「マルチメディア，分散，協調とモバイル
(DICOMO2018)シンポジウム」 平成30年7月

シミュレーションを用いたヘテロジニアス分散アプリケーション設計手法の検討

郡浦 宏明[1] 中田 侑[1] 杉本 健[1] 木下 雅文[1]

IoT データ活用における処理スループット向上と高拡張性実現のため，分散実行基盤が広く用いられている．既存の分析アプリケーションを分散実行基盤に実装する過程で，複数種類の分散処理が組み合わさる状態(ヘテロジニアス分散アプリケーション)であるとき，時間ごとのリソース使用状況や外部サービスアクセス頻度について予測が難しく，ボトルネック検証や再設計の工数が増加している．これを可視化し設計にフィードバックするため，シミュレーションベースのヘテロジニアス分散アプリケーション設計手法を構想し，本手法を実現するための分散処理シミュレータを試作した．試作版を用いて複数の処理フローでリクエスト処理完了時間を評価し，提案手法における分散処理シミュレータの妥当性と有効性を示した．

Simulation-based design method for heterogeneous distributed applications

HIROAKI KONOURA[1] YU NAKATA[1]
KEN SUGIMOTO[1] MASAFUMI KINOSHITA[1]

1. はじめに

IoT 技術が急速に普及する中，様々な組織や個人間で，データを分析して価値に繋げるデータ活用が活発化している．データ活用では多くの場合，最初に入手済データやオープンデータのクレンジング・分析から開始し，データ傾向把握や価値化の目処を立てた上で，徐々に分析結果を用いたアプリケーション検討や，可視化システム化等の高次元検討に移行していく[1-3]．このとき，一般的に，初期過程では小規模なシステムや限定されたデータを対象にモノリシックなアプリケーションで分析施行し，その後価値を検証できた段階で，大規模データを対象にマシンリソースを活用してデータ増加時の分析や運用面の検証に移行する．

データ活用試行過程では，Java†, Python¶ 等の言語で分析・可視化ロジックを組んで価値検証したあとに，高速化・高拡張性の観点で分散設計対応が必要となるケースが多く見られる．ここで，元々モノリシックな分析アプリケーションを分散させるにあたり，分散処理可能な箇所が多岐にわたり，各々の分散処理内容が異なる場合がある．そのようなアプリケーションを分散実行する場合，Apache Hadoop [4,5]‡, Apache Spark [6] 等の分散実行基盤での実装が想定される．ここで，モノリシックな分析アプリケーションから，複数の異なる分散処理可能なところを取り出して分散実装したアプリケーションを，ヘテロジニアス分散アプリケーションと定義するとき，ヘテロジニアス分散アプリケーション実装には以下の課題がある．

・N 種類の処理が，それぞれ異なる処理量やディスクアクセス，リソース使用量の振る舞いを見せる．このとき，これらの組み合わせによって生じる事象を予想しにくい．

・一般的に分散アプリケーションにおいて，データベース(DB)は中間結果の格納および参照の手段として使用される．DB アクセスでは，通信オーバヘッドを抑えて高スループットを実現するため，1 度に複数のリクエストを束ねて送信(バッチ化)することが望ましい．しかし，DB 同時接続数やネットワーク帯域等も鑑みたとき，最適なバッチ化の単位や通信タイミングを求めるのは容易でない．

このような場合の設計方針として，分析ロジックにおける処理ボトルネックを特定して効果が高い箇所に限定して設計するか，あるいは設計・実機評価・特性把握・再設計といった PDCA サイクルを多く回す方針が考えられる．しかしヘテロジニアスな分散アプリケーションの場合，一部の処理変更が他の処理に対して，予想困難な影響を与える可能性がある．このような分析アプリケーションについて，多様かつ頻繁な改良および差替が想定される状況で，短期間で変化に追随するためには，個々のアプリケーションのロジックを精査してボトルネック特定し設計するときに要する工数や時間の削減が重要である．

そこで，本稿では，ヘテロジニアス分散アプリケーション設計を容易化し，工数や設計時間を短縮するための手法として，シミュレーションを用いたヘテロジニアス分散アプリケーション設計手法を検討する．本手法では，開発者がヘテロジニアス分散アプリケーションの設計例を作り，

1 (株) 日立製作所
† Java: Oracle Corporation の登録商標
¶ Python: Python Software Foundation の登録商標
‡ Apache, Hadoop: The Apache Software Foundation の登録商標

要件を定めたのち，それらをヘテロジニアス分散アプリケーション生成エンジンにかけることで，シミュレーション実行を経て，要件を満たす設計を導出することを想定する．本設計手法を用いることで，ヘテロジニアスな分散アプリケーションで発生する事象を提示して設計者の知見を高めるとともに，短い試行時間のなかでより最適な設計を採ることができる．

