Vol.2022-HPC-187 No.10
2022/12/1

行指向なコマンド/パイプラインへの増分計算の適用

佐藤 碧[1,a)]　新美 礼彦[2,b)]

概要：本論文では，シェルスクリプトにおける行指向なコマンド/パイプラインに増分計算を適用するコマンドである incrementalize を提案する．行指向なコマンド/パイプラインとは，出力が入力によって決定され，またそれらの入力を分割して与えても，最終的に得られる出力は，入力を分割せずに与えた場合の出力と同様になるようなコマンド/パイプラインを指す．そのような性質を満たすコマンド/パイプラインの再実行時に，以前と同様の入力が与えられた場合，それに対応する出力を事前にキャッシュしておくことで，再実行することなく出力を返すことが可能である．incrementalize は，そのようなコマンド/パイプラインを対象として，再実行時におけるキャッシュの検索と読み出し，新規の入力に対する出力のキャッシングを自動的に行う．これにより，前述の性質を満たす任意のコマンド/パイプラインに対して，与えられた入力のうち，新規に計算する必要がある部分についてのみ計算する増分計算を適用して処理速度を向上させる．そして，本提案の有効性を検証するために評価実験を行った．

Apply Incremental Computation to Line-oriented Commands/Pipelines

Abstract: This paper proposes a command “incrementalize”, which applies incremental computation to line-oriented commands/pipelines in shell script. Line-oriented commands/pipelines refer to that their outputs are determined by their inputs, and their final outputs are the same as the outputs with the non-divided inputs even if commands/pipelines are given divided inputs. Such commands/pipelines can output data without re-calculation by caching previous outputs, which are corresponding to specific inputs, when they get the same inputs as before. For their commands/pipelines, “incrementalize” searches and read caches on re-execution, and caches their outputs for new inputs automatically. Using the proposed command, we can improve the execution time of their commands/pipelines by applying incremental computation, which calculates only yet non-calculated new inputs. And we evaluate the efficiency of this proposition through experiments.

1. はじめに

UNIX シェル [1] は，様々な環境で使用されるインターフェースであり，そのコマンドを記述したシェルスクリプトは，データの加工やタスクの自動化等の処理を実行するためのツールである．そのなかで使用される各種コマンドは無数に存在しており，それらはシェルスクリプトで記述されている場合もあれば，まったく別のプログラミング言語で記述されていることもある．そのため，一般に，シェルスクリプトを実行する場合，各コマンドがどのような処理を実行するかは，実際に実行してみるまでわからない．

PaSH[2] は，そのような各コマンドに対して自動的に並列処理を適用するシステムであり，事前知識を利用してそれらのコマンドについて適切な並列処理を特定するフレームワークを実装している．そして，このフレームワークは，シェルスクリプトへの並列処理の適用以外にも利用できる [3]．そのうち，本論文ではシェルスクリプトへの増分計算の適用に注目した．それにより，PaSH とは異なるアプローチでシェルスクリプトの処理速度を向上させることが可能であるからである．増分計算とは，計算への入力が変更された際に，入力全体を再計算するのではなく，出力のうち変更された入力に依存する部分だけを計算することによって計算を効率化する技法である．通常，増分計算

[1] 公立はこだて未来大学大学院 システム情報科学研究科
Future University Hakodate, Graduate School of Systems Information Science
[2] 公立はこだて未来大学大学 システム情報科学部
Future University Hakodate, Faculty of Systems Information Science
[a)] g2119016@fun.ac.jp
[b)] niimi@fun.ac.jp

