スケールアウトするシステムの実現に成功している．

スケールアウトが求められている分野の 1 つに，複数のトランザクションをまとめて処理を行うバッチ処理がある．たとえばクレジットカードの決済処理は，オンラインショッピングや電子マネーの普及によって今後処理量が増加することが予想される．このようなアプリケーションではデータの保持のためにデータベースを利用しており，アプリケーションを単純に並列化すると，データベースへのアクセスのための通信

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.

や，特定のノードへのアクセスの集中により，十分なスケーラビリティを得ることができない．

そこで我々は，データベースを利用するアプリケーションのスケールアウトを実現するためのシステムを開発した．本システムには以下のような特徴がある．

- スケールアウトの実現
  ローカルディスクを持つ多数の計算機を利用し，データは各計算機に分散して保持する．計算の分散は，計算の並列性を考慮するとともに，データへのアクセスが自ノード内で閉じるように，データの分散に合わせて指定する．これにより，同一データへのアクセスによるロックの競合や，データアクセスのための通信をなくすことができる．また，計算にともなうすべてのディスクアクセスは自ノード内に閉じるので，システム全体のディスク転送速度を最大限に引き出すことができる．結果として，高いスケーラビリティを実現できる．
- トランザクションのサポート
  データはデータベースやファイルに保存され，データ更新にともなう不整合を防ぐためのトランザクション処理をサポートする．トランザクション処理が複数の計算機にわたる場合でも，データの整合性は保証される．
- 高い生産性
  並列スケルトン[4),12)]の考え方を基にしており，プログラマは通信や同期など並列処理のための詳細を意識することなく，プログラムのロジックに集中してプログラムを記述することができる．
- 耐故障性
  大量の計算機を利用すると，故障が発生しやすくなる．本システムでは計算の途中で一部の計算機が故障しても計算を継続することができ，特別なハードウェアを利用しなくても高い信頼性を実現できる．

本論文では，これらの特徴を実現するために我々が作成した，Java を拡張した言語について，その仕様を説明する．2 章で関連研究について述べた後，3 章で設計方針について，4 章で言語仕様の詳細について説明する．その後，5 章で我々が実装したシステムについて概説し，そのシステムでの性能を 6 章で示す．

2. 関 連 研 究

並列プログラムを記述するための仕組みの 1 つに，スケルトン並列ライブラリ[4),12)]がある．これは，Map や Reduce といった，並列プリミティブと呼ばれる処理の組合せでプログラムを記述し，各プリミティブの実行はシステムが自動的に並列化して行うものである．これにより，ユーザは同期処理やスケジューリングなどの並列処理の詳細を気にすることなく並列プログラムを記述することができる．

スケルトン並列ライブラリの成功例として，Web 検索アプリケーション向けのライブラリである MapReduce[6)] や Hadoop[1)] がある．Web 検索は大量のデータを扱うアプリケーションの 1 つで，多くの場合入力はファイルからの逐次読み取り，出力はファイルへの追記であり，ランダムアクセスやファイルの更新は存在しない．MapReduce や Hadoop は，このような特徴に特化することで，高い並列性，信頼性，記述性を実現している．

また，データフローグラフに基づく並行プログラミングモデルとして，Dryad[8)] や WebSphere Datastage[3)] がある．Dryad では computation vertices と communication edges からなるデータフローグラフを記述し，並列実行環境において computation vertices を各 CPU に割り当てることで並列実行を行う．また，WebSphere Datastage では，入出力データとプロセスを接続することでプログラムを記述する．入力データはユーザが指定したキーによって分割され，並列に実行される．

これらの並列計算モデルと本研究との最も大きな違いは，トランザクション処理のサポートである．スケルトン並列ライブラリでは各並列プリミティブの並列性を利用して，プリミティブ単位での並列化を行うが，トランザクションをサポートするためには各プリミティブの実行制御に加えてトランザクション単位での制御が必要になる．

また，MapReduce や Hadoop との違いとしてデータの分散方法の指定がある．MapReduce や Hadoop では入出力データの保持にグローバルファイルシステム[13)]を利用しており，その配置はファイルシステムが自動的に決定する．一方，本システムではデータの分散方法はユーザが明示的に指定する．データベースでは複数のテーブルやフィールドにまたがって関連性の強いデータが存在することが多い．関連性の強いデータは同時にアクセスされることが多いので，これらのデータを同一の計算機が保持するようにユーザがデータの分散を指定することで効率良く実行できるようになる．

並列実行環境においてデータベースを扱う方法として分散データベース[11)]がある．分散データベースではデータを複数の計算機に分散して保持しているが，アプリケーションからは単一のデータベースとして見

える．そのため，アプリケーションはデータの実際の分散方法を意識する必要がなく，アプリケーションからのリクエストは分散データベースシステムが自動的に分散処理する[5),9),10)]．一方，本システムでは，計算にともなうデータのアクセスが自ノード内で閉じるように，ユーザが明示的にデータの分散方法を指定する．この結果，ユーザに制御されたデータアクセスや通信に従って，効率の良い実行を実現する．

3. 設計方針

1 章で説明した特徴を実現するために採用した設計方針について説明する．

3.1 スケールアウトの実現

データベースを利用するアプリケーションでは，アプリケーション自身のスケールアウトだけでなくデータベースのスケールアウトも考える必要がある．

データベースをスケールアウトさせる方法として，分散データベースがある．分散データベースではデータベースに格納するデータを複数の計算機に分散して配置する．アプリケーションからは仮想的に 1 つのデータベースに見えており，アプリケーションがどの計算機にアクセスしても，データベースシステムは内部で目的のデータが存在する計算機にアクセスして，データを取得・更新する．この方法では，データベースの分散方法と独立に，アプリケーションの計算を分散実行することができる．しかし，計算ノードからのデータアクセスのための通信や，複数の計算ノードからの同一データへのアクセスによるロックの競合が発生し，スケーラビリティが低下してしまう．

