Elixirを基く画像処理フォールトトレラントパイプラインの構築と評価

Construction and evaluation of image processing fault-tolerant pipeline based on elixir

2024年03月

北九州市立大学大学院 国際環境工学研究科

ヨウ　サイケイ

（指導教員：山崎　進）

概 要

概要の内容

目 次

第1章　はじめに

1.1　背景

近年，機械学習や画像処理技術は現代では非常に普及しており，私たちの日常生活のさまざまな局面で広く活用されている．たとえば，医療画像，衛星画像分析，物体認識，スペクトル分析などである．また，機械学習や画像処理の精度を向上させるために，機械学習や画像処理の大規模化・複雑化が顕著に進んでいる [1].

このような機械学習や画像処理では，大量のデータを高速処理することが求められるため，高速化を実現する 1 つの手段としてマルチコア等を使った並列処理が重要となる．ただし，機械学習や画像処理のシステムにおいて，マルチコアプロセッサシステム上でコードを実行しても，自動的にコードが効率化されるわけではない．また，並行・並列プログラミングにおいて, デッドロックや性能低下の問題がある [2].

機械学習や画像処理のソフトウェア全体で最適な並列化を実現するため，山崎らが Elixir に着目し，機械学習や画像処理のデファクトスタンダードである Python と比べて Elixir の潜在的優位性を示した [3].

機械学習や画像処理に Elixir をより効率的に使うために，Nx と Evision が開発された．Nx は高速的に機械学習に対処できる．Evision は画像処理に対処できる．

しかし，Nx と Evision を使って，巨大な画像を処理すると，メモリ不足などによる異常終了になることがある．特に Evision と Nx のアクセラレータである EXLA が NIF 関数を使用しており，NIF 関数が異常終了すると，Erlang VM 全体が異常終了になる可能性がある．

このように多様かつ複雑なアプリケーションは機械学習や画像処理システムの異常終了により，極端な場合，人命や巨額の金銭が危険にさらされる可能性がある．

したがって，このような機械学習や画像処理システムの構築において耐障害性はかなり重要な部分である．

機械学習や画像処理において耐障害性の高いパイプラインを構築するために，山崎進は 2022 年の ElixirConf で巨大な画像に対する堅牢な分散並列処理について講演し，HtPipe を提案した [5]．HtPipe は NIF 関数が異常終了することで，Erlang VM 全体が異常終了するのは回避できるが，並列処理で関数の作成と呼び出しには Broadway を使用する方が簡潔で効率的である．また，Broadway ではデータの生成部分と処理部分が分離されており，一時的に一部の機能が失われたとしても，データの流れは整然と処理される．

したがって，そのようなパイプラインには，まだ改善すべき部分が存在していると考えられる．このような背景から，我々は，Broadway のデータ取り込みとデータ処理機能を機械学習や画像処理パイプラインとして活用することに着目する．

1.2　目的

この研究の目標は，機械学習や画像処理を行う Broadway パイプラインを構築し，その中で HtPipe を用いることで，耐障害性を向上させたパイプラインの開発と評価を行うことで

ある．

1.3 アプローチ

1.3.1 研究手段

本研究では，耐障害性を向上させたパイプラインの開発するための足掛かりとして，Elixir のライブラリ関数の Broadway を HtPipe と機能等価なパイプラインとして設計することに取り組む．

一つは，Broadway 自体の Supervisor を使って，Broadway 内の個々のプロセスを監視し，巨大な画像を処理すると，メモリ不足などによる異常終了が発生した場合に再起動する．これによって，異常終了になることを避ける．

もう一つは，Node の動作における親と子のノード間のメッセージ通信を介して，クラッシュの原因となる NIF 関数は子ノードで呼び出されて，このようにして，子のノードがクラッシュした後でも，親ノードが影響を受けない．これによって，NIF が異常終了しても，起動元の Erlang VM に波及することはないである．

