Vol.2021-SE-207 No.37
2021/3/2

手続き型プログラムのFaaS基盤での実行可能性向上

上野 優[1,a)]　新居 雅行[2,b)]

概要：計算機科学の実験環境として Jupyter Notebook が広く使われているが，その計算機クラスタの管理は一般的に容易ではない．一方，クラスタ管理不要で需要に応じた計算資源の利用を可能にするクラウドサービスの一つとして FaaS がある．しかし Jupyter Notebook で書かれた手続き型プログラムを FaaS で実行しようとすると，FaaS のリソース割り当てが他のサービスに比べて小さいため，多くの場合実行できない．本稿ではこの問題に対して，一連の手続き型プログラムを細かく分割することにより，リソース割り当て制限内に収める手法を提案する．さらに提案手法に基づいて既存のノートブックの分割を行い，その機能性と効率性を評価した．

Improving the Executability of Procedural Programs on FaaS Infrastructure

1. はじめに

機械学習や計算機科学の実験の記録や共有を円滑に行うために Jupyter Notebook [1] が広く使われている．Jupyter Notebook で作成されるプログラム（以下，ノートブック）は一般に，実験の記録や共有を容易にするために，コードの並びと実行順がほぼ一致する手続き型プログラムとなる．

このように作成されたノートブックは，例えば実験成果のデモンストレーションとして，実験環境外で再利用することもできる．特に多人数が同時に再利用する場合，少数人が用いる実験環境サーバと比べてより大きなリソースを提供できる計算機クラスタが必要となるがその構築・運用は容易でない．

サーバの構築・運用が不要で，需要に応じてリソースを提供する高スケーラブルな計算機環境として PaaS や FaaS がある．特にパブリッククラウドの FaaS は PaaS と比べて待機時間にコストが生じないことから，一時的なデモ環境として有力と考えられる．

しかし，既存のノートブックを FaaS で実行しようとすると，FaaS のリソース割り当てが比較的小さいためそのまま実行できないことがある．ノートブックを分割して各ノートブックの使用するリソース量を減らしたとしても，FaaS のリソース割り当ての制約を満たすとは限らない．そこで本稿では，実際に分割して実行できるようにするために，プログラムのデータ依存分析に基づいた分割箇所の決定および分割方法を提案する．

本稿の構成は次のとおり．2 節では FaaS の特徴，リソース割り当てが小さいことに起因する移植の問題，および既存手法の課題およびその解決アプローチを述べる．3 節では，データ依存解析に基づくプログラム分割位置の決定方法，プログラムの分割方法，および分割をつなぐ方法を述べる．4 節では手法の評価結果を，5 節では関連研究を，そして 6 節では実用に向けた課題を述べる．

[1] 株式会社富士通研究所
[2] ライフマティックス株式会社
a) ueno.masaru@fujitsu.com
b) nii@lifematics.co.jp

2. 課題と解決アプローチ

2.1 Function as a Service (FaaS)

FaaS は，開発者があらかじめ準備した任意のプログラムを，クラウド上で実行できるサービスである．

FaaS におけるプログラム実行の仕組みを述べる．開発者は (a) 実行したいプログラム，(b) プログラムが依存するファイル（例えば依存ライブラリや辞書データ），(c)FaaS から呼び出されるイベントハンドラ からなるパッケージを作成し，起動条件と合わせて FaaS にプロビジョニング（事前配備）する．FaaS では，起動条件が満たされると実行環境インスタンスが生成されその上に (a)〜(c) が展開され，イベントハンドラ (c) を通じて目的のプログラム (a) が呼び出される．プログラムの実行が終わるとそのインス

表 1　パブリッククラウドの FaaS のリソース割り当て
Table 1　FaaS resource quotas in public clouds

	AWS Lambda	Google Cloud Function
パッケージあたりサイズ上限	250MB	500MB
1実行あたり実行時間上限	900秒（15分）	540秒（9分40秒）
メモリ上限	約10GB	3GB

タンスは破棄され資源プールに戻される．

FaaS の長所として，プログラムの実行環境である計算機の管理が不要であること，自動的に計算資源がスケールアウトするためデモのような突発的な需要にも対応できること，プログラムが使用した CPU 時間に対してのみ課金され待機時間にコストが生じないことが挙げられる．