本システムでは，ユーザはアクセスしようとするデータの分散に合わせて，計算にともなうデータのアクセスが自ノード内で閉じるように，計算の分割を指定し，計算は Owner-Computes Rule[7),14)] によって行われる．この結果，各ノードにおけるデータアクセスはローカルな操作となり，同一データへのアクセスによるロックの競合や，データアクセスのための通信をなくすことができる．この結果，高いスケーラビリティの実現が可能である．異なるノードに保存されているデータにアクセスする必要がある場合には，明示的に通信を行って計算を移動させる．

3.2 トランザクションのサポート

MapReduce や Hadoop では，データの更新などの副作用のある操作は行わない．しかし，バッチ処理ではデータ更新を行う必要があり，これを実現するためにトランザクション処理をサポートしている．また，トランザクショナルファイルを用意し，データベースだけではなくファイルにおいてもトランザクション処理を行えるようにした．

```
batchMain () {
   /* batch iteration outer loop */
   for (input = inputFile.getRecord (); input != null; ) {
      tx.begin();
      /* batch iteration inner loop */
      for (; checkPointInterval(); ) {
         /* get objects from DB */
         customer = getCustomer(input.customerID);
         account = getAccount(customer.accountKey);

         /* update account balance data to database */
         new_bal = account.balance - input.payment;
         account.setBalance(new_bal);
         /* read the next input data */
         input = inputFile.getRecord();
      }
      tx.end();
   }
}
```

図 1 バッチプログラムの例
Fig. 1 An example of batch programs.

```
task {
  subtxn(inputSet, inputFile) { // read the input data
    customerSet = creader.read(customerTable, inputSet);
    accountSet = areader.read(accountTable, customerSet);
    subtractor.join(acountSet, customerSet, new_bal);
    aupdater.write(new_bal, accountTable);
  }
}
```

図 2 本システムを用いたプログラムの例
Fig. 2 An example of programs that uses our system.

3.3 生産性

本システムでは，並列スケルトンの考え方に基づき，抽象度の高い記述を実現している．

逐次処理を行うバッチプログラムの例と，それを本システムを用いて並列処理するように記述した例をそれぞれ**図 1** と**図 2** に示す．図 2 の変数 `inputFile` はファイル，`customerTable` および `accountTable` はそれぞれデータベースのテーブルを表しており，それ以外の変数はデータの集合を表している．

このように分散データの集合に対して処理を記述すると，システムが並列に処理を行う．ユーザが並列処理のための詳細について意識する必要はない．具体的な処理の内容は，**図 3** のように個々の要素に対する処理を記述することで指定する．

3.4 耐故障性

多くの計算機を利用する場合，システム全体で見るといずれかの計算機が頻繁に故障することになる．そこで，本システムではファイルやデータベースの不揮発データのバックアップを保持し，あるノードが故障しても他のノードがそのデータを引き継いで計算を継

```
class Subtractor extends Join {
  boolean equals(Account account, Customer customer) {
    return account.accountKey == customer.accountKey;
  }
  void join(Account account, Customer customer, Output out){
    out.collect(new NewBal(account.accountKey,
                           account.balance - input.payment);
  }
}
```

図 3 プリミティブの実装例
Fig. 3 Sample implementation of primitive.

続できるようにしている．これは，バックアップ用のノードを専用に用意するのではなく，各ノードがそれぞれ他のノードのバックアップを保持するような構成も可能である．

あるトランザクションの途中で障害が発生した場合には，バックアップを保持するノードはそれまでに受信した更新をロールバックし，トランザクションの最初から処理を再開する．

4. 言語仕様

本章では，本システムにおける言語仕様について説明する．

4.1 分散データ

本システムでは，計算ノードに分散するデータの集合を扱う．これには，データベースのテーブルやファイルの不揮発データ領域を表す PartitionedDataStore と，メモリ上に読み込まれた揮発データを表す PartitionedSet とがある．

4.1.1 不揮発データ

本システムが扱う不揮発データは，計算ノードに分散してデータベースまたはファイルに保持する．これらの分散した不揮発データ領域を表すために PartitionedDataStore（PD）を利用する．PD は，データが存在する場所や分割方法，およびそのデータへのアクセス方法を指定する．データのアクセスには，PD がデータベースのテーブルを表す場合には SQL が用いられ，ファイルを表す場合には Java の直列化機構が用いられる．

4.1.2 揮発データ

PD から読み込まれて計算に利用される揮発データの集合を表す概念として，粒度の異なる以下のようなものがある．これらの関係を**図 4** に示す．

- Set
  複数の計算ノードにまたがるデータの集合全体を表す．

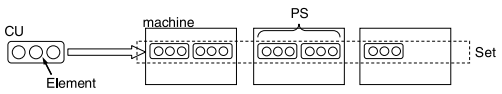

図 4 揮発データの関係
Fig. 4 Relationship between volatile data.

