修士論文

シェルスクリプトへの増分計算の適用

公立はこだて未来大学大学院　システム情報科学研究科
情報アーキテクチャ領域

佐藤 碧

指導教員　　新美 礼彦

提出日　　2023 年 2 月 13 日

Master's Thesis

Apply Incremental Computation to Shell Script

by

Aoi Sato

Graduate School of Systems Information Science, Future University Hakodate
Media Architecture Field
Supervisor:　　Ayahiko Niimi

Submitted on February 13, 2023

Abstract– This study proposes leveraging cache to apply incremental computation to shell scripts. This proposal contains the "CaSh" command, which applies incremental computation to pure commands/pipelines in the shell script, and the "incrementalize" command, which does the same but to line-oriented commands/pipelines in the shell script instead. Pure commands/pipelines are those whose outputs solely based on their inputs. When these commands/pipelines get the same inputs as before, they can output data without recalculating it by caching past outputs that correspond to certain inputs. Line-oriented commands/pipelines are described as "pure" commands/pipelines, and their final outputs are the same as the outputs with the non-divided inputs even if the commands/pipelines are given divided inputs. When employing a cache that matches a portion of prior inputs thanks to the input divisonability attribute, such commands/pipelines can output data without recalculating even when the inputs are not the same as previous ones. For these commands/pipelines, "CaSh" and "incrementalize" search and read the cache on re-execution and automatically create a cache for their outputs for fetch inputs. Utilizing the two proposed commands, we can enhance the execution time of their commands/pipelines by applying incremental computation, which only calculates newly added inputs that have not yet been calculated. And through test, we assess this proposition's effectiveness. Then, the result shows that the two commands can make shell script executions faster.

Keywords: Shellscript, Incremental Computation, Cache

概 要： シェルスクリプトにキャッシュを使って増分計算を適用する手法を提案する．本提案には，シェルスクリプトにおける純粋なコマンドあるいはパイプラインに増分計算を適用するコマンドである CaSh と，シェルスクリプトにおける行指向なコマンドおよびパイプラインに増分計算を適用するコマンドである incrementalize の二つが含まれる．純粋なコマンドあるいはパイプラインとは，その出力が入力によって決定されるようなコマンド/パイプラインを指す．そのようなコマンドあるいはパイプラインは，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．行指向なコマンドあるいはパイプラインとは，純粋なコマンドあるいはパイプラインであり，かつそ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あるいはパイプラインを指す．そのようなコマンドあるいはパイプラインは，入力を分割できる性質を利用することで，以前と完全には同じでない入力が与えられた場合でも，一致している部分に関してキャッシュを使用することで，再実行することなく出力を返すことが可能である．CaSh および incrementalize は，それぞれ対象とするコマンド/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．その結果，両方のコマンドでシェルスクリプトを高速化可能であることを確認できた．

キーワード： シェルスクリプト, 増分計算, キャッシュ

目次

第 1 章　序論 1

1.1　背景 1

1.2　目的 3

1.3　章構成 4

第 2 章　関連研究 5

2.1　UNIX シェル 5

2.2　増分計算 5

第 3 章　背景知識 7

3.1　シェルスクリプト 7

3.2　並列化に関するコマンドの分類と性質 9

第 4 章　手法提案 13

4.1　概要 13

4.2　CaSh 15

4.3　incrementalize 15

第 5 章　CaSh 17

5.1　設計 17

5.2　パラメータ 18

5.3　処理 18

5.4　スクリプトの変形例 19

5.5　PaSh へのコントリビュート 22

第 6 章　incrementalize 23

6.1　chash 23

6.2　シェル関数 incrementalize 26

6.3　incrementalize 28

第 7 章　実験と評価 30

7.1 概要 30
7.2 CaSh 30
7.3 incrementalize 35
7.4 実験で考慮しなかった事項 51

第 8 章 おわりに 54
8.1 まとめ 54
8.2 今後の展望 54

付録 A 試作型 incrementalize 67
A.1 概要 67
A.2 設計 67
A.3 パラメータ 68
A.4 処理 68
A.5 実装上の工夫 70

付録 B 試作型/改良型 incrementalize の比較 72
B.1 実験目的 72
B.2 実験内容 72
B.3 実験環境 74
B.4 実験結果 74
B.5 実験のまとめ 81

付録 C incrementalize の実験結果 84

第 1 章

序論

1.1 背景

UNIX シェル [1] とは，主に UNIX 系 OS へのインターフェースを提供するコマンド/プログラミング言語である．そのコマンドを記述したスクリプトである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として利用されている．