一方 FaaS の短所として，実行環境のリソース割り当て量が比較的小さいことが挙げられる．**表 1** に本稿執筆時点の FaaS (AWS Lambda, Google Cloud Function) のリソース割り当てを示す．リソース割り当て量は年々緩和されているものの，FaaS 関数とその依存を固めたパッケージサイズの上限が数 100MB であることや実行時間上限が数分〜10 数分程度であることが，既存プログラムの移植に当たって問題となる．クラウドサービス事業者の視点からすると，FaaS はリソース割り当てを厳しくする代わりにリソース利用効率を高めることができ，その結果計算資源を低価格で提供できるサービスともとらえられる．

2.2　移植の妨げとなる FaaS のリソース制約

パブリッククラウドの FaaS がユーザに割り当てるリソース量は比較的少なく，本節で述べるように既存プログラムの移植の妨げとなっている．

2.2.1　パッケージサイズ上限

AWS Lambda の場合, パッケージサイズが上限の 250MB を超えるとプロビジョニングできない．例えば形態素解析器の辞書（Python ライブラリ Janome の場合約 180MB）を他のライブラリとともにプロビジョニングできない．

2.2.2　実行時間上限

数分〜数 10 分の上限を超えると実行が打ち切られ，途中までの処理結果が失われる．

2.3　既存手法とその課題

2.2 節で述べたリソース制約を回避する既存手法として次が挙げられる．

2.3.1　単純な解決策

パッケージサイズ上限を回避するために，コードが依存するファイル群をそのままパッケージに格納するのではなくパターンマッチにより不要なファイルやライブラリを除外してから格納する方法や，大きいファイルを圧縮してから格納し実行時に展開する手法がある [2]．しかし，依存ファイルや依存ライブラリの数が増えればこの手法では解決できない．

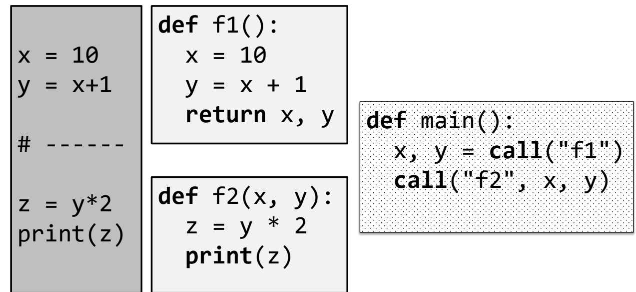

図 1　コード分割の例
Fig. 1　Example of chaining functions

依存ファイル群をパッケージ内に入れるのではなく，実行時にパッケージ外から取得することでパッケージサイズ上限を回避する方法も考えられる．しかし，取得のオーバーヘッドがかかることが懸念される．

AWS Lambda に限れば，Elastic File System に移し実行時にアタッチする方法（2020 年 6 月以降）や，Docker イメージに移す方法（2020 年 12 月以降）もあるが，他の FaaS には適用できず汎用的でない．

実行時間上限に関しては，アルゴリズムの工夫や並列化により高速化する方法が考えられる．しかし，機械的に行うことはできず作り替えは容易ではない．

2.3.2　コード分割による解決策

文献 [3,4] の Function Chain パターンは，コードを複数に分割してつなぐことでコードあたりの実行時間を短くし，実行時間上限の問題を回避する方法である．

図 1 にコード分割の実例を示す．図 1(a) は分割前の手続き型プログラム（コード）である．図 1(b) は図 1(a) を 2 つに分割し，それぞれを関数としたものである．分割位置の後段が参照する前段の変数は，前段の関数の返値および後段の関数の引数として渡すものとする．また，分割したコードを順次呼び出し図 1(a) と同一の結果を得るために，図 1(c) のようなコードが必要になる．図 1(c) は前段の関数 f1 を呼び出し，その返値を後段の関数 f2 の引数として呼び出すコードである．

この手法で分割することで同一の結果は得られるが，このままではパッケージサイズ上限の問題に対して有効かどうかは明らかではなく，コードの分割箇所と分割方法も自明ではない．