- PartitionedSet（PS）
  各計算ノードが保持するデータの集合を表す．
- Computation Unit（CU）
  ユーザが指定した，同一の計算ノード上で実行するべき範囲を表す．これは，Partitioner を用いてユーザが指定する．関連性の強いデータや，データベースの同一のフィールドにアクセスする必要があるデータなどは同一の計算ノード上で実行したほうが効率的であるため，この CU を用いてこのようなデータによる分割を指定する．
- chunk
  システムが一度に処理するデータの範囲を表す．CU が物理メモリに収まらないほど大きい場合，システムは CU を複数の chunk に分割し，chunk 単位での実行を行う．これは実装上の単位であり，ユーザが意識する必要はない．
- element
  データの最小単位．map などの並列プリミティブにはこの element 単位で渡される．計算は element 単位で並列実行されるので，ユーザは並列実行可能な最小単位を element として記述する必要がある．

4.2 トランザクション

プログラム中の，並列計算をする箇所の構造は以下のようになっている．

```
PartitionedDataStore pd1;
PartitionedSet ps1;
Partitioner partitioner1, partitioner2;
task {
   subtxn(ps1, pd1, partitioner1) {
      // ...
   }
   subtxn(ps1, partitioner2) {
      // ...
   }
}
```

1 つの `subtxn` からなるタスクでは，各計算ノードは chunk ごとに読み込み → 計算 → コミットを繰り返し実行する．読み込んだデータは Partitioner によって分散される．Partitioner を指定しなかった場合，データの再分散は行われず，各計算ノードはローカルのデータを読み込む．また，読み込んだデータ全

体が 1 つの CU として扱われる．

タスクの途中でデータの分散を変更する場合，複数の subtxn を利用する必要がある．2 つめ以降の subtxn では PS と Partitioner を指定する．PS 中のデータは Partitioner が指定する計算ノードに送信され，subtxn 内の計算に利用される．2 つめ以降の subtxn の CU は，直前の subtxn の CU を，指定された Partitioner を用いてさらに細かく分割したものになる．トランザクションは subtxn ごとに作成され，タスクの最後で二相コミットによってコミットされる．

4.3 並列プリミティブ

プログラマは，Task 内で，PS や PD を引数とする並列プリミティブの呼び出しを記述する．また，並列プリミティブ内での，個々のデータに対して行うべき処理を記述する．

システムは，並列プリミティブの呼び出しで指定された PS や PD に関して自ノードが保持する各データについて，ユーザが指定した処理の実行を行う．この処理は，データを保持するすべての計算ノード上で，自動的に並列化して行う．これにより，並列化のための詳細な処理をプログラマから隠蔽することができる．

4.3.1 Read

Read プリミティブは不揮発データを，Task 内の計算のために揮発データに移動する．

Read プリミティブの呼び出しインタフェースは，以下のようになっている．

```
public void read(PartitionedSet indexps,
                 PartitionedDataStore pd,
                 PartitionedSet outputps)
```

このメソッドを呼び出すと，indexps で指定された PS のデータをキーとしてデータを pd から読み取り，結果を outputps に格納する．

また，Read プリミティブの挙動を指定するために，ユーザは以下のメソッドを実装する．

- Reader クラスの抽象メソッドである read と select
- データベースのテーブルを表す PD からの読み込みの場合，DBSelectAccessor インタフェースで定義されている getSingleFinderString, setFindParameter
- ファイルを表す PD からの読み込みの場合，FileSelectAccessor インタフェースで定義されている selectObjectSingle

read メソッドは indexps に渡された PS のデータからキーとして利用するオブジェクトを作成するために，select オブジェクトは PD から読み込んだデータから outputps に出力するオブジェクトを作成するために，それぞれ呼び出される．また，DBSelectAccessor インタフェースと FileSelectAccessor インタフェースで定義されているメソッドは，PD からデータを読み込む具体的な方法を決めるためのものである．テーブルを表す PD からの読み込みの場合は SQL を利用し，ファイルを表す PD からの読み込みの場合は Writer プリミティブで保存した Java オブジェクトを直列化復元して読み込む．

```
public class WarehouseReader extends Reader {
  public void select (Object in, OutputCollector out)
                                        throws SQLException {
    ResultSet val = (ResultSet)in;
    out.collect (new WarehouseInfo(
        val.getShort("wid"), val.getFloat("wTax")));
  }
  public void read (Serializable value,
                    OutputCollector out) {
    out.collect (value);
  }
}
public class WarehouseDataSet
               extends PartitionedDataStore
               implements DBSelectAccessor, Serializable {
  public String getSingleFinderString () {
    return "select wid, wTax from warehouse d" +
                                     " where w.wid = ?";
  }
  public void setFindParameters (PreparedStatement s,
            Object obj, int offset) throws SQLException {
    s.setShort(offset+1, ((InputFormat)obj).warehouseID);
  }
}
```

図 5 Read プリミティブの実装例
Fig. 5 Sample implementation of read primitive.

Read プリミティブの実装例を**図 5** に示す．これは，SQL を利用して入力 PS によって指定されるキーのフィールドを読み込む例である．

4.3.2 Write

Write プリミティブは Task 内で計算された揮発データを，別の Task に結果を渡すために不揮発データ領域に移動する．

Write プリミティブの呼び出し用のシグニチャは以下のようになっている．

```
public void write(PartitionedSet inputps,
                  PartitionedDataStore pd)
```

inputps は出力するデータを保持する PS，pd は出力先の PD である．

Writer の動作には，Insert，Update，Delete の 3 種類がある．どの処理を行うかは，Writer オブジェクトのコンストラクタ引数で指定する．

それぞれの処理の具体的な方法は，プログラマが PD の子クラスで該当するメソッドを実装することにより

```
public class DistrictUpdateWriter extends Writer {
  public void write (Serializable value,
        OutputCollector out) throws SkeletonException {
    out.collect(value);
  }
}
public class DistrictPartitionedData
               extends PartitionedDataStore
               implements DBUpdateAccessor, Serializable {
  public String getUpdateString () {
    return "update district set dnxtor = ?" +
                        " where (dwid = ? and did = ?)";
  }
  public void setUpdateParameters (PreparedStatement s,
                         Object obj) throws SQLException {
    WarehouseDistrictInfo val = (WarehouseDistrictInfo)obj;
    s.setInt (1, val.districtNextOrder);
    s.setShort (2, val.warehouseID);
    s.setShort (3, val.districtID);
  }
}
```

図 6 Write プリミティブの実装例
Fig. 6 Sample implementation of write primitive.