を実現するためには，専用のアルゴリズムやデータ構造を実装する必要がある．そのため，増分計算に対応していないシェルスクリプトやコマンドに，追加で増分計算を適用するためにはソースコードを修正する必要がある．前述のようにシェルスクリプトで使用される各コマンドは，いくつものプログラミング言語で実装されている可能性があるため，それら全てのソースコードを修正して増分計算を実装するのは容易ではない．ここで，前述の事前知識において，入力と出力の対応関係に関する分類が存在する．そのうち，入力が同じならその出力も同じになる性質をもっている分類に含まれるコマンドに関しては，その入力に対応する出力を事前に保存しておき，コマンド実行時にその入力が再度入力された際に対応する出力を返すことにより，入力全体に対する再計算を回避することができる．さらにそのうち，入力を分割して計算することが可能な性質をもつコマンドに関しては，前述のような入力全体ではなく，行単位で入力と出力を保存することによる再計算の回避が可能である．この性質は，コマンドの入力と出力のみに関係した性質であり，コマンドが実際にどのような処理を実行するかは無関係である．よって，この性質を利用して増分計算を実現することができれば，そのような性質をもったコマンドに対してそのコマンドのソースコードを修正することなく増分計算を適用することが可能であると言える．

本論文では，シェルスクリプトにおける行指向なコマンド/パイプラインに増分計算を適用するコマンドである incrementalize を提案する．なおこのコマンドは，PaSH が提供する事前知識にコマンドの事前知識に関するフレームワークと併用することで，コマンド/パイプラインのみならずシェルスクリプト全体に増分計算を適用することを想定している．ただし，本論文内では，実際に併用するところまでは扱わず，incrementalize 単体についてのみ評価する．ここでいうシェルスクリプトにおける行指向なコマンドとは，前述のように，それに対する入力が同じならその出力もまた同じものになり，さらに入力を分割して与えても，最終的に得られる出力は，入力を分割せずに与えた場合の出力と同じになるようなコマンドを指す．また，そのような性質を持つコマンドのみを UNIX シェルの機能であるパイプラインを使用してそれらの入出力を接続することで得られるパイプラインは，そのパイプラインを構成する個々のコマンドと同様に，パイプライン自体も同様の性質を持つ．incrementalize は，そのような性質を利用することで，対象となるコマンド/パイプラインに対して，再実行時におけるキャッシュの検索と読み出し，新規の入力に対する出力のキャッシングを自動的に行うことで増分計算を適用する．それによって，シェルスクリプトの処理速度を向上させる．

以後，本論文では，2 章では PaSH とそのなかで利用されるコマンドの分類に関する概要，3 章では増分計算の概要，4 章では incrementalize の提案，5 章では incrementalize の実装，6 章では incrementalize の実験と評価，7 章では関連研究についてそれぞれ述べ，最後に 8 章でまとめと今後の課題について述べる．

表 1 各 UNIX コマンドの並列化可能性の分類
Table 1 Classes of parallerizabilities for each command

分類名	純粋性	並列化	単純結合
Stateless	純粋	可	可
Parallerizable Pure	純粋	可	不可
Non-Parallerizable Pure	純粋	不可	不可
Side-Effectful	非純粋	不可	不可

2. PaSH

PaSH は，シェルスクリプトに自動的に並列処理を適用する CLI アプリケーションである．特徴は，その処理に各 UNIX コマンドの並列化に関する事前知識を使用することである．それらの事前知識は，主に各 UNIX コマンドの並列化についての分類と，実際の並列化の際に必要となる追加の処理についてである．

PaSH はまず，並列処理を適用するシェルスクリプトを解析して，シェルスクリプトと相互変換が可能なデータフローグラフに変換する．そのデータフローグラフについて，前述の事前知識に基づいてデータフローグラフを変換することで並列化を適用してから，再度シェルスクリプトに変換しなおすことでシェルスクリプト並列化を適用する [4]．このとき，その事前知識によって正しく並列化が適用可能であることが保証される場合にのみデータフローグラフを変換することで，その変換の有無によりシェルスクリプトから得られる出力に差が発生しないようにしている．こうすることで，事前知識が正しいという前提が必要にはなるが，変換前のシェルスクリプトの文脈を破壊することなく安全にスクリプトを変換することが可能である．