1.3.2 評価手法

実装を行った Broadway パイプラインを二重システム制御実験で評価する．つまり，Broadway パイプラインを使用した実験と Broadway パイプラインを使用しない実験の対照実験が行う．

評価には Fault-Injection (以下，FI と略す) を使用して，子プロセスをシャットダウンする関数 1 と abort を伴う NIF 関数 2 を作成する．

さらに，作成した関数 1 と関数 2 をそれぞれパイプラインがインストールされている環境とインストールされていない環境で呼び出す．Broadway パイプラインの回復機能に基づくパイプラインに関するフォールトトレランスの効果評価する．

1.4 論文の構成

本論文の構成は，以下のとおりである．2 章に本手法を実装するうえで基となる Broadway と Node について述べる．3 章において，機械学習や画像処理を行う Broadway パイプラインの提案と実装方法について説明し，第 4 章で実装成果の評価結果を示す．第 5 章で本稿をまとめる．

第2章　関連研究

2.1　Broadway

Broadway は，Elixir チームによってデータパイプラインを作成および管理できるツールである．イベントと，メトリクス，自動応答，障害処理などの運用機能に重点を置いている [4]．本章では，2.1.1 節で，Broadway の耐故障性能の基盤技術となる Broadway の Supervisor 動作について簡単に述べたうえで，2.1.2 節で Broadway におけるデータ処理パイプラインを作成するための構成要素 GenServer について述べる．

2.1.1　Broadway の Supervisor 動作

Broadway の Supervisor は職場でスーパーバイザーが従業員のグループに対して責任を負うのと同じように, 割り当てられたプロセスに対して責任を負う．下位プロセスも上司に報告する必要があり, 上司はすべてがスムーズに実行されていることを確認する必要がある．これを実現するために, スーパーバイザには, 他のプロセスを効果的に管理できる一連の機能が付属している． プロセスを開始および停止したり, システム内で予期しないエラーが発生した場合にはプロセスを再起動したりできる．これらは終了をトラップするように構成されているため, 監視対象プロセスがエラーで終了すると, そのエラーは隔離され, それ以上伝播することはないである．

Broadway の Supervisor 動作は，主に以下の部品により 構成されており，それぞれの関係を図 2 に表す.

ツリーの最上部には, プロジェクト スーパーバイザーである Broadway Pipeline Supervisor があり,application.ex ファイルで定義されている．

```
def start(_type, _args) do
children = [
# Starts a worker by calling: Broadway.worker.start_link(arg)
]
# See https://hexdocs.pm/elixir/Supervisor.html
# for other strategies and supported options
opts = [strategy: :one_for_one, name: Broadway.Supervisor]
Supervisor.start_link(children, opts)
end
```

Broadway Pipeline Supervisor は,Broadway のさまざまなコンポーネントをすべて監視し, エラーが発生した場合は再起動する機能を持っている．Broadway Pipeline Supervisor が監視するプロセスとその目的を以下に示す.

- ProducerSupervisor: データ プロデューサー プロセスの監視を担当する． この監視ツリーには, 問題が発生している特定のデータ プロデューサーのみを再起動する必要があるため,:one_for_one の戦略が採用されている． すべてが問題なければ, 他のすべてのプロ

デューサーは実行を続けることができる．======= • ProducerSupervisor: データ プロデューサー プロセスの監視を担当する． この監視ツリーには, 問題が発生している特定のデータ プロデューサーのみを再起動する必要があるため,:one_for_one の戦略が採用されている．すべてが問題なければ, 他のすべてのプロデューサーは実行を続けることができる．

• ProcessSupervisor: プロデューサーからのデータを消費するワーカー プロセスを監視する責任がある．この監視ツリーには,:one_for_all の戦略がある．これが :one_for_one ではない理由は作成する処理コールバック関数がステートレスである必要があり, 発生したエラーはプロセスをクラッシュさせることなく処理できるためである． エラーが発生してプロセスがクラッシュした場合は,Broadway に関連する内部簿記の一部に問題が発生している可能性があ, すべてのコンシューマーを再起動する必要がある．