指定する．Insert，Update では，PD がデータベースのテーブルを表す場合は SQL を利用して，ファイルを表す場合はオブジェクトをシリアライズして，それぞれ出力する．また，Update，Delete では，更新・削除対象のデータかどうかを判別するためのメソッドを実装する．

また，Writer オブジェクトでは，PS 中のデータを PD に渡す形式に変換することができる．

Write プリミティブの実装例を**図 6** に示す．この例では，入力に渡された PS の値を用いて，同じ ID を持つレコードを更新している．

4.3.3 Map

PS に格納されている個々のデータに対する処理を行うために使用する．Map プリミティブの呼び出しインタフェースは以下のようになっている．

```
public void map(PartitionedSet sourceps,
                PartitionedSet targetps)
```

`sourceps` に格納されている各データに対して，ユーザが記述したメソッドを繰り返し呼び出し，その結果を `targetps` に格納する．

Map プリミティブの実装例を**図 7** に示す．この例では，各入力データに対して，対応する `DeliveryInfo` オブジェクトを生成している．

4.3.4 Reduce

1 つの PS に格納されている複数のデータにまたがる処理を行うために利用する．Reduce プリミティブの呼び出しインタフェースは以下のようになっている．

```
public void reduce(PartitionedSet sourceps,
                   PartitionedSet targetps)
```

```
public class GenerateDeliveryInfoMapper
             extends Mapper<OutputFormat, Serializable>  {
  public void map (OutputFormat val,
                   OutputCollector<Serializable> out) {
    DeliveryInfo o = new DeliveryInfo();
    o.setDataFromOrderInfo(val);
    o.deliveryDate = getCurrentDateAsInt();
    o.deliveryTime = getCurrentTimeAsInt();
    o.carrierID = random.nextInt(10);
    out.collect(o);
  }
}
```

図 7 Map プリミティブの実装例
Fig. 7 Sample implementation of map primitive.

```
public class ComputeTotalAmountReducer
                              extends Reducer<OrderInfo> {
  public void reduce (Iterator itr, OutputCollector out)
                              throws SkeletonException {
    OrderInfo o = new OrderInfo();
    while (itr.hasNext()) {
      o.amount += ((OrderInfo)itr.next()).amount;
    }
    out.collect(o);
  }
}
```

図 8 Reduce プリミティブの実装例
Fig. 8 Sample implementation of reduce primitive.

`sourceps` 内のデータを返す Iterator を引数に渡してユーザが実装したメソッドを呼び出し，その結果を `targetps` に格納する．

Reduce プリミティブの実装例を**図 8** に示す．この例では，売上げの合計を計算している．

Reduce プリミティブでは，CU 単位で処理を行う．図 8 の例は，Partitioner によって入力の PS が何らかの単位（たとえば店舗など）で CU に分割されていることを仮定しており，この単位ごとの合計を計算するものである．

4.3.5 Join

2 つの PS に格納されている，対応する項目についての処理を行うために利用する．Join プリミティブの呼び出しインタフェースは以下のようになっている．

```
public void join(PartitionedSet sourceps1,
                 PartitionedSet sourceps2,
                 PartitionedSet targetps)
```

入力に指定された 2 つの PS の，現在処理中の chunk 内にあるデータのうち，対応するものについてユーザが定義したメソッドを呼び出し，その結果を出力の PS に格納する．入力の 2 つの PS の間で，どのデータが対応するかは，ユーザが Joiner に記述したメソッド `equals` によって決定する．

3.3 節の図 3 で紹介したプリミティブの実装例は，

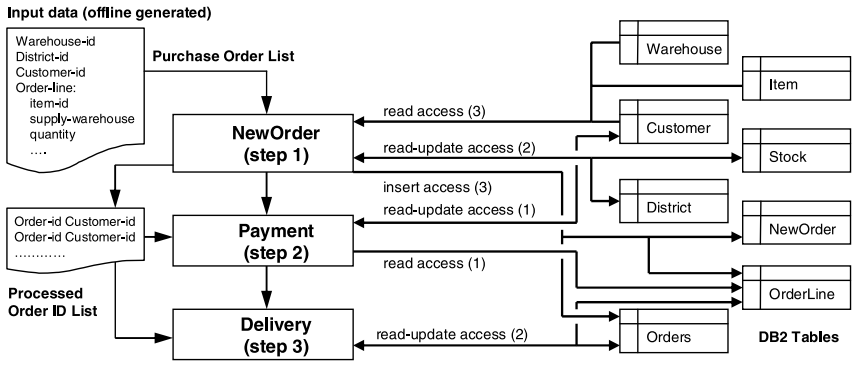

図 9 TPC-C の処理の流れ
Fig. 9 Execution flow of TPC-C.