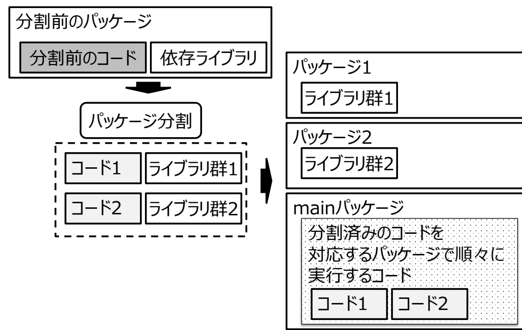

図 2 パッケージ分割のアイディア
Fig. 2 Ideas for package splitting

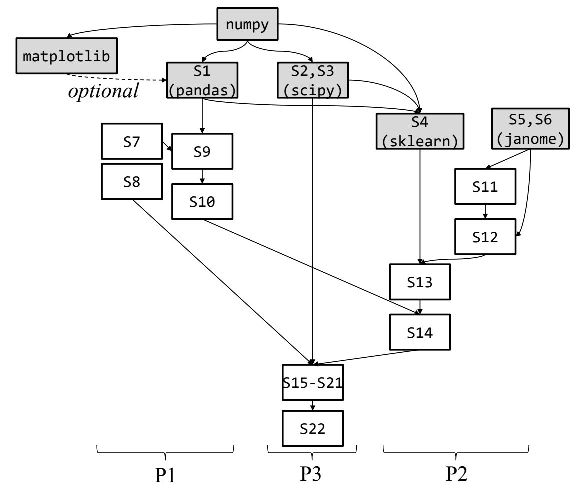

図 3 コード A.1 のデータ依存グラフ
Fig. 3 Data flow graph of the code A.1

2.4 解決アプローチ

2.4.1 コードの分割箇所の決定

手続き型プログラムのデータフロー解析によって得られるライブラリサイズに基づいて分割を決定するアプローチを 3 節で示す.

2.4.2 パッケージの分割および呼出

分割後のコードを FaaS で順に実行するためのパッケージを作成する.

図 2 に，与えられたパッケージを 2 つのパッケージに分割するアイディアを示す．分割後のパッケージ 1，2 は，分割後のコード 1，2 の依存ライブラリを含むパッケージである．ライブラリ群 1，2 は，元のライブラリ群のサブセットとなるため，各パッケージサイズは減少する．分割済みのコード 1，2 は，パッケージ 1，2 に含めても良いが，本稿では管理を簡単にするため別のパッケージ内にまとめて収めることとする．「main パッケージ」は，分割済みのコードを，対応するパッケージで順々に実行するためのパッケージである.

2.4.3 パッケージサイズの削減

コード分割を伴うパッケージ分割を行ったとしても，パッケージあたりのライブラリサイズの問題が解決しない場合は，オーバーヘッドがかかるものの，2.3.1 節で述べた実行時に依存ファイルを取得する方法と組み合わせることでパッケージのサイズを削減する.

単に実行時インストールを行うだけでなくコード分割も合わせて行うことで，1 つ 1 つのプロセスの実行時間の分割もでき実行可能性の向上に寄与するはずである.

3. FaaS 基盤での実行可能性向上のための手続き型プログラムの分割

ここでは Python 形式のノートブックのコード部分を出力した付録 A.1 のプログラムを AWS Lambda に移植することを題材として，手法を説明する．依存ライブラリサイズの合計が AWS Lambda の上限を超えるため，そのままではプロビジョニングに失敗する.

3.1 題材とするコード

題材のコードの主要な処理部分は，依存ライブラリの違いに着目すると次の 4 つの部分からなることがわかる．ここで Sn (n は整数) は付録 A.1 の文番号と対応する.

(1) S7-S10： データのロード・前処理
(2) S11-S14: sklearn で特徴抽出，janome で形態素解析
(3) S15-S17: scipy でクラスタリング
(4) S18-: グラフ描画，出力

なお S1〜S6 は依存ライブラリ読み込みの文 (import 文) である.

図 3 は，このコードのトップレベルの文を節点，データ依存関係（データフロー）を辺とするグラフである．ここで文 S から文 T へのデータ依存があるとは，文 S で定義された変数が再定義されずに文 T で参照されることである．灰色のブロックは依存ライブラリを表し，図 3 では特にサイズの大きいライブラリのみ記載している．分割位置を決めるには，コードから直接読み込まれるライブラリだけでなく，図 3 の numpy のように推移的に依存関係にあるライブラリの存在に留意する．なお破線辺は，始点のライブラリが必ずしも読み込まれるわけではない（ライブラリのメタ情報として依存関係が陽に宣言されていない）ことを示す．破線辺始点のライブラリ matplotlib は pandas のバックエンドとして動作し，S7-S21 では利用されず S22 で暗に利用されるライブラリであり，全てのパッケージで利用されるわけではない．この暗黙のデータ依存の存在は与えられたコードだけでは確認できず，実際に稼働させることで確認できた.