PaSH が利用する事前知識では，シェルスクリプトにおける各 UNIX コマンドの分類を特定することが可能である．**表 1** にその分類を示す．純粋性は，その分類に含まれる UNIX コマンドの出力がユーザからの入力にのみ依存して決定されるかどうかを表す．例えば，実行時にその瞬間の時刻を表示するような UNIX コマンドは，ユーザからの入力ではなく，ハードウェアから取得できる時刻情報を使用して出力を決定するため，純粋性はないと言える．並列化は，その分類に含まれる UNIX コマンドが，任意の入力について行ごとに入力を分割してからそれらに対して個別にコマンドを適用して得られる出力を連結してするという方式による並列化可能かどうかを表す．単純結合は，先の入力分割による並列化における出力の結合時に，単純に結合することが可能かどうかを表す．単純に結合できない場合，専用の処理を使用して個別に出力を結合する必要が

ある．その専用の処理は事前知識に含まれている．なお，Stateless は Parallerizable Pure のサブクラスであるため，Stateless に分類されるコマンドは Parallerizable Pure に分類されるコマンドであると考えることができる．同様に，Parallerizable Pure は Non-Parallerizable Pure のサブクラスであり，Non-Parallerizable Pure は Side-Effectful のサブクラスである．厳密には，これらの分類は UNIX コマンド単位ではなく，そのオプション引数や環境変数等のコマンドが実行する処理を決定するパラメータによっても変化する．よって，同じコマンドでも，指定されたオプション引数によっては，分類が変化する可能性がある．ここでは簡単化のために，それらの分類は UNIX コマンド単位で決定されるものとして説明した．

3. 増分計算

増分計算とは，計算において入力が変更された際に，その変更によって影響を受ける出力部分のみを計算して以前の出力のうちの適切な部分を更新することで，入力全体を再計算することを回避して効率よく出力を計算するためのソフトウェア技法である [5][6]．増分計算のアルゴリズムの概要は，計算を f，入力を x，x に関しての増分計算のための補助情報を Δx とすると，増分計算は $f(x+\Delta x)$ として表現できる．このとき，$x+\Delta x$ は変更された入力および変更された箇所の情報あるいは変更される前の入力等である．ただし，実際にどのような情報が必要になるかは増分計算のアルゴリズムによって異なる．通常，増分計算を適用にするためには，専用のアルゴリズムやデータ構造，ライブラリを使用して実装する必要がある．

増分計算が使用されている代表的なソフトウェアには，表計算ソフト [7] やビルドシステム [8] などがある．表計算ソフトであれば，あるセルが更新されたときに，そのセルを参照している関数があるセルについてのみ再計算することで，表全体の再計算を実行することを防ぐ．ビルドシステムであれば，ソースコードやオブジェクトファイル，実行ファイル同士のタイムスタンプを比較するなどして，必要なファイルのみを再コンパイルすることで，コードベース全体がコンパイルされることを防ぐ．このようにすることで，入力の修正に伴う処理の再実行に要する時間を短縮し，生産性を向上させることが可能である．

4. 提案

本論文では，シェルスクリプトにおける行指向なコマンド/パイプラインに増分計算を適用するコマンドである incrementalize を提案する．

行指向なコマンド/パイプラインとは, PaSH による UNIX コマンドの分類の内，Statelsss に分類されるものを指す．Stateless に分類されるコマンドの出力は，入力が同じなら，そのコマンドを何度実行してもその出力もまた同じになる．パイプラインを構成するコマンドが全て Stateless に分類されるなら，そのパイプラインの先頭のコマンドへの入力が同じであるとき，そのコマンドの出力もまた同じであり，それはそのまま後続のコマンドへの入力となり，やはりその入力も同じになる．後続のコマンドも Stateless であるため，入力が同じになることで，その出力もやはり同じになる．これは，Stateless に分類されるコマンドをいくつパイプで連結しても同様である．よって，シェルスクリプトにおけるパイプラインについて，それを構成するコマンドが全て Stateless に分類されるなら，そのパイプラインも Stateless に分類されるコマンドであるとして解釈できる．