ここで，シェルスクリプトの実行速度を向上させることを考える．その場合，主に以下の三つの方法が考えられる．

- 各コマンドの性能向上
- インタプリタの性能向上
- スクリプトの改善

前述のように，シェルスクリプトは UNIX シェルの各種コマンドから構成される．そして，シェルはそれらのコマンドを先頭行から順次実行していく．そのため，それらのコマンドの処理速度が向上すれば，シェルスクリプト全体の処理速度も向上することになる．しかし，シェルスクリプトに記述される各種 UNIX コマンドは無数に存在する．そして，それらのコマンドは，一般に様々なプログラミング言語によって実装されている．そのため，各種 UNIX コマンドの処理速度向上は，それを実装する各種プログラミング言語ごとの関心事である．コンパイラやインタプリタの改善，実装しているアルゴリズムの改善などが考えられる．例えば，プログラミング言語の一つである Ruby は，JIT コンパイラを導入して処理速度を向上させるといった取り組み [2] をしている．

また，インタプリタの性能向上についても，前述の各コマンドの性能向上と同様である．これは，インタプリタ自体も一つの UNIX コマンドであるためである．シェルには，組み込みコマンドや予約語が存在する．そのため，そういったコマンドについては，インタプリタの性能改善で処理速度を改善できるかもしれない．だが，そういったコマンドはシェルスクリプトに含まれるコマンドのなかでは少数である．

すると，残りの選択肢はスクリプトの改善である．これは，例えばプログラマがよりよい，

高速に処理を実行できるようなスクリプトを書くことで実現可能である．だが，それは自明であるため，ここではそれ以外の方法を考えることにする．プログラマによる人手を使わない場合，機械的にスクリプトを変形することが考えられる．そのためには，機械的にスクリプトを解析して，適切な変形を適用する必要がある．その際，スクリプトの文脈を破壊することなく変形しなければならない．だが，これは非常に困難である．なぜなら，シェルから見ると，シェルスクリプトが実際にどのような処理をするのかが不明だからである．

シェルスクリプトに記述される各種 UNIX 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．シェルには，一般にいくつかの組み込みコマンドや予約語が実装されている．それらのように，シェル自身が提供するコマンドであれば，そのコマンドがどのような処理をするのかをシェル自身がわかっていると言える．だが，そのようなコマンドは限定的であり，多くのコマンドは，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たりする．

まったく別のプログラミング言語で実装されているようなコマンドの場合，そのコマンドがどのような処理をするのかは，シェルにとってはまったくもって不明である．なぜならシェルは，それらのコマンドについて，組み込みコマンドや予約語ではないためにシェル上では存在しないコマンドであることしか解釈できず，それぞれのコマンドの違いを判別できないからである．技術的には，リバースエンジニアリングによって実行ファイルやオブジェクトファイルを解析するなどして，それらのコマンドの挙動を解析することが可能であるかもしれない．だが，一般に，リバースエンジニアリングは人間の手によって処理される部分に強く依存している [3]．そのため，一般にシェルが，そのような処理によって，コマンドがどのような処理をするかを理解することは困難であると言える．

シェルスクリプトによって記述されたコマンドの場合，実装によっては，シェルから，そのコマンドがどのような処理をするかを理解することが可能かもしれない．実装に使用されるコマンドが，シェルの組み込みコマンドや予約語のみの場合，それらのコマンドは全てシェルが，なにをするのかを理解できているコマンドだからである．しかし一般に，シェルスクリプトによって記述されたコマンドは，そのなかにシェルの組み込みコマンドや予約語だけではなく，別のプログラミング言語で実装されたコマンドが含む．すると，シェルスクリプトによって記述されたコマンドの処理の内，別のプログラミング言語で実装されたコマンドを使用した部分については，そこでどのような処理が実行されるのかを理解することが困難である．処理の一部が不明であれば，処理全体でも最終的にどのような処理をするのかが不明である．よって，やはり別のプログラミング言語で実装されているようなコマンドの場合と同様に，シェルスクリプトによって記述されたコマンドの場合でも，一般にシェルがそのようなコマンドがどのような処理をするのかを理解することは困難である．

このように，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[4] 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 [5]．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．

1.2 目的

本論文では，シェルスクリプトに対して増分計算を適用して処理速度を向上させる手法を提案する．そして，その提案手法を，UNIX コマンドである CaSh と incrementalize として実装する．

CaSh は，シェルスクリプトにおける純粋な計算を対象にして増分計算を適用するコマンドである．純粋な計算とは，出力が入力によってのみ決定されるような計算である．CaSh は，入力として受け取ったシェルスクリプトに含まれている，そのような性質をもつ UNIX コマンドに対してその入出力をキャッシングすることで増分計算を適用する．

incrementalize 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．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．