3.2 分割箇所の決定

題材のコードは 3.1 節で述べた (1)〜(4) の部分に分かれており，(2) の依存ライブラリサイズが大きいことからこの部分を独立したパッケージとする，すなわち (1), (2), (3, 4) の文をそれぞれそれぞれパッケージ P1, P2, P3 に分けることとするとパッケージ P1 は pandas, numpy，パッケージ P3 は pandas, numpy, scipy, matplotlib を利用し，いずれもライブラリサイズの制約を満たす．しかしパッケージ P2 は，janome, sklearn，numpy，pandas，scipy を利用し，このままではライブラリサイズの制約を満たさない．図 3 の P1, P2, P3 はパッケージ P1, P2, P3 に含まれる文を指している．

そこで一部のライブラリをパッケージから除き，2.3.1 節で述べた実行時にインストールする方法をとった．コード分割と実行時インストールを組み合わせることで，全てのパッケージのサイズ制約を満たすことができた．

分割箇所をまたぐ文の間にデータ依存がある場合，2.3.2 節で述べた方法で，そのデータを引き継ぐこととする．

3.2.1 通信量やストレージ利用量の最適化に向けた指針

分割位置の決定に当たって，考慮できれば望ましい指針を列挙する．

- 分割後の異なるパッケージが同じライブラリをなるべく重複して含まないこと
  例えば，可視化用のライブラリ、統計処理のライブラリを、別々のパッケージに配置できれば、各パッケージあたりの容量およびその総和は小さくなる．

- 通信するデータサイズが小さいこと
  分割後のコードの前段で生じた値を後段で利用する場合の通信量を減らす．分割を細かくすればするほどライブラリサイズは小さくなる一方で，通信量は増える．

- ライブラリ中立なデータ構造を通信すること
  特定のライブラリのデータ形式の通信を避けることで，分割前後でそのライブラリへの依存が重複することを避ける．

3.3 文中での分割

図 3 のグラフでは，制御構文や関数定義文のように複数の文を含む文もそうでない文も，等しく不可分な節点として表している．Python では複数の文を含む文として，if 文・while 文・for 文・try 文・def 文・with 文などがある．

このような文中での分割が必要となる場合，個別に対処が必要になる．付録 A.1 のコードの元となったコードは，大きな関数定義文中からさまざまなライブラリに依存した処理を行っている場合にあたり，関数呼出を呼出先コードで置き換えるインライン展開を行って関数定義文を無くすことで対処している．同様に for 文中で分割するためには，ループ文を展開することで対処する．

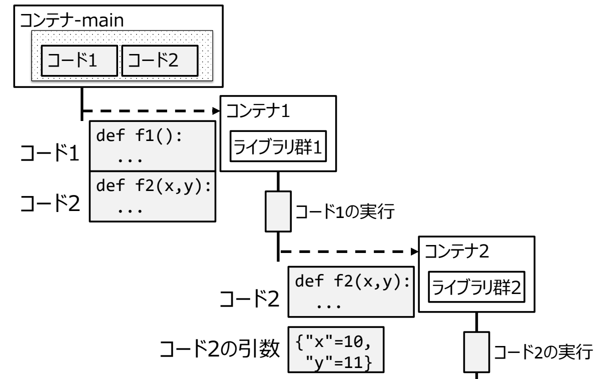

図 4 実行シーケンス（非同期 RPC）
Fig. 4 Sequence diagram (asynchronous RPC)

3.4 遠隔手続き呼出 (RPC)

分割したコードはそれぞれ別のインスタンスで実行されるため，遠隔手続き呼出 (RPC) によって一連の処理を「つなぐ」必要がある．

RPC の呼出側は一般に，呼出に付随するペイロードをシリアライズし，RPC リクエストを送信する．被呼出側は，ペイロードのデシリアライズ，処理の実行，および戻値の返信を行う．ペイロードとして，分割済みコードとデータ依存関係にあるデータ構造を送受することで，一連の処理を「つなぐ」ことを実現する．

AWS Lambda の場合，FaaS 関数を直接呼び出す際のペイロードサイズにも数 MB 程度の上限があり，上限を超える依存データを送受できない．そこで本手法では，シリアライズされたペイロードはオブジェクトストレージ (Amazon S3) に格納し，その URI を FaaS 関数に渡すこととする．

3.4.1 非同期呼出

FaaS の実行時間上限を考慮すると，後続の関数の終了を待たずに前段の関数を終了できると望ましい．すなわち同期呼出を非同期呼出に変換すれば良く，その一つの方法として継続渡しスタイル (CPS) に変換すればよい．図 1(b)(c) のような順に関数を呼びその結果を次の関数に渡す（直接スタイルの）コードは機械的に CPS に書き換えることができることが知られている．**図 4** に，図 1(b)(c) のコードを非同期呼出に変換した後のシーケンス図を示す．遠隔呼出のペイロードとして継続を渡すことで，分割したコードを各パッケージから展開されるコンテナインスタンスで順々に非同期に実行する．