以下，2 章では一般的なアプリケーション分散実行基盤の需要と概要について説明する．3 章では，ヘテロジニアス分散アプリケーションの実装が必要な背景と，実装における課題を説明する．4 章では，ヘテロジニアス分散アプリケーション設計手法の構想と，その実現に必要となる分散処理シミュレータについて説明する．5 章では，分散処理シミュレータの有効性・妥当性を評価した結果を述べ，6 章で結論を述べる．

2. アプリケーション分散実行基盤の概要

1 章に示したように，データ活用試行において，小規模なシステムや限定されたデータを対象にデータ価値を検証できたあとで，大規模データを対象にマシンリソースを活用してデータ増加時の分析や運用面の検証に移行するケースがある．このとき，大規模なストリーミングデータおよびバッチデータを対象に，複数の物理サーバまたは仮想サーバのリソース(CPU ，メモリ，ディスク)を最大限に活用して処理高速化および柔軟なリソーススケーリングを実現するために，Apache Hadoop [4,5]，Apache Spark [6] をはじめとするアプリケーション分散実行基盤が広く用いられている．

Apache Hadoop や Apache Spark は，分散処理させたいデータおよび処理内容について，定められた記法に則って設計することで，分散処理を実現する．Apache Hadoop は，分析者が Key と Value の組み合わせを作る “Map” 処理と，その組み合わせを集約する “Reduce” 処理を開発することで，大量データ処理を実現する．Apache Spark は，RDD (Resilient Distributed Dataset) と呼ばれる「不変で並列実行可能なデータセット」でデータを保持し，用意されたメソッドを使ってデータのフィルタリングや連結，カウントなどの処理を行う．

3. ヘテロジニアス分散アプリケーション設計の課題

分析アプリケーションでは，予め定めた分析目的に応じて既存の分析ロジックを使い，所望の結果を得る．一例として，図 1 は Python の機械学習ライブラリ scikit-learn [7] における分析アルゴリズム選択のフローを示す．データ分析者は，このフローに示されるように，自分の持つデータや分析目的に応じて分析アルゴリズムを選択し，分析アプリケーションを開発する．

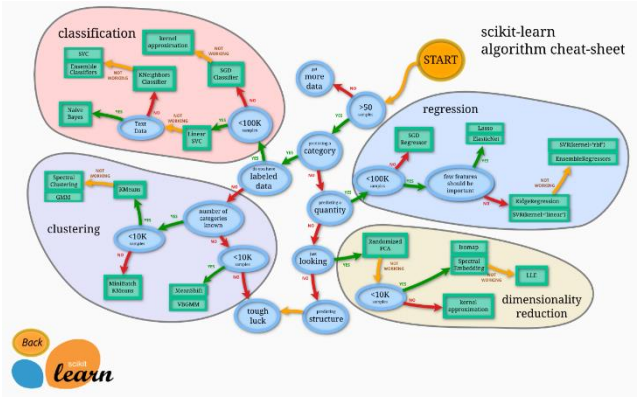

図 1 Python 機械学習ライブラリ (scikit-learn) チートシート [7]

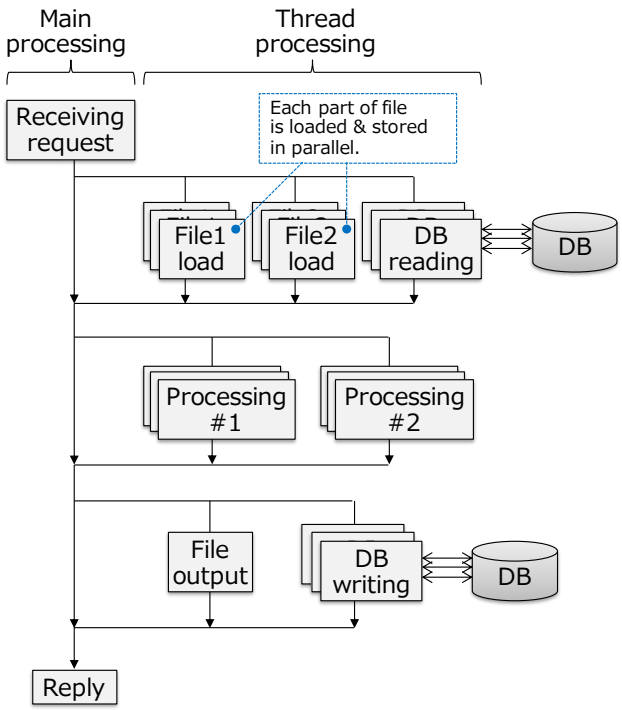

図 2 モノリシックアプリケーションを分散実行させるときのメイン処理およびスレッド処理の流れ

このとき，1 章で述べたように，初期段階ではデータ傾向把握や価値化の目処を立てる目的のため，小規模データを対象に，手元の PC やタブレットなどの端末で分析処理をかけ，その後価値検証できた段階で大規模データを対象とするようなスモールスタートのケースがよく見られる．このような手順を踏むとき，初期のアプリケーションは分散実行を想定していないモノリシックなアプリケーションである場合が多い．このモノリシックなアプリケーションを，高速化・高拡張性の観点で分散実装するとき，既存アプリケーションの内部には複数種類の並列実行できる箇所が存在する．参考として，図 2 にモノリシックアプリケーションの分散処理実装のイメージを示す．この図は，複数ファイルのそれぞれをさらに分解して取り込む処理や，取