incrementalize は，行指向なコマンド/パイプラインを実行した際に，その入力の各行に応じた出力をキャッシュとして保存しておくことで，再度同じコマンド/パイプラインしたときに以前と同じ入力行が入力された際に，コマンド/パイプラインを実行することなくキャッシュからその出力を読み出して返す．そして，新規の入力についてのみコマンド/パイプラインを実行する．これは，行指向なコマンド/パイプラインが持つ，入力を改行で分割してから個別にそのコマンド/パイプラインを適用しても同じ出力が得られる性質を利用することで実現される．その結果，行指向なコマンド/パイプラインに対して増分計算を適用することが可能になる．ただし，実装の都合上，入力の行単位ではなく，複数行単位でキャッシュしたほうが現実的である．増分計算のワーストケースは，入力が全て新規入力の場合である．このとき，入力行全てに対して個別にコマンド/パイプラインを適用するとなると，コマンドを実行する際のオーバーヘッドが各行数分だけ発生することになる．例えば，コマンドを実行する際のオーバーヘッドが 0.005s であるとすると，1000 行を処理するために 5.0s ものオーバーヘッドが発生することになる．一方で，一度に 10 行単位でキャッシュするようにすると，このオーバーヘッドは 0.5s になる．ただし，複数の入力行を 1 つのキャッシュで扱うと，そのうちの 1 行でも変更された場合，そのキャッシュは出力として再利用できなくなる．つまり，1 つのキャッシュで扱う入力行数を増やせば増やすほど，入力が変更された際にキャッシュを出力として再利用できなくなる可能性が高くなるということである．よって，1 つのキャッシュで扱う入力行数として適切な値を指定する必要がある．

5. 実装

incrementalize は，補助コマンド chash と，それを利用して増分計算を実行するシェル関数 incrementalize によって構成される．

5.1 chash

chash は，入力の読み込みとそれに対するハッシュ値の計算，そしてその入力のファイルシステムへの書き出しを行う補助コマンドである．C 言語によって実装されており，これは，同様の処理をシェルスクリプトで実装した場合のハッシュ値の計算速度やある程度巨大な入力データを扱う場合のエラーといった問題を回避するためである．パラメータは，一つのキャッシュで扱う入力の行数 *lines* と，読み込んだ入力を書き出すファイルを配置するディレクトリへのパス */path/to/tmp* である．これらのパラメータは，chash を実行する際のコマンドライン引数として指定する．なお，ハッシュ値の計算には，高速な非暗号学的ハッシュアルゴリズムである xxHash[9] を使用する．これは，入力に対するフィンガープリントの取得が主目的であり，高速にハッシュ値を求めることができれば問題ないのであり，暗号学的なハッシュ値の強度は不要だからである．

chash は，標準入力から 1 行ずつデータを読み出す．読み出したデータは，ハッシュ値を更新するための入力として使用したあと，*/path/to/tmp* 配下のファイルへ書き出される．このとき，それまでに読み出した行数の総数がパラメータとして事前に指定した行数 *lines* に到達した場合，あるいは標準入力からの読み出しが完了した場合，その時点でのハッシュ値とそれまでの入力を書き出した */path/to/tmp* 配下のファイルへのパスを標準出力に出力する．その後，読み出した行数の総数とハッシュ値計算の状態を初期状態にリセットし，さらに入力を書き出すファイルを新たなファイルに変更する．これらの処理を標準入力から全ての入力を読み出し終えるまで繰り返す．

5.2 シェル関数 incrementalize

シェル関数 incrementalize は，シェルスクリプト上の関数として実装され，コマンドと入力の組み合わせに対応したキャッシュが存在するかの確認，存在する場合のキャッシュの読み出しと存在しない場合のキャッシュの書き出しを行う．パラメータは，増分計算を適用するコマンドないしパイプライン *commands* と，*commands* を一意に特定する値 *command_hash* である．これらのパラメータは，シェル関数 incrementalize のコマンドライン引数として指定する．また，chash の出力をシェルのパイプラインを使用して標準入力から受け取る．それらの出力は 1 行ごとに処理される．