3.5 パッケージ作成とプロビジョニング

FaaS パッケージの作成・プロビジョニングを容易にする開発フレームワークが多く公開されている．ここでは，AWS 上のリソースのプロビジョニングを自動化する AWS CloudFormation テンプレートを用いて，FaaS 関数の定義，そのイベントハンドラ・依存ライブラリ群・起動条件・使用リソース量の設定，プロビジョニングを一括で行うこととした．

依存ライブラリ群は，3.1 節で述べたデータ依存グラフから分かる，分割後のコードから読み込まれるライブラリ一式をまとめたものである．ただし容量削減のため，実行時に用いないテスト用ファイルや実行時に再度生成できるバイトコード (`*.pyc`) はパッケージから除外することとした．

4. 評価

4.1 既存ノートブックの分割実験

3 節の手法を，付録 A.1 のプログラムを対象に，機能性および効率性の観点で評価した．

4.1.1 機能性：分割後のコードの FaaS での実行可能性

付録 A.1 のプログラムに対して 3 節の手法を適用し 3 つのパッケージに分割することで，そのままでは FaaS で実行できないプログラムが FaaS で実行できるようになることを確かめた．

ただし，同プログラムの元となったノートブックには続きがあり，そこでは 1 つのライブラリが単独で数 GB の辞書ファイルを利用していたためその箇所の移植は本手法では不可能であった．

4.1.2 効率性：分割後のコードの実行時間

分割前のコードを仮想マシン (VM) で実行した場合と，分割後のコード群を FaaS 上で実行した場合の総実行時間を比べた．実験条件は次のとおりである．

VM VMware vSphere 5.5, 8 vCPU, 16GB Memory
FaaS AWS Lambda, 3GB Memory, 初回起動時（コールドスタート）
入力データ 約 16000 行 CSV, 約 19MB

表 2 に実行時間と分割後の通信量を示す．合計時間より，分割前後の実行時間はほぼ等しかった．

RPC の実行時間に影響するオーバーヘッドには次の (1)〜(6) がある．(1)〜(3) は呼出側の，(4)〜(6) は被呼出側のオーバーヘッドである．

(1) シリアライズ
(2) S3 にアップロード
(3) リクエスト送信
(4) コンテナの起動
(5) S3 からダウンロード
(6) デシリアライズ

表 2 実行結果
Table 2 Execution result

比較対象	各関数の実行時間 [s]				合計 [s]
	main	P1	P2	P3	
VM（分割前）	-	-	-	-	**236.2**
VM（分割後）	0.0	0.5	220.7	14.9	**236.1**
FaaS（分割後）					
起動時間	3.6	3.4	3.4	3.6	14.0
実行時間	0.1	2.1	204.0	9.3	215.6
合計	3.7	5.6	207.4	12.8	**229.6**

(a) 実行時間

呼出元・呼出先	通信量 [MB]
P1からP2	12.1
P2からP3	19.7

(b) 通信量

表 3 公開ノートブックでよく用いられるライブラリのサイズ
Table 3 Sizes of popular libraries in public notebooks

ライブラリ名	Version	推移的依存関係にあるライブラリの合計サイズ	ライブラリ単体のサイズ
numpy	1.19.4	22.2 MB	22.2 MB
matplotlib	3.3.3	54.7 MB	32.6 MB
pandas	1.1.4	64.1 MB	41.0 MB
scikit-learn	0.23.2	105.1 MB	25.2 MB
scipy	1.5.4	78.5 MB	56.4 MB
seaborn	0.11.0	155.2 MB	2.2 MB
ipython	7.19.0	19.5 MB	5.7 MB

ライブラリ単体のサイズは Linux 環境 (Ubuntu 20.04 64-bit, Python3.7) にインストールしたディレクトリから得た．推移的依存関係にあるライブラリの合計は，各ライブラリのメタデータからライブラリ間の依存関係を求めそのディレクトリサイズを合計して得た．

このうち，(4) は表 2 に示すとおり 3.4〜3.6 秒程度であった．一方，(1), (2), (5), (6) は転送量と Lambda-S3 間の帯域幅に依存して変わると考えられるが，評価実験ではその所要時間は起動時間に比べて非常に小さく，表には記載していない．(3) の所要時間はほぼ 0 秒であった．

4.2 公開されているノートブックに対する調査

インターネットに公開されているノートブックを対象に，FaaS での実行可能性を調査した．ここでは，FaaS での実行可能性とはこれまでに述べてきた AWS Lambda の制約を満たすかどうかのこととする．