り込んだデータに対する処理，ファイルや DB への出力処理が，それぞれ分散処理可能な対象であることを示している．このような異なる分散処理対象を含むアプリケーションについて，本稿ではヘテロジニアス分散アプリケーションと定義する．

1 種類の処理を分散実行する分散アプリケーションと，図 2 のように様々な分散処理が同時進行するヘテロジニアス分散アプリケーションの設計を比較したとき，以下のような違いが見られる．

・ヘテロジニアスな処理によるマシンリソース使用状況：

ヘテロジニアス分散アプリケーションでは，分散実行する N 種類の処理が，いずれかのノードに振り分けられ，それぞれ異なる CPU 使用量，メモリ使用量，ディスクアクセスの振る舞いをするなかで，その重ね合わせで各ノードのリソース使用状況が変化していく．このとき，各ノードでどの時間にどの処理が重なり合い，何が起こるかを予想しにくい．また，処理途中でスレッド使用数が極端に増加する事象が想定される．

・ヘテロジニアスな処理による外部サービスアクセス状況：

DB などの外部サービスにアクセスする際，通信オーバヘッドを抑えて高スループットを実現するため，1 度に複数のリクエストを束ねて送信(バッチリクエスト化)することが望ましい．しかし，外部サービスへの同時接続数やネットワーク帯域等を鑑みたとき，最適なバッチ化の単位や通信タイミングを求めるのは容易ではない．ヘテロジニアスな処理が外部サービスへのバースト的なアクセスを発生させ，結果的に外部サービスがボトルネックとなり処理速度が低下する事象が想定される．

本来，このように分散実行基盤上にヘテロジニアス分散アプリケーションを設計・実装する際には，既存のモノリシックアプリケーションにおいてボトルネックとなっている処理を特定し，その部分を中心に分散設計を採ることが望ましい．もしくは，設計・実機評価・特性把握・再設計というような PDCA サイクルを複数回実行することが望ましい．一方で，データ活用においては，

(1) 既存のモノリシックアプリケーションを開発した分析者と分散システムの設計者が異なり，システム設計者は分析者が開発したアプリケーションの内部仕様を把握していない状況で対応する，

(2) 分析アプリケーションの仕様が頻繁に変わる，

(3) 分析時間短縮のため少しでも高速化できたほうがよい，

などの複合的な状況が多く見られる．そのため，ヘテロジニアス分散アプリケーションの設計開発において，ボトルネック処理の特定や再設計にかける工数をできるだけ抑えて，高速化・スケーリング性などの要件に対して最大効果を得られるような設計手法が期待されている．