このように，シェルスクリプトや UNIX コマンド，パイプラインを対象に増分計算を適用する UNIX コマンドによって，その処理速度を向上させる．

1.3 章構成

1 章では本論文の概要について述べる．2 章では本論文の関連研究について述べる．3 章では本論文に関する背景知識について述べる．4 章では提案手法について述べる．5 章では提案手法の実装の一つである CaSh について述べる．6 章では提案手法の実装の一つである incrementalize について述べる．7 章では，それぞれの提案手法の実装について行った実験と評価について述べる．8 章では，本論文のまとめについて述べる．

また，本論文には付録が存在する．付録 A 章では，本論文の提案手法の実装の一つである incrementalize の試作版について述べる．付録 B 章では，本論文の提案手法の実装の一つである incrementalize の試作版と改良版について述べる．なお，改良版の incrementalize とは，6 章で述べているものを指す．付録 C 章では，量の関係で省略した一部の実験結果について述べる．

第 2 章

関連研究

2.1 UNIX シェル

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 [6]．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．

JC[7] は，UNIX コマンドの出力をデータ交換用のデータフォーマットである JSON に変換する UNIX コマンド兼 Python ライブラリである．このコマンドは，コマンドの出力をパースすることで JSON に変換する．また，コマンドの出力のみならず，特定のファイルフォーマットについても JSON 変換可能である．それらのデータをパースする際，JC は事前にユーザから提供された情報を用いる．JC は，各コマンドやファイルフォーマット用のパーサーを実装しており，JC のユーザは実行時にどのパーサを使用するかを指定する．これらは，UNIX コマンドに関する事前知識を使用した処理であると言える．

シェルスクリプトについては，3.1 にて詳しく述べる．

2.2 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 [8][9]．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 [10] やビルドシステム [11] 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．

第 3 章

背景知識

3.1　シェルスクリプト

本論文におけるシェルスクリプトに関する基本的な用語を説明する．

3.1.1　コマンド

シェルスクリプトは，UNIX シェルのコマンドの羅列から構成される．

UNIX シェルのコマンドは，そのコマンド名によって特定される．ただし，実際には，コマンド名は，そのコマンドの実行ファイル名である．そのため，同一のコマンドであっても，異なるコマンド名を持っていたり，逆に，まったく別のコマンド名であっても同一のコマンドである場合もある．また，同一のコマンドであっても，コマンドのバージョンや実装が異なる場合もある．

UNIX シェルのコマンドは一般に，指定された単一のコマンドに対して，コマンドライン引数が与えたり，環境変数を指定することでその動作を変更することが可能である．よって，同一のコマンドを実行する場合でも，実行時のコマンドライン引数や環境変数の値が異なれば，そのコマンドは異なる動作をすることになる．そのため，コマンドがどういった動作をするのかを特定する場合は，単なるコマンド名の特定だけでなく，コマンドライン引数や環境変数といった，コマンドの動作を決定するための要素全てを考慮する必要がある．

以下に，UNIX コマンドの一つである echo コマンドを実行した場合の出力を用いて，コマンドライン引数の違いによってコマンドの動作が変化する例を示す．echo コマンドは，コマンドライン引数で受け取った値をそのまま出力する UNIX コマンドである．よって，同じ echo コマンドを実行しても，コマンドライン引数が異なれば，その出力は異なる．ここで，$で始まる行はコマンドプロンプトを示す．

```
$ echo foo
foo
```

```
$ echo foo bar
foo bar
```

同様に，UNIX コマンドの一つである ls コマンドを実行した場合の出力を用いて，環境変数の違いによってコマンドの動作が変化する例を示す．ls コマンドは，引数無しで実行した場合，カレントディレクトリにあるファイルやディレクトリのファイル名一覧を出力する UNIX コマンドである．そして，QUOTING_STYLE という環境変数の値によって，出力するファイル名のクオーテーションのフォーマットを指定することが可能である．例えば，QUOTING_STYLE という環境変数に c という値を設定した場合，ls コマンドは，ファイル名をプログラミング言語の一つである C 言語における文字列リテラルであるようにフォーマットして出力する．ただし，ls コマンドの実装によっては，QUOTING_STYLE という環境変数を参照しない場合もある．ここでも，$で始まる行はコマンドプロンプトを示す．

```
$ ls
foo.txt bar.txt
$ QUOTING_STYLE=c ls
"foo.txt" "bar.txt"
```