標準入力から読んだ行はパースされ，入力のまとまり *inputs* から算出されたハッシュ値と，*inputs* を書き出したファイルへのパスに分割される．そのとき取得したハッシュ値と，コマンドライン引数から取得した値 *command_hash*，そして事前に指定されたキャッシュファイルを保存しておくディレクトリへのパスから，キャッシュファイルへのパス *cache_path* を特定する．そのパスが指す先にファイルが存在し，かつキャッシュファイルが正常に作成されたことを示すファイル *complete* のタイムスタンプが，キャッシュファイルよりも新しい場合，コマンドライン引数で指定されたコマンドないしパイプラインと，*inputs* の組み合わせにおける出力がキャッシュされていると判断し，そのキャッシュファイルを cat コマンドによって標準出力に書き出す．先の条件を満さない場合，*inputs* を *commands* への入力として指定して実行し，また同時にコマンドライン引数として *cache_path* を指定した tee コマンドにパイプして，コマンドの実行と同時にその出力をキャッシュする．その処理が完了したら，*cache_path* に対応する *complete* を作成して，作成されたキャッシュファイルが完全であることを示す．これらの処理を，標準入力からの読み込みが完了するまで繰り返す．

5.3 incrementalize

incrementalize は，シェルスクリプトで実装されており，コマンドないしパイプラインに増分計算を適用するための CLI インターフェースを提供するコマンドである．chash とシェル関数 incrementalize が必要とするパラメータを，コマンドライン引数ないし環境変数経由で取得し，またそれらのパラメータを chash とシェル関数 incrementalize に渡して実行する．

6. 実験と評価

incrementalize を使用してコマンド/パイプラインに増分計算を適用した場合の処理時間を評価した．

6.1 実験内容

incrementalize を適用可能な条件である Stateless に分類されるいくつかのコマンド/パイプラインについて，それぞれ incrementaliize を使用しない通常実行時の場合，キャッシュが存在しない状態で incrementalize を適用した場合，実験で使用する入力についてのキャッシュが生成される状態で incrementalize を適用した場合の，3 パターンについて処理時間の計測を行った．これは，incrementalize を使用した増分計算によってコマンド/パイプラインの処理時間がどの程度変化するのかを確認するためである．また，incrementalize によってキャッシュが生成される際に発生するオーバーヘッドが処理時間にどの程度の影響を与えるのかを確認するためでもある．そのほか，incrementalize を実行する際に指定するパラメータの 1 つである，1 つのキャッシュで扱う行数の違いによる処理時間の変化を確認するために，$2^{10}(= 1024)$ から $2^{30}(= 1073741824)$ の間の 11 通りの行数を指定して incrementalize を実行し，その処理時間を測定した．実際に指定した行数は，実験結果を示すグラフ内に記載した．シェルスクリプトのインタープリタとして bash を使用した．スクリプトの処理時間は

表 2 評価実験で実行するコマンド/コマンドライン及びメタデータ
Table 2 Command/pipeline and their metadata for evaluation experiments

No.	コマンド/コマンドライン	ファイルサイズ (バイト)	行数
1.	sed 's /[^[:alnum:]]/ /g'	1075953000	5114500
2.	grep '[opq]'	1091209300	8233600
3.	tr ' ' 'n' | grep '.'	1073778000	2334300
4.	tr −c '[a−z][A−Z]' 'n' | grep '[A−Z]' | sed 1d | sed 1d | sed 2d | sed 3d | sed 5d | tr −c '[A−Z]' 'n' | tr −d 'n'	1073747200	31580800

表 3 実験環境
Table 3 The environment for experiments

AMI イメージ	Ubuntu 22.04
インスタンスタイプ	m6i.xlarge
vCPU	4
メモリ	16 (GiB)
EBS	50GB (SSD)

bash の予約語である time を使用して計測した．処理時間は，各条件においてそれぞれ 10 回計測を行い，それらの平均とした．処理時間を計測する incrementalize の実行前には，OS のディスクキャッシュを開放しておき，ディスクキャッシュの存在の有無による処理時間のはらつきを抑制した．また，処理時間を計測する incrementalize の標準出力は /dev/null にリダイレクトして破棄し，出力を描画することによって発生するオーバーヘッドを排除した．