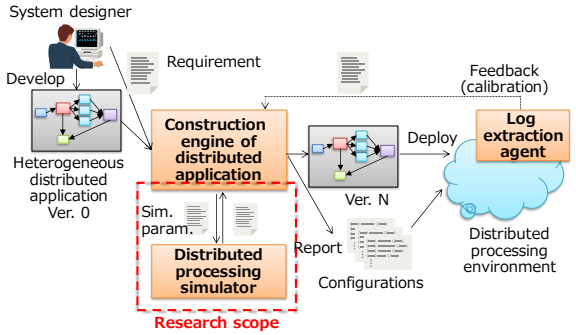

図 3 シミュレーションベースのヘテロジニアス分散アプリケーション設計手法の構想

4. シミュレーションベースのヘテロジニアス分散アプリケーション設計手法

4.1 概要

ヘテロジニアス分散アプリケーションを対象に，設計を容易化し，工数や設計時間を短縮するための手法として，図 3 に示すシミュレーションベースのヘテロジニアス分散アプリケーション設計手法を検討した．本設計手法では，開発者が初期バージョンのヘテロジニアス分散アプリケーションを設計し，その設計を要件定義とともにヘテロジニアス分散アプリケーション構築エンジンに入力する．このとき，初期バージョンのヘテロジニアス分散アプリケーションは，処理内容や処理フローが正しく記述されていることのみが必須条件であり，それらがどのノードで処理されるか，どの単位で処理をまとめるか，といった内容を指定する必要はない．分散アプリケーション構築エンジンは，まず初期バージョンのヘテロジニアス分散アプリケーションと要件定義から評価関数を作ったのち，分散処理シミュレータへの入力を与える．分散処理シミュレータは，与えられた設計やシミュレーションパラメータに従ってシミュレーションを実行し，分散アプリケーション構築エンジンに対して処理時間やリソース使用状況などの結果を返却する．分散アプリケーション構築エンジンは，それらの結果を評価関数に与えて，結果を評価する．この分散アプリケーション構築エンジンと分散シミュレータの間の試行を，焼きなまし法(Simulated Annealing)[8]や遺伝アルゴリズム(Genetic Algorithm)[9,10]などの大域的最適解を求めるアルゴリズムに従い，設計の一部やシミュレーションパラメータを変えながら繰り返すことで，要件を満たす設計を探索する．こうして得られた最終バージョンの設計を，分散実行基盤上に実装する．このとき，実機評価とシミュレーション評価の間での乖離を解消するため，ログ収集フィードバックを受けてシミュレーションパラメータを実機評価に近づくよう調節するキャリブレーション機能も想定する．

このような一連の設計手法を実現するために，分散処理シミュレータに求められる項目を以下に示す．

・再現性：分散実行基盤のアーキテクチャ構成を再現できる．

・実行結果透明性：評価関数に与えられるような結果を得ることができる．例として，全リクエスト処理完了時間(≒単位時間あたりのスループット), 1 リクエストあたりの平均レスポンス時間，ノードごとのリソース平均・最大使用量，閾値超過発生回数などが挙げられる．

・高速性：実機評価に比べて実行時間が高速である．

次節では，プロトタイプとして試作した分散シミュレータについて説明する．

4.2 分散処理シミュレータ

ヘテロジニアス分散アプリケーションの動作が再現可能な，分散処理シミュレータについて説明する．分散実行基盤において，各処理 (イベント) が並列に実行される様子を再現するために，Python の離散イベントシミュレータライブラリ Simpy [11] を用いて実装した．なお，離散イベントシミュレーションとは，主に待ち行列モデルの混雑現象を分析・評価するためのシステムであり，離散的に発生するシステムの状態変化を起こすトリガ(イベント)を捉えて処理の進行状況を追うために用いる．

図 4 は，プロトタイプの分散処理シミュレータ上で構築した仮想の分散処理基盤アーキテクチャおよびその構成要素を示す. ここで想定した分散処理基盤アーキテクチャは，主にアダプタ，ディスパッチャ，メッセージプロセッサからなる．アダプタはファイルや REST 通信，管理コンソール実行等を契機にメッセージを取り込んだのち，メッセージを送出(ディスパッチ) する．ディスパッチャは，アダプタからディスパッチされたデータをキューに格納し，カスタマイズ可能なルーティングストラテジと呼ばれるメッセージ振り分けロジックに従い，メッセージをメッセージプロセッサに送る．メッセージプロセッサは，設計者が開発したロジックに従ってメッセージを処理する．なお，ノードは，物理マシンまたは仮想マシンに相当し，それぞれのノードにアダプタと複数のメッセージプロセッサが展開されるほか，ディスパッチャは全てのノードにまたがり，ノード間のアクセスを担う．この構成を採ることで，アダプタに届いたイベントは異なるサーバのメッセージプロセッサに振り分けられ，それぞれのサーバのリソースを用いて並列分散処理される．

図 4 において，シミュレーションマネージャが，シミュレーション対象のコンポーネントを管理する．対象コンポーネントは，クライアント，アダプタ，ディスパッチャ(キュー，ルータ)，メッセージプロセッサ，およびそれらを繋ぐケーブルを含んで構成される．この構成を前提とし，クライアントからのイベント到着を契機に，各コンポーネントに順番にイベントが引き渡されていき，イベント処理が