この Join プリミティブの例である．この例では，顧客の口座情報から利用料金を引き落とし，その残高を結果として出力している．

5. 実行環境

5.1 実行方法

プログラマが記述したプログラムは，コンパイラを用いてマスタ用のコードとワーカ用のコードにコンパイルされる．マスタ用のコードでは，タスク以外の部分は通常の Java プログラムとしてコンパイルされる．この部分には，任意の Java コードを記述することができる．タスクの入り口では，マスタはワーカに該当タスクの実行を指示する．ワーカがタスクの処理を終了すると，マスタに通知する．マスタが全ワーカからタスクの終了の通知を受け取ると，マスタは後続の処理を続ける．

ワーカ用のコードは SPMD（Single Program Multiple Data）になっており，すべてのワーカで同じコードが実行される．ワーカ用のコードにはタスク内の処理のみが含まれており，マスタからの指示によって実行される．

ワーカ用のコードは，あらかじめ各計算ノード上に配置する必要がある．これには，たとえばネットワークファイルシステムを利用することができる．

5.2 耐故障性

本システムでは，ワーカが保持するデータは PS と PD の 2 種類しかない．PD はデータベース・ファイルともにトランザクション処理をサポートしており，PS はタスク内でしか保持されない．そのため，システムは自由にタスク単位での再実行を行うことができる．

そこで本システムでは，耐故障性を高めるために，PD への書き込みを他のマシンに複製し，ワーカの障害発生時には該当ワーカの複製を保持しているマシンが，障害発生時に実行していたタスクから実行を再開する機能を備えている．

6. 実験

TPC-C ベンチマーク[2] を基にしたバッチアプリケーションを本システムを用いて作成し，その性能を測定した．

6.1 TPC-C

TPC-C は，物流業における受発注処理を行うオンライントランザクションシステムのベンチマークであるが，このうち NewOrder，Payment，Delivary の 3 つの処理について，ファイルに保存された入力データを処理するバッチアプリケーションを作成した．ソースコードのうち，NewOrder の主要部分を付録に載せた．

処理の流れと，各処理でアクセスするテーブルを**図 9** に示す．Item テーブルのデータは全ノードにコピーされており，それ以外のテーブルはすべて Warehouse の ID によって各ノードに分散して保持している．

NewOrder では，入力の受注情報を Warehouse ID によって分散させてから受注処理を行う．注文を受けた Warehouse に在庫がない場合は他の Warehouse で処理を行う必要があり，通信が発生する．Payment と Delivery では NewOrder の結果を入力とし，計算はすべてローカルで行われる．

6.2 環境

実験には IBM BladeCenter JS21 を用いた．各ブレードの構成は PowerPC 970MP 2.5 GHz（dual-core）×2，8 GB Memory，10,000 rpm SAS HDD で，OS は Linux 2.6 を利用した．各ブレードは 1 Gb Ethernet によって接続している．Java VM には IBM Developer Kit，Java Technology Edition，Version 1.5.0 を利用した．

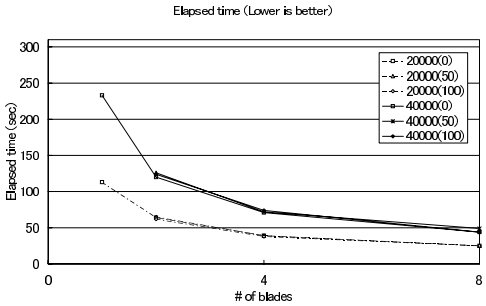

図 10 実行時間
Fig. 10 Execution time.

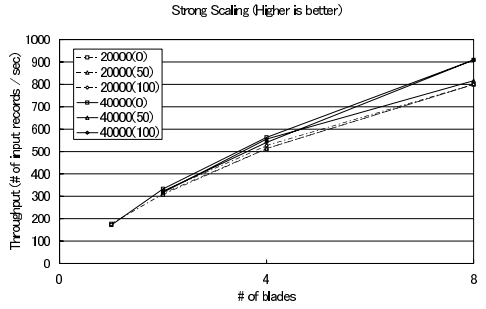

図 11 スループット
Fig. 11 Throughput.

表 1 スループット（record/sec）
Table 1 Throughput (record/sec).

		ブレードの台数			
レコード数	remote ratio	1	2	4	8
20,000	0%	177	313	513	800
20,000	50%	-	308	513	800
20,000	100%	-	323	526	800
40,000	0%	172	333	563	909
40,000	50%	-	317	556	816
40,000	100%	-	323	541	909

なお，我々が作成したシステムでは耐故障性のための機能もあるが，この実験の目的は本論文で紹介した言語で記述したプログラムの性能を測定することであるため，測定は耐故障性を無効にした状態で行っている．

6.3 結　　果

ワーカの数を 8 台まで変化させて性能を測定した結果を**図 10** と**図 11**，および**表 1** に示す．入力のデータは，レコード数が 20,000 と 40,000 のそれぞれの場合に対して，NewOrder において通信が必要になる入力の割合（remote ratio）を 0%，50%，100%と変えて測定した．

1 台で実行した場合と比較して，4 台の場合で最大約 3.3 倍，8 台の場合で最大約 5.3 倍の性能となった．remote ratio を変えても性能はほとんど変わらなかった．

このアプリケーションではデータベースに書き込むデータ量が多いためディスク IO がボトルネックとなる．データベースに書き込むデータに比べ，ノード間を転送されるデータは少ないため，通信時間は大きな影響を与えない．本システムでは，データベースへのアクセスは自ノードに閉じているため，データアクセスの競合による性能低下が発生しない．したがって，通信が必要な場合でも，性能の低下が見られない，と考えられる．

なお，この実験では 1 台のワーカで 4 つの CPU を利用しているが，このアプリケーションはディスク IO がボトルネックとなっているため利用する CPU の数を減らしても性能はほとんど変化しない．

スケーラビリティを阻害している要因としては，タスクの最後での同期がある．タスクの最後ではトランザクションをコミットするために同期をとる必要があるが，このときにタスクの処理に時間がかかった一部のワーカを他のワーカが長時間待たされる，という状況が発生していた．このアプリケーションは多数のタスクからなっているため，この同期による性能低下の影響が特に大きく出たものと考えられる．