表 2 に，実行したコマンド/パイプラインとその入力ファイルのメタデータを示す．入力ファイルは，特定のファイルをいくつも連結することによって作成されている．そのため入力ファイルの内容は，同様の内容が繰り返し出現することに注意する．これらのコマンド/パイプラインとその入力は，PaSH が評価に使用したものの一部であり，それらを本実験に向けて一部修正したものである．なお，それ以外のコマンド/パイプラインについても同様の実験を行ったが，そのうち，検索と編集を行う単体コマンドである sed，検索のみを行う単体コマンドである grep，短いパイプライン，長いパイプラインをそれぞれ代表的な例として本論文で取り上げた．

6.2 実験環境

実験環境として，Amazon EC2 を利用した仮想環境を用いた．構成を**表 3** に示す．

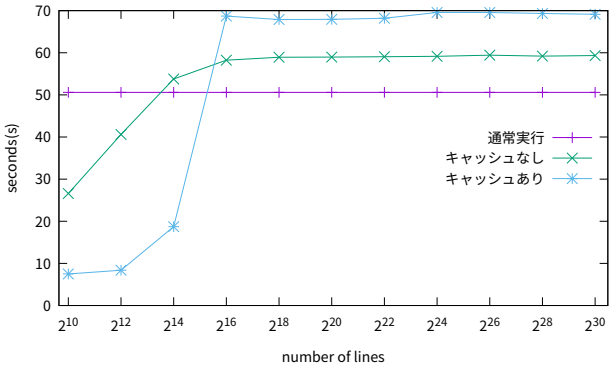

図 1 sed 単体に増分計算を適用した場合の実行時間
Fig. 1 Execution time of single sed applied incremental computation

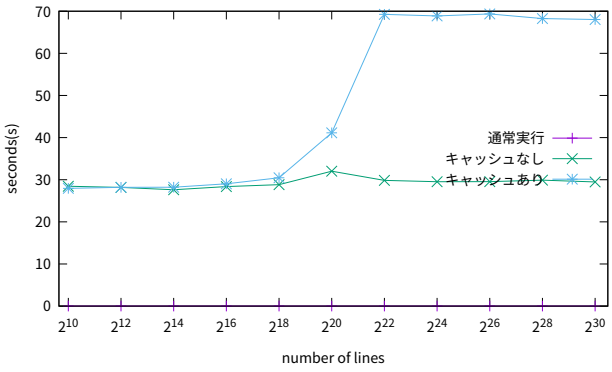

図 2 grep 単体に増分計算を適用した場合の実行時間
Fig. 2 Execution time of single grep applied incremental computation

6.3 実験結果

sed コマンドの単体実行を対象として incrementalize によって増分計算を適用した場合の実行時間を**図 1** に示す．一度にキャッシュする行数が 2^{14} 以下の場合の処理時間は，キャッシュが存在する場合に，通常実行と比較しておよそ 2.7 倍から 6.8 倍に高速化している．一方で，一度にキャッシュする行数が 2^{16} 以上の場合の処理時間は，キャッシュが存在する場合に，通常実行と比較しておよそ 0.7 倍になっている．このことからわかるように，一度にキャッシュする行数にもよるが，例えコマンド単体が対象であっても，増分計算を適用することで処理時間を高速化できることを確認できた．一方で，grep コマンドの単体実行を対象として incrementalize によって増分計算を適用した場合の実行時間を示した**図 2** からわかるように，コマンドによっては増分計算の適用により逆に処理時間を増大させる場合があることがわかる．

また，長いパイプラインを対象にして incrementalize によって増分計算を適用した場合の実行時間を**図 3** に示す．いずれの一度にキャッシュする行数の場合の処理時間においても，キャッシュが存在する場合に，通常実行と比較しておよそ 1.2 倍から 3.1 倍に高速化している．一方で，短