4.2.1 GitHub に公開されているノートブック一般に対する調査

表 3 は GitHub (`https://github.com`) に公開されているノートブックを対象に，そこから読み込まれているライブラリのうち出現回数が大きいライブラリとサイズを示す．上位 7 件は文献 [5] で報告されているものを用いた．この表から，複数のライブラリを組み合わせて使うとパッケージサイズの問題が容易に生じることがわかる．

表 4 調査対象とした自然言語処理のノートブックとその依存ライブラリ
Table 4 A list of surveyed NLP Notebooks and their dependencies

#	Category	Title	直接読み込まれるライブラリ	依存ライブラリサイズ合計
1	NLP 101	An Introduction to Word Embeddings	spacy, pandas, sklearn, numpy, matplotlib, gensim, tqdm	335 MB
2		NLP with Pre-trained models from spaCy and StanfordNLP	spacy_stanfordnlp, spacy, tabulate, stanfordnlp, IPython	1620 MB
3		Discovering and Visualizing Topics in Texts with LDA	gensim, spacy, pyLDAvis, pandas	281 MB
4	Named Entity Recognition	Updating spaCy's Named Entity Recognition System	tabulate, spacy, sklearn, IPython, tqdm	226 MB
5		Named Entity Recognition with Conditional Random Fields	eli5, nltk, sklearn_crfsuite, sklearn, scipy, joblib	125 MB
6		Sequence Labelling with a BiLSTM in PyTorch	pandas, nltk, numpy, sklearn, torchtext, matplotlib, torch, tqdm	1707 MB
7	Text classification	"Traditional" Text Classification with Scikit-learn	eli5, pandas, sklearn, numpy, seaborn, matplotlib	192 MB
8	Sentence similarity	Simple Sentence Similarity	tensorflow, pandas, nltk, numpy, sklearn, matplotlib, scipy, seaborn, requests, tensorflow_hub, gensim, torch	3253 MB
9		Data Exploration with Sentence Similarity	tensorflow, pandas, tensorflow_hub, sklearn, numpy, matplotlib, gensim	1757 MB
10	Multilingual word embeddings	Introduction	gensim, fastText_multilingual, wordfreq	175 MB
11		Cross-lingual sentence similarity	pandas, nltk, numpy, sklearn, scipy, matplotlib, gensim, wordfreq	291 MB
12		Cross-lingual transfer learning	keras, numpy, wordfreq	1638 MB
13	Transfer Learning	Keras sentiment analysis with Elmo Embeddings	tensorflow, pandas, tensorflow_hub, numpy, matplotlib, keras	1686 MB
14		Text classification with BERT in PyTorch	pandas, sklearn, numpy, transformers, matplotlib, ndjson, torch, tqdm	1700 MB
15		Text classification with a CNN in PyTorch	sklearn, torchtext, torch, tqdm	1616 MB
16		Multilingual text classification with BERT	pandas, sklearn, numpy, seaborn, transformers, matplotlib, ndjson, torch, tqdm	1703 MB

4.2.2 自然言語処理・機械学習のノートブックに対する調査

公開されている自然言語処理の Python ノートブック形式のプログラム群を対象に，ライブラリサイズと FaaS での実行可能性を調査した．対象としたプログラム群は自然言語処理の専門家である Yves Peirsman らが作成したものであり，`https://github.com/nlptown/nlp-notebooks` のリポジトリに公開されている．プログラム群は，6 のカテゴリ・16 のプログラムからなり，自然言語処理ライブラリの入門から転移学習のような発展的な内容までを含む．

表 4 に，対象としたプログラム群の一覧と，それぞれの依存ライブラリを示す．依存ライブラリサイズ合計のセルは，250MB 未満を白色，250MB 以上・1000MB 未満を淡い灰色，1000MB 以上を濃い灰色で背景を塗り分けている．なお，プログラムの中にはライブラリに加えて 100MB 程度の入力 CSV データに依存するものもあるが，入力データはライブラリを呼び出すコードと異なり単体で外部ストレージに容易に移転できると考えられるため，依存ライブラリサイズ合計列の集計対象には含まない．

濃い灰色が背景の 9 個は，パッケージサイズの制約を満たすコード分割ができないものである．これは単体で巨大なライブラリ torch (約 1GB), tensorflow (約 1.5GB), stanfordnlp（torch に依存）に依存するためである．

淡い灰色が背景の 3 個 (#1, #3, #11) は，ライブラリサイズ合計が 250MB 以上であり，そのままプロビジョニング不可能なものである．このうちプログラム#1 は，前半は gensim, spacy (推移的依存も含めて約 229MB)，後半は pandas, sklearn, matplotlib (同 188MB) に依存するように 2 分割することでプロビジョニング可能となった．同様にプログラム#3 は，前半は spacy, pandas，後半は gensim, pandas に依存するように分割することでプロビジョニング可能となった．#3 については元のプログラム，分割後のプログラムのいずれも実行時間が上限を超えないことも確かめた，