7. ま と め

データベースを利用する並列アプリケーションを作成するために我々が開発したシステムについて，その設計と実行環境について説明した．本システムではデータはユーザが指定した方法により分散して保持され，Owner-Computes Rule によって実行される．これにより通信やデータアクセスの競合を最小限にし，スケーラビリティの高いプログラムを作成することができる．トランザクションのサポートにより，複数のノードにまたがる処理においてもデータの整合性を保障している．プログラマは Java を拡張した言語を用いて，通信や同期処理などの並行処理の詳細について考慮することなく，高い抽象度でプログラムを記述することができる．

参 考 文 献

1) http://lucene.apache.org/hadoop/
2) http://www.tpc.org/tpcc/
3) IBM corporation: WebSphere DataStage. http://ibm.com/software/data/integration/datastage/
4) Cole, M.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation, *Research Monographs in Parallel*

and Distributed Computing, Pitman, London (1989).
5) Cornell, D.W. and Yu, P.S.: On optimal site assignment for relations in the distributed database environment, *IEEE Trans. Softw. Eng.*, Vol.15, No.8, pp.1004–1009 (1989).
6) Dean, J. and Ghemawat, S.: MapReduce: Simplified dataprocessing on large clusters, *6th Symposium on Operating System Design and Implementation* (*OSDI*), pp.137–150 (2004).
7) Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Compiling Fortran D for MIMD distributed-memory machines, *Comm. ACM*, Vol.35, No.8, pp.66–80 (1992).
8) Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks, *Annual European Conference on Computer Systems* (*EuroSys*) (2007).
9) Johansson, J.M., March, S.T. and Naumann, J.D.: The effect of parallel processing on update response time in distributed database design, *Proc. 21st international conference on Information systems*, pp.187–196 (2000).
10) Kossmann, D., Franklin, M.J., Drasch, G. and Ag, W.: Cache investment: Integrating query optimization and distributed data placement, *ACM Trans. Database Syst.*, Vol.25, No.4, pp.517–558 (2000).
11) Ozsu, M.T. and Valduriez, P.: *Principles of Distributed Database Systems*, 2nd Edition, Prentice Hall (1999).
12) Rabhi, F. and Gorlatch, S.: *Patterns and Skeletons for Parallel and Distributed Computing*, Springer-Verlag (2002).
13) Soltis, S.R., Ruwart, T.M. and O'Keefe, M.T.: The Global File System, *Proc. 5th NASA Goddard Conference on Mass Storage System*, pp.319–342 (1996).
14) Zima, H.P., Bast, H.-J. and Gerndt, M.: SUPERB: A tool for semi-automatic MIMD /SIMD parallelization, *Parallel Computing*, No.6, pp.1–18 (1988).