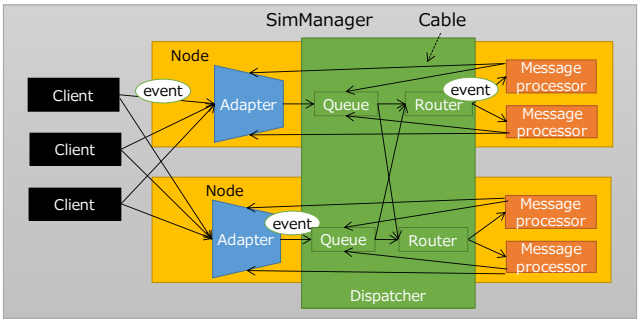

図 4 分散処理基盤を含む，分散処理シミュレータの構成

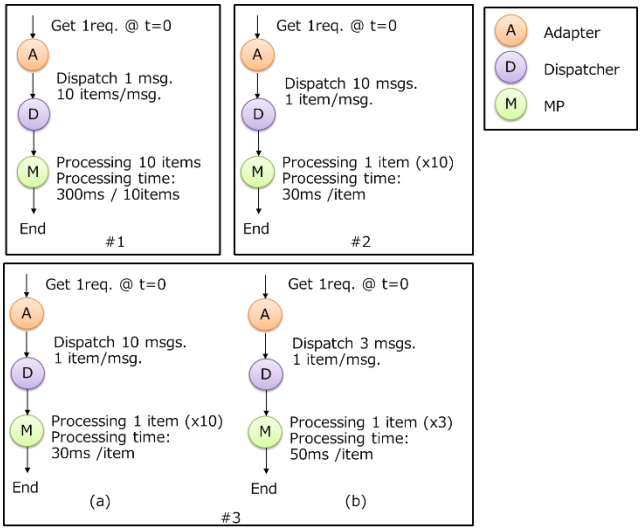

図 5 3 種類の処理フローイメージ

進行する．ユーザは，シミュレーションマネージャに対して，シミュレーション時間や，コンポーネントの数，メッセージプロセッサの同時実行数(並列度)，ディスパッチャのキュー長などのパラメータを与えて動作させることができる．

ここで，クライアントから送られる各イベントは，次の処理内容が何であるかを記憶しておくため，図 5 にあるような処理フローを記載したグラフ情報を保持している(グラフ情報保持には Python ライブラリ NetworkX [12] を使用した). この処理フローを記載したグラフ情報は，たとえば CSV や JSON 形式で与えることを想定する．この例では，最初にアダプタでリクエストを受け取った後，処理 A を行ったのち，1 リクエストを N 個のメッセージに分けてディスパッチし，メッセージプロセッサで処理 B と処理 C を実行する様子を記載している．別の表現をすると，図 5 の処理が，リクエスト発行先クライアントやディスパッチャ部分のルータのルーティングストラテジおよび内部状態カウンタ，メッセージプロセッサの使用状況に従って，図 4 のコンポーネントのいずれかに割り当てられ，処理フローに記載した順番に処理されていく．この処理フローは，それぞれのコンポーネントで要する処理時間，メッセージディスパッチ件数，キュー・ルータでの振分けロジック(ルーテ

ィングストラテジ)などを記載することができる．また，非同期処理だけでなく，複数のメッセージプロセッサの処理の同期待ち合わせを表現することができる．

ここまで，任意の分散実行基盤アーキテクチャを仮定し，これを再現する分散処理シミュレータの構成について説明した．次章では，ここまで説明した分散処理シミュレータについて，シミュレーション結果の妥当性と有効性を検証する．

5. 分散シミュレータを用いた評価

5.1 妥当性の検証

本節では，開発したプロトタイプ分散処理シミュレータに関して，出力結果の妥当性と有効性を検証する．図 3 に示した設計手法のコンセプトにおいて，分散処理シミュレータがヘテロジニアス分散アプリケーション構築エンジンに返すべき結果として，たとえば以下のものが挙げられる．

・全リクエストの処理完了時間 (≒単位時間当たりのスループット)
・1 リクエストあたりの平均レスポンス時間
・ノードごとのリソース (CPU，メモリ，ディスク) 平均使用量，最大使用量
・ネットワーク使用量
・ノード間や外部サービスのアクセス回数，同時接続数
・上記のような数値に閾値を設定した際の，閾値超過回数・閾値超過時間

ヘテロジニアス分散アプリケーション構築エンジンでは，これらの結果を用いることで評価関数にしたがって設計の良し悪しを判断し，数値化することを想定する．本稿では，これらのうち分散実装による高速化の観点で最も重要な全リクエストの処理完了時間に注目した．図 5 に示すように，1 件あたり 30ms を要する処理を 10 件分実行する想定のもと，10 件の処理を直列に実行したとき(#1)，前述の 10 件の処理を並列に実行したとき(#2)，前述の 10 件の処理を並列実行するのと同時に 1 件あたり 50ms を要する処理が 3 件発生したとき(ヘテロジニアス分散アプリケーションのとき)(#3)の 3 つのケースで処理時間を評価し比較した．なお，シミュレーションパラメータとして，表 1 に示すように，ノード数を 1，メッセージプロセッサの同時実行数(並列度)の上限を 10，アダプタの処理時間を 10ms，ディスパッチャを構成するルータの処理時間を 10ms，全てのコンポーネントの処理時間の標準偏差を 5ms，ディスパッチャのキュー長を無限とした．また，ディスパッチャ部のルータがメッセージ配信先ノードを選択するロジック(ルーティングストラテジ)は，ラウンドロビンとした．図 6 は，このときそれぞれの処理フローが分散基盤のコンポーネント上にどのようにマッピングされるかを示している．#1 は 1 つのメッセージプロセッサのみを使用し，#2 は全てのメッセージプロセッサを使用し，#3 はメッセージプロセッサの同時実行数 10 を処理対象メッセージ数 13 が上回るため，いずれかのメッセージがキューに格納され待ち状態になることを示している．シミュレーションが，この状況を再現できているかどうか，処理結果ログから検証する．