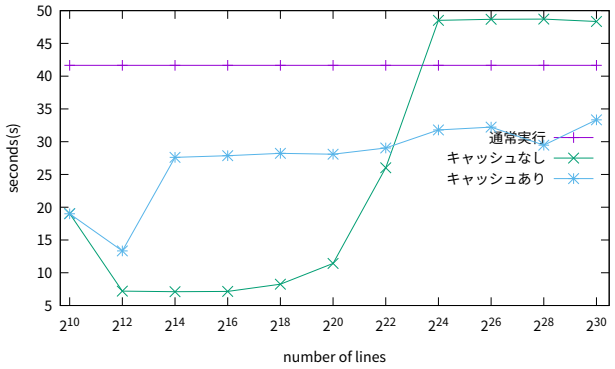

図 3 長いパイプラインに増分計算を適用した場合の実行時間
Fig. 3 Execution time of a long pipeline applied incremental computation

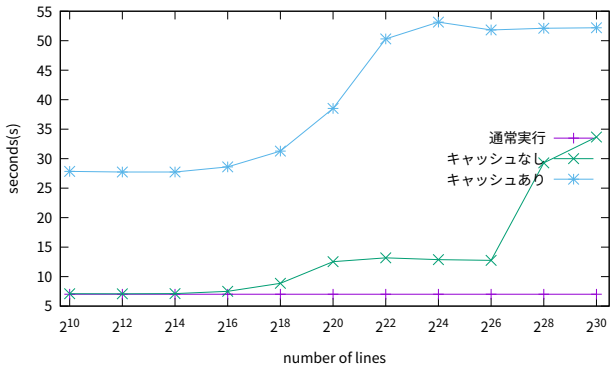

図 4 短いパイプラインに増分計算を適用した場合の実行時間
Fig. 4 Execution time of a short pipeline applied incremental computation

いパイプラインを対象として incrementalize によって増分計算を適用した場合の実行時間を示した**図 4** からわかるように，先のコマンド単体の場合と同様，パイプラインによっては増分計算の適用により逆に処理時間を増大させる場合があることがわかる．

このように，incrementalize は Stateless に分類されるコマンド/パイプラインについて，必ずしもその処理時間を向上できるとは限らないことがわかる．

通常実行とキャッシュが存在しない場合の処理時間を比較すると，通常実行が非常に高速であった grep コマンドの単体実行を対象として incrementalize によって増分計算を適用した場合を除けば，一度にキャッシュする行数にもよるが，キャッシュが存在しない場合の処理時間が通常実行の実行時間と同等か，それよりも高速であることがわかる．そのため，incrementalize によってキャッシュが生成される際に発生するオーバーヘッドが処理時間に与える影響は，無視しても問題ない程度であると考えられる．このとき，キャッシュが存在しない場合の処理時間が通常実行の実行時間よりも高速である場合がある．これは例えキャッシュが存在しない状態で incrementalize を実行していたとしても，その実行の最中に生成されたキャッシュが後続の処理において利用されるからだと思われる．前述のように，この実験における入力ファイルの内容は，特定の内容をいくつも繰り返し連結することによって構成されている．そのため，このような現象が発生する可能性はありえると考えられる．

キャッシュが存在する場合としない場合の処理時間を比較すると，後者のほうが処理時間が高速である場合がある．これは，キャッシュの読み出しのオーバーヘッドが，キャッシングのオーバーヘッドを上回っているからだと思われる．

一度にキャッシュする行数に注目すると，今回の実験で実行したいずれのコマンド/パイプラインにおいても，一度にキャッシュする行数は少ないほうが処理時間が高速になる傾向にあることがわかる．既存のキャッシュをどれだけ再利用できるかを考えると，一度にキャッシュする行数がより少ないほど，つまりキャッシュがより細かいほうが都合がよい．例えば，10000 行ごとにキャッシュを生成する場合と 1000 行ごとにキャッシュを生成する場合を考える．あるキャッシュについてそれに対応する入力が 1 行だけ変更された場合，前者の場合であれば，ある 10000 行分の入力に対する出力のキャッシュが再利用されないことになるが，後者の場合，単純に考えると，1000 行分のある入力に対応する出力のキャッシュが再利用されないことになるが，残りの 9000 行の入力については，それに対応する出力のキャッシュを再利用することが可能である．よって，incrementalize によって適用される増分計算においては，その処理時間の面でもキャッシュの再利用の面においても，一度にキャッシュされる行数が少ないほうがよい傾向にあると言えることがわかる．