付 録

A.1 NewOrderDelivery.mgn

```
package com.ibm.cso.mugen.demo.tpccbatch;

import java.io.Serializable;
import java.sql.SQLException;
import java.util.ArrayList;
import com.ibm.cso.mugen.demo.tpccbatch.setup.CreateInputData;
import com.ibm.cso.mugen.runtime.*;
import com.ibm.cso.mugen.runtime.common.*;
import com.ibm.cso.mugen.runtime.exception.SkeletonException;
import com.ibm.cso.mugen.datapartition.*;
import com.ibm.cso.mugen.txfile.*;
import com.ibm.cso.mugen.runtime.worker.OutputCollector;

public class NewOrderDelivery {
  private void executeNewOrder (Job _job) {
    String persistenDB = "db2://trltpcc:trltpcc@host";
    String volatileDB = "txfile://";
    String[] wTablePDuris = {persistenDB + "/TPCC/Warehouse",
                             persistenDB + "/TPCC/District"};
    PartitionedDataStore wTablePD =
                    new WarehousePartitionedData(wTablePDuris);
    PartitionedDataStore inputPD = new InputFileData
                                  ("file://cso/tpcc/input.txt");
    //-----------------------------------------------------------------
    // 1st task: customer order data extraction and redistribute
    //-----------------------------------------------------------------
    PartitionedDataStore dTablePD = new DistrictPartitionedData
                         (persistenDB + "/TPCC/District");
    PartitionedDataStore cTablePD = new CustomerPartitionedData
                         (persistenDB + "/TPCC/Customer");
    String[] sTablePDuris = {persistenDB + "/TPCC/Item",
                             persistenDB + "/TPCC/Stock"} ;
    PartitionedDataStore sTablePD =
                    new StockPartitionedData (sTablePDuris);
    PartitionedDataStore oTablePD = new OrdersPartitionedData
                        (persistenDB + "/TPCC/OrderLine");
    PartitionedDataStore nTablePD = new NeworderPartitionedData
                        (persistenDB + "/TPCC/NewOrder");
    PartitionedDataStore orderSpWhPD = new MemoryPartitionedDataStore
              (volatileDB + "/tpccmem/remote_order",
              OrderInfoSupplyWarehousePartitioner.class, pworkers);
    PartitionedDataStore cTempPD = new MemoryPartitionedDataStore
              (volatileDB + "/tpccmem/customert",
               CustomerInfoWarehousePartitioner.class, pworkers);

    /* for restoring input from memoryPD (after redistributed) to PS */
    PartitionedSet inputPS = new PartitionedSet (inputPD);
    /* for accessing to Warehouse and District tables */
    Reader warehouseDistrictTable = new WarehouseDistrictReader ();
    PartitionedSet wdTempPS = new PartitionedSet (inputPD);
    /* for accessing to Customer table */
    Reader customerTable = new CustomerInputReader ();
    PartitionedSet cTempPS = new PartitionedSet (inputPD);

    Joiner wDCTable = new WDCJoiner
                (new WarehouseDistrictInfo(), new CustomerInfo());
    Writer districtTable = new DistrictUpdateWriter ();
    /* for creating remote order list for each customer */
    Joiner createOrder = new CreateOrderJoiner
                  (new InputFormat(), new CustomerInfo());
    PartitionedSet orderPS = new PartitionedSet (inputPD);
    /* for storing cTempPS into memoryPD fir reuse in later tasks */
    Writer memWriter = new MemoryPDWriter ();

    task {
      subtxn(inputPD, inputPS, new InputDataWarehousePartitioner()) {
        /* extract records from Warehouse and District tables */
        warehouseDistrictTable.read (inputPS, wTablePD, wdTempPS);

        /* extract records from Customer table */
        customerTable.read (inputPS, cTablePD, cTempPS);

        /* combine the two PartitionedSet */
        wDCTable.join (wdTempPS, cTempPS, null);

        /* update District table */
        districtTable.write (wdTempPS, dTablePD);

        /* create order list (mixing both local and remote)
           from all customers */
        createOrder.join (inputPS, cTempPS, orderPS);

        /* store customer data set into memoryPD
           for reuse in the later tasks */
        memWriter.write (cTempPS, cTempPD);
        memWriter.write(orderPS, orderSpWhPD);
      }
    } // end of 1st task

    //-----------------------------------------------------------------
    // 2nd task: process orders at supply warehouse
    //-----------------------------------------------------------------
    PartitionedDataStore orderLoWhPD = new MemoryPartitionedDataStore
              (volatileDB + "/tpccmem/local_order",
               OrderInfoWarehousePartitioner.class, pworkers);
    /* for restoring orders from memoryPD (after redistributed) to PS */
    PartitionedSet orderSpWhPS = new PartitionedSet (orderSpWhPD);
    /* for processing orders for each customer with local stock */
    Joiner processOrder = new ProcessOrderJoiner
```

```
                    (new OrderInfo(), new ItemStockInfo());
    Writer stockTableWriter = new StockUpdateWriter ();
    Reader itemStockTable = new ItemStockReader ();
    PartitionedSet sTempPS = new PartitionedSet (cTempPS);

    task {
      subtxn(orderSpWhPD, orderSpWhPS, new OrderInfoItemPartitioner()) {
        /* accessing Item and Stock tables separately */
        itemStockTable.read (orderSpWhPS, sTablePD, sTempPS);
        /* process orders in supply warehouse */
        processOrder.join (orderSpWhPS, sTempPS, null);
        /* update Stock table */
        stockTableWriter.write (sTempPS, sTablePD);
        memWriter.write(orderSpWhPS, orderLoWhPD);
      }
    } // end of 2nd task

    //-------------------------------------------------------------------
    // 3rd task: final registration of customer orders
    //-------------------------------------------------------------------

    PartitionedDataStore olTablePD = new OrderlinePartitionedData
                         (persistenDB + "/TPCC/OrderLine");
    PartitionedDataStore newOrderFilePD = new NewOrderOutputFileData
              (volatileDB + "/tpccmem/neworderoutput",
               NeworderPartitioner.class, pworkers);
    oTablePD = new OrdersPartitionedData
                  (persistenDB + "/TPCC/Orders");
    /* for restoring orders from memoryPD (after redistributed) to PS */
    PartitionedSet orderLoWhPS = new PartitionedSet (orderLoWhPD);
    /* for inserting order information into tables */
    Writer orderLineTable = new OrderLineInsertWriter (Writer.INSERT);
    Writer ordersTable = new OrdersInsertWriter (Writer.INSERT);
    /* output file of NewOrder */
    Writer neworderOutputWriter = new MemoryPDWriter();

    task {
      subtxn(orderLoWhPD, ordrLoWhPS, new OrderInfoItemPartitioner()) {
        /* insert OrderLine table */
        orderLineTable.write (orderLoWhPS, olTablePD);
      }
      subtxn(cTempPD, cTempPS, new CustomerInfoWarehousePartitioner()) {
        ordersTable.write (cTempPS, oTablePD);
        neworderOutputWriter.write(cTempPS, newOrderFilePD);
      }
    } // end of 3rd task
  }
}
```

A.2 DistrictPartitionedData.java

```
package com.ibm.cso.mugen.demo.tpccbatch;

import com.ibm.cso.mugen.datapartition.*;
import com.ibm.cso.mugen.datapartition.DBUpdateAccessor;
import com.ibm.cso.mugen.datapartition.Partitioner;
import com.ibm.cso.mugen.datapartition.Processors;
import java.io.Serializable;
import java.sql.*;

public class DistrictPartitionedData extends PartitionedDataStore
                        implements DBUpdateAccessor, Serializable {
  String finder1 = "update ",
         finder2 = " set dnxtor = ? where (dwid = ? and did = ?)";
  private String pstmt = null;

  public DistrictPartitionedData () { super(); }
  public DistrictPartitionedData (String uri) { super(uri); }
  public DistrictPartitionedData (String uri,
          Class<? extends Partitioner> partitioner, Processors p) {
    super(uri, partitioner, p);
  }

  public String getUpdateString () {
    if (pstmt == null)
      pstmt = finder1 + getTableName() + finder2;
    return pstmt;
  }

  public void setUpdateParameters (PreparedStatement s, Object obj)
                                              throws SQLException {
    WarehouseDistrictInfo val = (WarehouseDistrictInfo)obj;
    s.setInt (1, val.districtNextOrder);
    s.setShort (2, val.warehouseID);
    s.setShort (3, val.districtID);
  }
}
```