• BatchersSupervisor: バッチを動的に監視するために使用できる．この監視ツリーには,:one_for_all の戦略がある．これが :one_for_one ではない理由は ProcessSupervisor と同じである．

• Terminator: Broadway パイプラインを適切に停止する責任がある． これは, すべてのコンシューマ プロセスに, 終了後にプロデューサに再サブスクライブしてはならないことを通知する．また, すべてのプロデューサに対して, 現在のイベントをすべてフラッシュし, 後続のデータ要求を無視するように通知する．

Broadway Pipeline Supervisor には :rest_for_one という監視ポリシーがあることも重要である． その理由は, プロデューサー監視ツリーがクラッシュした場合, 親 Supervisor は後続のすべての監視ツリーを再起動し, パイプラインを動作中の新しい状態に復元できるためである．

これらすべてにより,Supervisor は耐故障性能の基盤技術となる．

2.1.2 Broadway の GenServer 動作

GenServer(generic server の略) は, クライアントとサーバーの関係のサーバーを実装するための動作モジュールである．状態を保持したり, コードを非同期に実行したりするために使用されるだけではない．また, 標準的なインターフェイス関数セットがあり, トレースおよびエラー報告の機能も含まれる [11].

GenServer モジュールは,GenServer の動作に必要ないくつかの関数のデフォルト実装を提供する．これらの関数はコールバックとして機能する．コールバックを実装するときは, 次の 2 つのことを知っておく必要がある．

• コールバック関数が受け取る引数

• どのような戻り値がサポートされている

GenServer の動作に関する次のコールバック関数について説明する．

• handle_call/3

handle_call/3 は, クライアントから GenServer プロセスへの同期呼び出しを処理するために使用される．

• handle_cast/2

handle_cast/2 は, 通常は他のプロセスまたはタイマーによってトリガーされる非同期イベントを処理するために使用される．

• init/1

init/1 は,GenServer プロセスの状態を初期化するために使用される必須のコールバック関数である． これはプロセスの開始時に呼び出され, 通常はいくつかの初期化操作を実行す

るために使用され, プロセスの初期状態を含むタプルを返す.

これら 3 つのコールバック関数が一緒になって GenServer のコア動作を形成し, 同期および非同期メッセージを処理できるようになり,Broadway パイプラインが同時実行および並列実行を処理できるようになる．また,Broadway パイプラインをカスタマイズできる．更にバックグラウンドでも実行される存続期間の長いプロセスを作成し, より優れた制御と柔軟性を提供する.

2.2 Node

Node は，Elixir が Erlang 言語から継承するライブラリ関数である [8]．Node により Elixir は Erlang の強力な分散機能へアクセス可能である [10].

Node には，作成された各ノードは実行中の Erlang ランタイム システムとなる．ランタイム システムには, 同時実行性, 分散, フォールト トレランスのサポートが組み込まれている.

これによって,Elixir コンテキストでの配布は, 言語の多くの機能がコードを変更せずにネットワーク上で動作することを意味する． 同時に, プロセスやメッセージ パッシングなどの基本的なプリミティブとプロセス リンクやモニターなどのより高度な概念も含まれる.

相互に接続する Erlang VM インスタンスは,Erlang ノードのクラスターと呼ばれる． Erlang ノードがクラスターに接続すると, その ID が他のすべてのノードに伝達され,Erlang ノードと他のすべてのノードの間に別のネットワーク接続が設定される．これは、いわゆる全結合メッシュになる．すべてのノードは他のすべてのノードに接続されているため, クラスター内の合計接続数は二次関数的に増加する．接続されたノードのセットのすべての接続の合計 c は

$$\sum_{c=1}^{n-1} c = \frac{n(n-1)}{2}$$

で定義される．通常のポイントツーポイント トラフィックに加えて,Erlang ノードは各ノード接続上でティックと呼ばれるハートビート メッセージを送信し, リモートノードがまだ生きているかどうかを確認する.