白色が背景の 4 個 (#4, #5, #7, #10) は，ライブラリサイズ合計が 250MB 未満であり，本手法を適用せずにプロビジョニング可能なものである．このうちプログラム#4, #5, #7 は，反復的な学習を行う単一の文（for 文）の実行に時間がかかるため，実行時間上限を超えてしまう．

なお，#1, #10, #11 に関してはデータが入手できず実行時間上限を超えないかは確かめられていない．

まとめると，本手法は単体で巨大なライブラリに依存するプログラム 9 つには適用できない．残りの 7 つのうち，そのままではプロビジョニング不能な 3 つに関して，分割によってプロビジョニング可能なものが 2 つあることを確かめられた．このうちの 1 つは実行時間の制約も満たすことを確かめた．しかし，特に反復的な学習を行うプログラムでは単一の文の実行時間が長く，本手法の効果がないものもある．

5. 関連研究

計算を分割し FaaS で並列分散処理することで，コスト最適化をねらったアイディアが提案されている．文献 [6] は，コンパイルやユニットテストなどのジョブを，与えられた Makefile などの有向非巡回グラフ (DAG) に基づいて分割する．DAG として与えられない依存ファイルを別途動的解析する仕組みも備える．文献 [7] は，MapReduce の Map フェーズを FaaS で並列分散実行するアプローチをとる．Map フェーズでデータ集合の要素を変換するマッピング関数は独立に実行できるものが与えられるため，分割は行わない．このように文献 [6] の分割も，文献 [7] のマッピング関数の抽出も，FaaS のリソース割り当ての制約を条件とするものではないため，FaaS のリソース制約を満たさない分割が生じる恐れがある．なおこれらの文献では，分散実行する処理とその依存ファイル一式のパッケージング方式は，2.3.1 節の「実行時にインストールする」方式を

ⓒ 2021 Information Processing Society of Japan

採っている．

文献 [3,4,8] では，コード分割により FaaS の実行時間の制約を回避するアプローチが示されている．さらに文献 [8] では，コード分割位置に応じてデータ依存が変わり，その結果通信時間が大きく変化するという実験結果が示されている．しかし，本稿で述べたパッケージサイズの制約に関しては考慮されていない．

文献 [3,9] では，FaaS 関数が直接 FaaS 関数を呼び出すパターンの欠点が挙げられている．同期呼出すると呼出側がブロックされている間もコストがかかるため呼出側・被呼出側のコストが重複すること，（FaaS 関数をまたぐ）エラーハンドリングが困難になること，密結合すること（例えば一連の FaaS 関数呼出が最も遅いものに律速される），などであり，ジョブキューを介して非同期呼出にすることを推奨している．本稿では同じ問題意識から同期呼出は非同期呼出とした．密結合については高速化を目的化としていないため問題としなかった．ただしエラーハンドリングについては，本稿では単純なコードを扱っていたため問題にならなかったが，より複雑なケースでは問題になり得るため検討の余地がある．

6. 実用に向けた課題

実用に向けた課題を以下に述べる．

まず, 分割したコードの保守性の問題があげられる．コード分割に当たってインライン展開を行ったため，コードの可読性は明らかに悪化した．ループ展開はパッケージサイズの削減には寄与しなかったため適用しなかったが，適用すると可読性は悪化する．前節で述べたエラーハンドリングの問題も解決しなければならない．これらの保守性の問題を解決するには，分割後のコードは直接メンテナンスするのではなく，分割後のコードを自動生成するアプローチが良い．

また，本手法では分割位置の候補は 1 通りではないがその最適な選択は行っておらず，転送サイズや実行時間を考慮した最適化の余地がある．

本手法を適用できない例として，機械学習のモデル学習が挙げられる．理由として，モデル学習のように規模の大きな探索的な計算を行うには現在の FaaS のリソース割り当て制約は厳しいことと，GPU が使える FaaS が現状少ないことがある．また，深層学習の著名なフレームワークである Tensorflow や PyTorch はいずれも単体で 1GB 超のライブラリであり，C/C++で書かれた巨大なオブジェクトファイルを内包するため容易には分割できず，本手法の適用が難しい．

7. おわりに

本稿では，これまで着目されてこなかった FaaS のパッケージサイズ上限の問題を解決する，コード分割に基づく特定の FaaS に依存しないアプローチと実例を示した．本アプローチにより分割が可能であり，分割によるオーバーヘッドは小さく，実運用に耐え得る手法であることも確認できた．

FaaS は計算資源の利用効率が高く，運用不要で，スケーラビリティがある点で，開発者・クラウドサービス事業者の双方にメリットのあるプラットフォームである．しかし，実際にはリソース制約の厳しさに因る使い勝手の悪さを感じることも多くある．今後この短所を補うことで，FaaS のより広い応用が期待される．

参考文献

[1] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S. et al.: Jupyter Notebooks-a publishing format for reproducible computational workflows., *ELPUB*, pp. 87–90 (2016).