7. 関連研究

POSH は，本論文で利用した PaSH と同様に，コマンドに関する事前知識を使用することでシェルスクリプトの処理速度を向上させるフレームワークである [10]．本論文では増分計算を，PaSH では並列処理を適用したが，こちらはシェルスクリプトの実行時におけるネットワークストレージへのアクセスによって発生するネットワーク IO をできるだけ削減するために，処理をリモートのホストに対して移譲することで，シェルスクリプトの実行速度を向上させる．

8. おわりに

シェルスクリプトにおける行指向なコマンド/パイプラインに増分計算を適用するコマンドである incrementalize を実装した．また，条件を満たすいくつかのコマンド/パイプラインを対象として incrementalize を実行し，それによって適用される増分計算がコマンド/パイプラインの処理時間を高速化できるかを評価した．その結果，incrementalize

はコマンド/パイプラインの処理時間を高速化できることがわかったが，一方で高速化できないコマンド/パイプラインが存在することもわかった．

今後は，より多くのコマンドやパイプラインを対象として incrementalize を適用し性能評価を行う．また，今回は各入力行単位ではなく複数の入力行単位のキャッシングによる評価を行ったが，より少ない行数でのキャッシング時の性能評価を行う．そのほか，実際に incrementalize と PaSH を組み合せて，コマンド/パイプライン単位だけでなくシェルスクリプト全体に対して増分計算を適用できるようにする．

参考文献

[1] Bourne, S. R.: Unix time-sharing system: the unix shell, *The Bell System Technical Journal*, Vol. 57, No. 6, pp. 1971–1990 (online), DOI: 10.1002/j.1538-7305.1978.tb02139.x (1978).

[2] Vasilakis, N., Kallas, K., Mamouras, K., Benetopoulos, A. and Cvetković, L.: PaSh: light-touch data-parallel shell processing, *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 49–66 (2021).

[3] Greenberg, M., Kallas, K. and Vasilakis, N.: Unix Shell Programming: The next 50 Years, *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, New York, NY, USA, Association for Computing Machinery, p. 104–111 (online), DOI: 10.1145/3458336.3465294 (2021).

[4] Handa, S., Kallas, K., Vasilakis, N. and Rinard, M. C.: An Order-Aware Dataflow Model for Parallel Unix Pipelines, *Proc. ACM Program. Lang.*, Vol. 5, No. ICFP (online), DOI: 10.1145/3473570 (2021).

[5] Ramalingam, G. and Reps, T.: A Categorized Bibliography on Incremental Computation, *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, New York, NY, USA, Association for Computing Machinery, p. 502–510 (online), DOI: 10.1145/158511.158710 (1993).

[6] Pugh, W. and Teitelbaum, T.: Incremental Computation via Function Caching, *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, New York, NY, USA, Association for Computing Machinery, p. 315–328 (online), DOI: 10.1145/75277.75305 (1989).

[7] Hammer, M. A., Phang, K. Y., Hicks, M. and Foster, J. S.: Adapton: Composable, Demand-Driven Incremental Computation, *SIGPLAN Not.*, Vol. 49, No. 6, p. 156–166 (online), DOI: 10.1145/2666356.2594324 (2014).

[8] Erdweg, S., Lichter, M. and Weiel, M.: A Sound and Optimal Incremental Build System with Dynamic Dependencies, *SIGPLAN Not.*, Vol. 50, No. 10, p. 89–106 (online), DOI: 10.1145/2858965.2814316 (2015).

[9] Collet, Y.: xxHash: Extremely fast hash algorithm (2016).

[10] Raghavan, D., Fouladi, S., Levis, P. and Zaharia, M.: POSH: A Data-Aware Shell, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, pp. 617–631 (online), available from ⟨`https://www.usenix.org/conference/atc20/presentation/raghavan`⟩ (2020).