この一般的なアプローチにより, 高い拡張性と耐障害性を備えたネットワーク ノードを構築できる． 分散 Erlang の元の設計は, プライベートの安全なネットワーク内で実行されることを目的としていた． したがって, デフォルトでは Erlang のノード間通信は暗号化されず [9], 共有秘密 (Cookie) によってのみ保護される．異なるネットワーク内のノード間でメッセージを送信する時, 同じ Cookie で起動されたノードのみが相互に正常に通信できる.

第3章　提案手法

3.1　提案手法の概要

今回提案する一つ手法は，Elixir 言語で画像処理に Broadway のフォールトトレランスと並列処理を適用することを目指すというものである．Broadway は，スケーラブルなデータストリーム処理システムを構築するための Elixir のライブラリである．この提案は，特に画像の読み取り，画像の分割，保存操作などの画像バッチ処理にブロードウェイを使用することを目的としている．

既存手法からの変更点は，画像処理に Broadway を使用すると，並列処理プログラムがより簡潔に記述され，画像の流れがより明確になり得る．また，Broadway パイプラインで画像タスクを処理するときに監視および回復メカニズムにより，システムの耐障害性が向上し，プロセッサのクラッシュによる画像処理の中断が軽減される．

図 3 に既存手法 (C 言語を利用) と提案手法 (作成した Broadway パイプラインを利用) の実行フローの比較図を示す．

一つ手法は

3.2　実装例

以下で画像処理における Broadway パイプラインについて述べる．

3.2.1　画像処理におけるパイプラインの実装

パイプラインの基本構成:
def start_link(_opts) do
Broadway.start_link(_ __MODULE__ _,
name: _ __MODULE___,
producer: [
module: {Counter, 1},
transformer: {_ __MODULE__ _, :transform, [ ]},
concurrency: 1
],
processors: [
default: [concurrency: 4]
],
batchers: [
sqs: [concurrency: 2, batch_size: 10],
s3: [concurrency: 1, batch_size: 10]
]

)
end
画像の読み取り:
def init(_a) do
img1 = "https://upload.wikimedia.org/wikipedia/en/7/7d/Lenna_%28test_image%29.png"
|> HTTPoison.get!()
|> then(&(&1.body))
|> Evision.imdecode(Evision.Constant.cv_IMREAD_COLOR())
{:producer, img1} end
画像の分割:
def handle_demand(demand, img1) when demand > 0 do
src_file = "Lenna.png"
Evision.imwrite(src_file, img1)
div_size = 256
dst_file_ext = Path.extname(src_file)
dst_file_basename = Path.basename(src_file, dst_file_ext)
dst_files =
Stream.unfold(0, fn counter -> counter, demand - 9 end)
|> Stream.map(&"#dst_file_basename_#{&1}#{dst_file_ext}")
div_img =
Evision.imread(src_file)
|> Evision.Mat.to_nx()
|> Nx.to_batched(div_size)
|> Enum.map(&Evision.Mat.from_nx_2d(&1))
z_img =
Enum.zip(div_img, dst_files)
|> Enum.to_list()
{:noreply, z_img, dst_files}
end
コマンドラインに出力:
def handle_message(:default, message, _context) do
message.data
|> Enum.map(fn img, dst_file -> Evision.imwrite(dst_file, img) end)
|> IO.inspect
end

3.2.2 耐障害性を用いるパイプラインの実装

第4章　評価実験

本章では，Broadway パイプラインの耐故障性に関する評価実験を行う．まず 4.1 節で，実験を行う際に用いた環境について述べる．そして 4.2 節で提案手法の評価について述べ，4.3 節で障害の発生による耐故障処理の実験について述べる．