[2] Schep, D.: Serverless Python Requirements, `https://www.npmjs.com/package/serverless-python-requirement`.

[3] Taibi, D., El Ioini, N., Pahl, C. and Niederkofler, J.: Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review, *CLOSER 2020 - Proceedings of the 10th International Conference on Cloud Computing and Services Science*, SCITEPRESS, pp. 181–192 (2020).

[4] Leitner, P., Wittern, E., Spillner, J. and Hummer, W.: A mixed-method empirical study of Function-as-a-Service software development in industrial practice, *Journal of Systems and Software*, Vol. 149, pp. 340–359 (2019).

[5] Pimentel, J. a. F., Murta, L., Braganholo, V. and Freire, J.: A Large-Scale Study about Quality and Reproducibility of Jupyter Notebooks, IEEE Press, (online), available from ⟨https://doi.org/10.1109/MSR.2019.00077⟩ (2019).

[6] Fouladi, S., Romero, F., Iter, D., Li, Q., Chatterjee, S., Kozyrakis, C., Zaharia, M. and Winstein, K.: From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers, *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, USENIX Association, pp. 475–488 (online), available from ⟨https://www.usenix.org/conference/atc19/presentation/fouladi⟩ (2019).

[7] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I. and Recht, B.: Occupy the Cloud: Distributed Computing for the 99Computing Machinery, (online), available from ⟨https://doi.org/10.1145/3127479.3128601⟩ (2017).

[8] Spillner, J., Mateos, C. and Monge, D. A.: FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC, *High Performance Computing* (Mocskos, E. and Nesmachnow, S., eds.), Cham, Springer International Publishing, pp. 154–168 (2018).

[9] Beswick, J.: Operating Lambda: Anti-patterns in event-driven architectures – Part 3 — AWS Compute Blog, , available from ⟨https://aws.amazon.com/blogs/compute/operating-lambda-anti-patterns-in-event-driven-architectures-part-3/⟩ (accessed 2021-01-30).

ⓒ 2021 Information Processing Society of Japan

Vol.2021-SE-207 No.37
2021/3/2

付　録

A.1 分割の実験に用いたコード

```
import pandas as pd                                      # S1
import scipy                                             # S2
from scipy.spatial.distance import pdist                 # S3
from sklearn.feature_extraction.text import TfidfVectorizer # S4
from janome.tokenizer import Tokenizer                   # S5
from janome.analyzer import Analyzer                     # S6

input_file = "./data/input.xlsm"     # S7
output_file = "./data/output.csv"    # S8

df = pd.read_excel(input_file)                       # S9
s_col = apply_filter(df["説明"])                      # S10
tokenizer = Tokenizer()                              # S11
analyzer = Analyzer([], tokenizer, [])               # S12
vect = TfidfVectorizer(analyzer=analyzer.analyze)    # S13
tfidf_matrix = vect.fit_transform(s_col)             # S14

dist_condensed = pdist(tfidf_matrix.todense(), metric="cosine") # S15
z = scipy.cluster.hierarchy.ward(dist_condensed)     # S16
clusters = pd.concat(                                # S17
    [
        pd.Series(
            scipy.cluster.hierarchy.fcluster(z, maxclust, criterion="maxclust"),
            index=s_col.index,
            name="cluster-{col_name}-{threshold}".format(
                col_name=s_col.name, threshold=maxclust
            ),
        )
        for maxclust in range(5, 20, 5)
    ],
    axis=1,
)
clusters_list = [clusters]    # S18
from functools import reduce # S19
columns = [                   # S20
    "作成日",
    # --- snip ---
    "説明",
    "FAQID",
]
df_with_cluster = reduce(     # S21
    lambda left, right: left.merge(
        right, how="left", left_index=True, right_index=True
    ),
    [df[columns]] + clusters_list,
)
# S22
df_with_cluster["cluster説明--5"].value_counts().plot(kind="bar")
```

ⓒ 2021 Information Processing Society of Japan