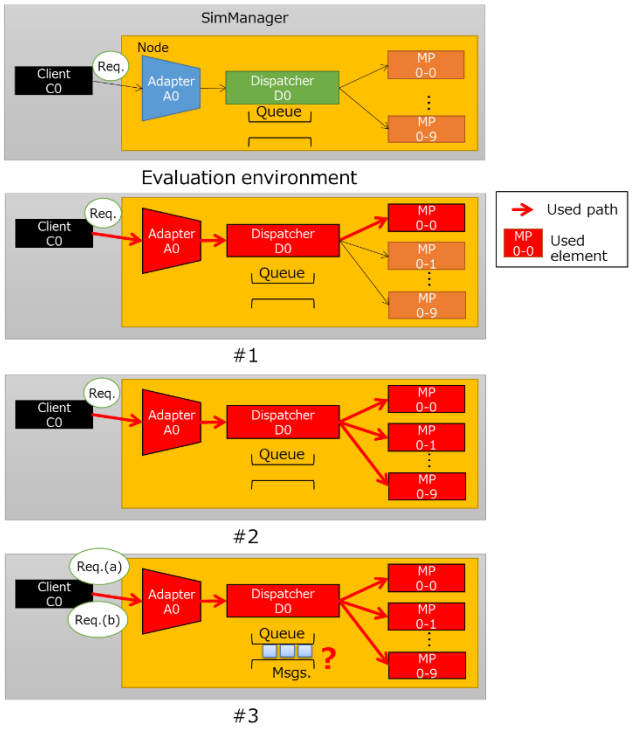

図 6 各データ処理フローのシミュレーションにおけるマッピングイメージ(図 5 に対応)

表 1 シミュレーションパラメータの設定

パラメータ	設定値
ノード数	1 (single node)
並列度	10
アダプタ処理時間	10 ms
ディスパッチャ処理時間	10 ms
各構成要素の処理時間 標準偏差	5 ms
キュー長	∞
シミュレーション時間	0 - 120ms
レポート出力頻度	1 / 5ms

表 2 は，図 5 のそれぞれの処理完了予想時間と，シミュレーションが出力した処理完了時間を示す．条件#1 のとき，リクエストはアダプタ，ディスパッチャを通過したのち，メッセージプロセッサにて 10 件分の処理として直列実行される．よって，10ms + 10ms + 30ms * 10 = 320ms と推測される．シミュレーション結果では 329ms となり，ほとんど結果が一致していることがわかる．続いて条件#2 のとき，

表 2 各処理フローの処理時間推定値と出力結果

フロー No.	推定値 [ms]	出力結果 [ms]
#1	320	329
#2	50	51（最初のメッセージ処理完了時間: 31）
#3	100	(a) 76 (最初のメッセージ処理完了時間: 37) (b) 100（最初のメッセージ処理完了時間: 69）

リクエストはアダプタで 10 件のメッセージに分けられ，それぞれディスパッチされてメッセージプロセッサに送られる．10 件のメッセージが 10 個のメッセージプロセッサで並列処理されるため，1 件あたりの処理時間は 10ms + 10ms + 30ms = 50ms と推測される．シミュレーション結果も 50ms となり，こちらも一致している結果が得られた．最あとに条件#3 のとき，処理フロー(a)の 10 件のメッセージと同時に処理フロー(b)の 3 件のメッセージがメッセージプロセッサに配信される．このとき，メッセージプロセッサの同時実行数上限が 10 であるため，同時に 13 件のメッセージを処理することはできない．そのため，処理フロー(a)(b)のうち 3 件のメッセージは，他の処理が終わるまで待つことになる．よって，合計処理時間は，アダプタの処理時間と，ディスパッチャの処理時間と，処理フロー(a)のメッセージ処理時間と，処理フロー(b)のメッセージ処理時間の合計で，100ms になると推測される．シミュレーションの実行結果も，合計処理時間は 100ms となっており，推測と一致している．なお，条件#3 のシミュレーション結果から，各処理フローの終了時間や，最初に処理が完了した時間もログ出力されている．これらの結果から，シミュレーションが推測通りの振る舞いをしていることを確認できた．

一方で，今回は開始時刻が同じという条件で 2 つの異なる分散処理をシミュレートしたが，処理の種類が N 個に増加し，ノード数が増加し，リソース使用状況などの複雑な条件を考慮していくとき，人手で処理時間を見積もることが困難になる．そのような場合に，分散シミュレータで発生事象を再現することが有効だと考える．