A.3 CustomerPartitionedData.java

```
package com.ibm.cso.mugen.demo.tpccbatch;

import java.io.Serializable;
import java.sql.*;
import java.util.*;
import com.ibm.cso.mugen.datapartition.*;

public class CustomerPartitionedData extends PartitionedDataStore
      implements DBSelectAccessor, DBUpdateAccessor, Serializable {
  private String finder = "select * from ";
  private String singlekey = "(cwid = ? and cdid = ? and cid = ?)";
  private String rangekey =
       "(cwid = ? and cdid = ? and cid >= ? and cid <= ?)";
  private String update1 =
       "update ", update2 = " set cbal = ?, cdelcnt = cdelcnt + 1";
  private String where = " where ";

  private String singlefinder_stmt = null;
  private HashMap multifinder_stmt_hash = new HashMap();
  private String rangefinder_stmt = null;
  private String update_stmt = null;

  public CustomerPartitionedData () { super(); }
  public CustomerPartitionedData (String uri) { super(uri); }
  public CustomerPartitionedData (String uri,
          Class<? extends Partitioner> partitioner, Processors p) {
    super(uri, partitioner, p);
  }

  public String getSingleFinderString () {
    if (singlefinder_stmt == null)
      singlefinder_stmt = finder + getTableName() + where + singlekey;
    return singlefinder_stmt;
  }
  public void setFindParameters (PreparedStatement s, Object obj,
                                 int offset) throws SQLException {
    List val = (List)obj;
    s.setShort(offset+1, ((Integer)val.get(0)).shortValue());
    s.setShort(offset+2, ((Integer)val.get(1)).shortValue());
    s.setInt  (offset+3, ((Integer)val.get(2)).intValue());
  }
  public int getNumberOfFindParameters() { return 3; }

  public String getUpdateString () {
    if (update_stmt == null)
      update_stmt = update1+getTableName()+update2+where+singlekey;
    return update_stmt;
  }
  public void setUpdateParameters (PreparedStatement s, Object obj)
                                                  throws SQLException {
    CustomerInfo val = (CustomerInfo)obj;
    s.setFloat (1, val.totalAmount);
    s.setShort (2, val.warehouseID);
    s.setShort (3, val.districtID);
    s.setInt (4, val.customerID);
  }
}
```

A.4 NewOrderOutputFileData.java

```
package com.ibm.cso.mugen.demo.tpccbatch;

import java.io.*;
import java.sql.*;
import java.util.Iterator;
import com.ibm.cso.mugen.datapartition.*;
import com.ibm.cso.mugen.txfile.TxFile;

public class NewOrderOutputFileData extends PartitionedDataStore
       implements FileSelectAccessor, FileInsertAccessor,
       Serializable {
  short warehouseID, districtID;
  int customerID, orderID;
  float wTax, dTax;

  public NewOrderOutputFileData () { super(); }
  public NewOrderOutputFileData (String uri) { super(uri); }
  public NewOrderOutputFileData (String uri,
          Class<? extends Partitioner> partitioner, Processors p) {
    super(uri, partitioner, p);
  }

  public OutputFormat selectObjectSingle
           (TxFile f, Serializable obj, Object paramObj) {
    return (OutputFormat)obj;
  }
```

```
  public OutputFormat insertObjectSingle (TxFile f, Object paramObj) {
    CustomerInfo rin = (CustomerInfo)paramObj;
    return new OutputFormat(rin.warehouseID, rin.districtID,
        rin.customerID, rin.ordNum, rin.warehouseTax, rin.districtTax);
  }

  public String getInsertString() {
    return ins1 + getTableName() + ins2;
  }
  public void setInsertParameters(PreparedStatement s, Object o)
                                                  throws SQLException {
    OutputFormat out = insertObjectSingle(null, o);

    s.setInt(1, 0);

    ByteArrayOutputStream bas = new ByteArrayOutputStream();
    try {
      ObjectOutput oos = new ObjectOutputStream(bas);
      oos.writeObject(out);
      oos.flush();
      oos.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    byte[] buf = bas.toByteArray();
    ByteArrayInputStream bis = new ByteArrayInputStream(buf);
    s.setBinaryStream(2, bis, (int)buf.length);
  }
}
```

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 22 日採録)

水野　謙（正会員）

1981 年生．2006 年早稲田大学大学院理工学研究科情報・ネットワーク専攻修士課程修了．同年日本 IBM（株）東京基礎研究所に入社．並列処理に関する研究に従事．日本ソフトウェア科学会会員．

菅沼　俊夫（正会員）

1982 年京都大学大学院工学研究科数理工学専攻修了．1992 年ハーバード大学大学院計算機学科修了．同年日本 IBM（株）東京基礎研究所に入社．HPF コンパイラ，Java JIT コンパイラの最適化に関する研究等に従事．現在，同研究所主管研究員．ACM 会員．

石崎　一明（正会員）

1992 年早稲田大学大学院理工学研究科修士課程修了．同年日本 IBM（株）入社．以来，東京基礎研究所において，並列処理，最適化コンパイラに関する研究開発に従事．現在は，サーバ上のワークロードの高速化に興味を持つ．博士（情報科学）．

古関　聰（正会員）

1969 年生．1998 年早稲田大学大学院理工学研究科電気工学専攻博士課程修了．同年日本 IBM（株）入社．以来，同社東京基礎研究所において，Java Just-in-Time コンパイラの開発に従事．工学博士．ACM 会員．

上田　陽平

1977 年生．2000 年東京大学理学部情報科学科卒業．2002 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同年日本 IBM（株）に入社．現在同社東京基礎研究所に勤務．並列・分散プログラミングの研究に従事．

小松　秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM（株）東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．