4.1　実験環境

実験を行っ た環境は以下の通りである．

- OS : macOS Sonoma ver 14.1.1
- CPU : Apple M1
- メモリ 16GB
- Elixir 言語コンパイラ : Elixir 1.15.4
- Erlang 言語コンパイラ : Erlang/OTP 26 [erts-14.0.2]
- Broadway バージョン : 1.0.7
- Evision バージョン : 0.1.2
- Nx バージョン : 0.3.0

4.2　評価手法

実験は 3 つの部分に分かれている．二重システム制御実験で，パイプラインのフォールト トレランスを評価する．

- 4.3.1 節ではパイプを使用して通常のタスクを処理する． パイプラインの可用性を確認する．
- 4.3.2 節では，FI を使用して、子プロセスをシャットダウンし，パイプラインで呼び出す関数 1 を作成する．タスクが異常終了したときの状況を実際にシミュレートする．プロセスの異常なシャットダウンに対する回復機能を確認する．
- 4.3.3 節では，FI を使用して，abort を伴う NIF 関数を作成し，パイプラインで呼び出す．NIF 関数の使用時に発生する abort をシミュレートする．NIF 関数が abort された後の Erlang VM システムのクラッシュに対する回復機能を確認する．

4.3　障害の発生による耐故障処理の実験

評価実験は, 安全性・実行速度の 2 つの観点から実施する．以下でそれぞれの 項目について説明する．

安全性:

安全性の検証手法として, 実際にメモリ不足などによる異常終了が発生した場合, 再起動ができないかどうか実験を行う．

実行速度:
既存の C 言語ライブラリを用いる場合と Elixir 言語のライブラリを用いる場合の実行速度の比較を行う.

4.3.1 画像処理における Broadway パイプラインの評価

4.3.2 Broadway パイプラインにおける回復機能 1 の評価

4.3.3 Broadway パイプラインにおける回復機能 2 の評価

第5章　まとめ

謝辞

参考文献

[1] Tom B. Brown, Benjamin Mann et al. 2020. Language models are few-shot learners. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc. Red Hook, NY, USA, Article 159, 1877–1901.

[2] Jean Bacon 1996. Concurrent Systems: An Integrated Approach to Operating Systems, Distributed Systems and Database. Addison-Wesley. pp.265–276.

[3] 山崎進，森正和，上野嘉大，高瀬英希: Hastega: Elixir プログラミングにおける超並列化を実現するための GPGPU 活用手法，情報処理学会第 120 回プログラミング研究会， Vol. 2018，No. 2，pp. 8，2018.

[4] Broadway: Build concurrent and multi-stage data ingestion and data processing pipelines with Elixir `https://github.com/dashbitco/broadway` (2023.10.2) .

[5] Susumu Yamazaki: Robust, Distributed, and Parallel Processing for Enormous Images `https://www.youtube.com/watch?v=RkMzCQm-Ws4t=1087s` (2023.11.12).

[6] Hsueh, M.-C., Tsai, T.K. and Iyer, R.K.: Fault Injection Techniques and Tools, IEEE Computer, Vol.30, No.4, pp.75–82 (1997).

[7] The Elixir Team: Elixir is a dynamic, functional language designed for building scalable and maintainable applications `https://elixir-lang.org`. (online), `https://github.com/elixir-lang/elixir` (2023.07.20).

[8] The Elixir Team: Functions related to VM nodes. Retrieved June 20, 2023 from `https://hexdocs.pm/elixir/1.16.0-rc.0/Node.html`

[9] Alexandre Jorge Barbosa Rodrigues and Viktória Fördős. 2018. To- wards Secure Erlang Systems. In Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang 2018). ACM, New York, NY, USA, 67–70. `https://doi.org/10.1145/3239332.3242768`

[10] Ericsson AB: Erlang Kernel Reference Manual Version 6.4, net kernel:monitor nodes. Retrieved June 20, 2023 from `http://erlang.org/doc/man/net_kernel.html`.

[11] The Elixir Team: GenServer: A behaviour module for implementing the server of a client-server. `https://hexdocs.pm/elixir/GenServer.html`