なお参考値として，シミュレーション実行時間についても説明する．Windows † 64-bit マシン (Inter® Core™ i7-8700 CPU@3.2GHz, RAM 16GB 搭載) で本シミュレーションの #3 を実行したときの実計算時間は 6ms であった．シミュレーション実行時間は，シミュレーション時間(120ms)やレポート出力頻度(1 行出力/5ms)に影響して変化することを確認している．離散イベントシミュレータでは，イベントの発生回数やレポート出力頻度，および同期処理のポーリング設定が実行時間に影響し，一方で分析処理時間は実行時間に影響しない．そのため，分析処理時間が秒オーダ以上であるような場合でも，シミュレーションを用いることで高速に事象を再現できる．

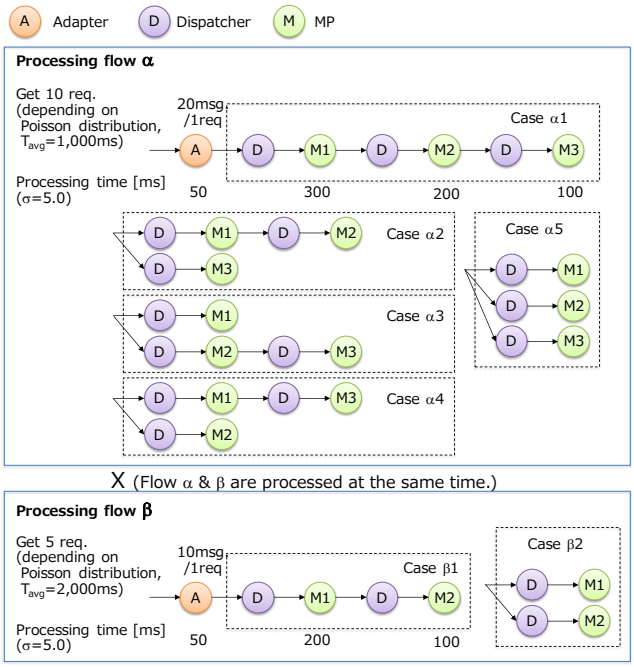

図 7 二つの設計フローからなるヘテロジニアス分散アプリケーションの例 (フローα, β が同時に実行される想定)

表 3 重付パターン

	重付パターン 1	重付パターン 2	重付パターン 3	重付パターン 4
w_1	100	100	1	1
w_2	100	1	100	1
w_3	1	1	1	1000
w_4	1	1	1	1
備考	フロー α, β 両方レイテンシ重視	フローα レイテンシ重視	フローβ レイテンシ重視	メッセージ最大滞留数重視

5.2 分散シミュレータを用いた設計最適化の試行

続いて，前節で検証した分散シミュレータを用いて，実際に設計最適化に利用可能かどうかを検証した．本来，設計最適化エンジンでは，ロジックが崩れない範囲内で設計フローを変化させたうえで(図 7 参照)分散シミュレータに入力し，シミュレーション結果を得たのち，以下のような評価関数 $f(n)$ に代入する．

† Windows: Microsoft Corporation の登録商標

表 4 シミュレーション結果と評価値

	シミュレーション結果				評価値 *f(n)* (太字はベスト，背景色つきはワースト)			
ケース	p1 [ms]	p2 [ms]	p3	p4 [%]	重付パターン1	重付パターン2	重付パターン3	重付パターン4
α1–β1	3,372	1,224	20	307	4.60.E+05	3.39.E+05	1.26.E+05	2.49.E+04
α2–β1	2,564	1,568	40	303	4.14.E+05	2.58.E+05	1.60.E+05	4.48.E+04
α3–β1	2,123	2,149	40	307	4.28.E+05	2.15.E+05	2.17.E+05	4.46.E+04
α4–β1	1,697	504	40	300	2.20.E+05	1.71.E+05	**5.24.E+04**	4.25.E+04
α5–β1	2,119	722	60	307	2.84.E+05	2.13.E+05	7.47.E+04	6.31.E+04
α1–β2	2,018	704	20	307	2.73.E+05	2.03.E+05	7.27.E+04	**2.30.E+04**
α2–β2	2,476	1,243	40	310	3.72.E+05	2.49.E+05	1.27.E+05	4.40.E+04
α3–β2	1,477	646	40	303	2.13.E+05	1.49.E+05	6.64.E+04	4.24.E+04
α4–β2	1,742	906	40	300	2.65.E+05	1.75.E+05	9.27.E+04	4.29.E+04
α5–β2	1,443	557	60	300	**2.00.E+05**	**1.45.E+05**	5.75.E+04	6.23.E+04

$$f(n) = \sum_{k=1}^{n} p_k * w_k \quad \ldots(1)$$

式(1) において，p_k はシミュレーション実行で得られたレイテンシ，スループット，CPU 使用率最大値などの結果であり，w_k は各結果に課す重みである．ユーザは，*f(n)* が小さいほど良い設計であるという前提のもと，重視するパラメータの影響が大きくなるように重み w_k を設定する．なお，局所最適解に陥らないようにシミュレーションに与える設計フローやパラメータを変化させる方法については，既存の大域的最適解を導くアルゴリズムを利用することができる．

ヘテロジニアス分散アプリケーションを対象に，上記の考え方を踏まえて設計を選ぶケーススタディについて述べる．図 7 は，論理的には同一だが，処理タイミングが異なる設計フローを示す．なお，各処理内容は独立で，かつ順序を入れ替えても成り立つものとする．このフローに対して，p_1=(フローαの平均メッセージ処理時間)，p_2=(フローβの平均メッセージ処理時間)，p_3=(ディスパッチャキューのメッセージ最大滞留数)，p_4=(CPU 使用率最大値)，の 4 つの結果を取得し，表 3 に示す重付パターンによって評価関数の計算結果がどのように変化するかを評価した．

表 4 は，計算処理結果と重付パターンごとの評価値を示す．結果から，重付パターン 1 のようなフローα, βの両方のレイテンシを最小化したいケースでは，フローα, βの処理を全て並列化した Case α5–β2，フローαを一部並列化した Case α3–β2の順に評価値が低く，これらの設計が望ましいことがわかる．一方で，重付パターン 3 のようなフローβのレイテンシのみを最小化したいケースでは, フローA を一部並列化した Case α4–β1，全処理を並列化した case α5–β2の順に評価値が低い. また, 重付パターン 4 のようなキューのメッセージ最大滞留数を抑えたいケースでは，フローB だけ並列化した Case α1–β2，フローα, βともに直列で実行する Case α1–β1の順に評価値が低い．これらの結果から，分散シミュレータに複数の設計フローを与えてシミュレーションを実行したのち，得られる結果に重み付けを行って評価することで，目的に応じた最適設計を導くことが可能であることを示した．

6. まとめ

近年，センサ/デバイスの普及や IoT データの増加を背景に，IoT データ活用の動きが加速している．大規模 IoT データ活用において，ストリーミング/バッチデータを対象に物理サーバ/仮想サーバのリソースを活用して高速処理するため，分散実行基盤が広く使用されている．

このとき，分析者が小規模データに対する分析などの加工処理を検証する段階ではその後の分散実行を想定しておらず，結果として複数の異なる分散処理対象を含んだロジックとなっているケースが存在する．このようなヘテロジニアスな分散アプリケーションを分散実行基盤上に実装するときに，ヘテロジニアスな処理が生み出すリソース使用状況や，データベース等外部サービスへのアクセス頻度について，事前に予想することが難しく，ボトルネック特定や設計変更に伴う影響の考慮で発生する工数を削減するための設計手法が期待されていた．

本稿では，設計容易化のための構想として，シミュレーションを用いたヘテロジニアス分散アプリケーション設計手法について検討した．そのなかで，本構想を実現するために必要な分散処理シミュレータについて，プロトタイプを試作した．プロトタイプの分散処理シミュレータは，ヘテロジニアスな処理を個々のイベントの処理に見立てて実行することができる．本分散処理シミュレータに関して，処理フローを変えたときの結果の変動を評価し，同一ロジックであっても処理フロー設計によって実行結果が異なることを示した．また，テストケースで本シミュレータを用いた設計最適化を試行し，有効性を示した．

参考文献

[1] “Top Ten Tips for Data Analysis to Make Your Research Life Easier!,” *http://www.statsmakemecry.com/smmctheblog/top-ten-tips-for-data-analysis-to-make-your-research-life-ea.html*

[2] “When Big Data Isn’t an Option,” *https://www.strategy-business.com/article/00250*

[3] “Six Tips to Suceed with Big Data,” *https://blog.kolabtree.com/tips-to-succeed-with-big-data/*

[4] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Journal of Communications on the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Proc. Mass storage systems and technologies (MSST)*, pp. 1-10, 2010.

[6] 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 *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 15-28, 2012.

[7] “scikit-learn: machine learning in Python – scikit-learn 0.19.1 documentation,” *http://scikit-learn.org/stable/*

[8] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, “Optimization by Simulated Annealing,” *Journal of Science*, Vol. 220, No. 4598, pp. 671-680, 1983.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, 2002.

[10] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, and A. J. Olson “Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function,” *Journal of Computational Chemistry*, vol. 19, no. 14, pp. 1639-1662, 1999.

[11] “Overview – SimPy 3.0.10 documentaion,” *https://simpy.readthedocs.io/en/latest/*

[12] “NetworkX - NetworkX,” *https://networkx.github.io/*