3.1.2 パイプライン

シェルスクリプトにおけるパイプラインは，UNIX コマンドを，制御オペレータである|で結ぶことで構成される．なお，パイプラインに対して，さらに制御オペレータ|で UNIX コマンドを結ぶことが可能である．つまり，シェルスクリプトにおいて，任意個の UNIX コマンドを制御オペレータ|で結びパイプラインを構成することが可能である．また，単一の UNIX コマンド実行であっても，制御オペレータ|が 0 個のパイプラインであると解釈できる．

コマンド A とコマンド B を制御オペレータ|で結んだ場合，コマンド A の標準出力は，コマンド B の標準入力に接続される．よって，コマンド A を実行した際の標準出力は，コマンド B を実行した際に標準入力を読むことで取得できる．また，いくつかのコマンドを制御オペレータ|で接続したパイプライン A について，パイプライン A の先頭のコマンドの標準入力として指定された入力を標準入力に，パイラプイン A の末尾のコマンドの標準出力に指定された出力を標準出力とした一つのコマンドであると解釈できる．すると，パイプラインを構成することは，既存のコマンドを組み合せることによって即席のコマンドを作ることであると解釈できる．

パイプラインの実行時，そのパイプラインを構成する各コマンドは，そのパイプラインを解釈したシェルのインタプリタの子プロセスとして，それぞれ別々のプロセス上で実行される．ただし，シェルの実装やオプションによっては，パイプラインの最後のプロセスだけは，

そのパイプラインを解釈したシェルのインタプリタを実行している親プロセス上で実行される場合もある．

以下に，UNIX コマンドを制御オペレータ|によって結んでパイプラインを構成して実行する例を示す．ls コマンドにコマンドライン引数として-1 を渡して実行すると，各ファイル名を一行ずつ出力するようになる．grep コマンドは，標準入力から読んだ値のうち，コマンドライン引数から受け取った文字列に合致する行のみを出力する．tr コマンドは，標準入力から読んだ値のうち，コマンドライン引数から受け取った文字の対応関係に応じて文字を置換する．wc コマンドは，標準入力から読んだ値の行数と単語数，バイト数を出力する．ここでも，$で始まる行はコマンドプロンプトを示す．

```
$ ls -1
foo.txt
bar.txt
$ ls -1 | grep f
foo.txt
$ ls -1 | grep f | tr 'a-z' 'A-Z'
FOO.TXT
$ ls -1 | grep f | tr 'a-z' 'A-Z' | wc
      1       1       8
```

3.2 並列化に関するコマンドの分類と性質

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 [12]．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

表 3.1 各 UNIX コマンドの並列化可能性の分類

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

定することが可能である．表 3.1 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．

3.2.1 Stateless の例

Stateless に分類されるコマンドの性質を，Stateless に分類されるコマンドである sed コマンドによって例示する．sed コマンドは，標準入力から読み込んだデータに対して，コマンドライン引数から受け取ったコマンドに応じて変更を加えて出力する．以下の例では，読み込んだ各行について，`bar` という文字列を発見した場合，それを `BAR` に置き換える．このように処理が各行に対して個別に適用される場合，複数行の文字列 `ss` について，`ss` を一つのコマンドに入力として与えた場合に得られる出力と，`ss` の各行をそれぞれ別のコマンドに入力として与えた場合に得られる出力は同一になる．ここでも，`$`で始まる行はコマンドプロンプトを示す．

```
$ { echo foo; echo bar; } | sed 's/bar/BAR/g'
foo
BAR
$ { echo foo | sed 's/bar/BAR/g';
    echo bar | sed 's/bar/BAR/g'; }
foo
BAR
```

3.2.2 Parallerizable Pure の例

Parallerizable Pure に分類されるコマンドの性質を，Parallerizable Pure に分類されるコマンドである wc コマンドによって例示する．wc コマンドは，標準入力から読み込んだデータに対して，行数と単語数，バイト数を計測して出力するコマンドである．論理的には，あるデータにおける行数と単語数あるいはバイト数といった統計情報は，そのデータの計測方法に関係なく同一である．そのため，そのデータ全体をそのまま計測した場合でも，あるいはそのデータを分割してから計測しても，最終的に得られる統計情報は変化しない．しかしながら，後者の場合，MapReduce[13] のように，データの分割とその結果の適切な結合が必要になる．そのため論理的には，Parallerizable Pure に分類されるコマンドも，Stateless に分類されるコマンドと同様に，処理が各行に対して個別に適用されると解釈可能であるものの，複数行の文字列 ss について，ss を一つのコマンドに入力として与えた場合に得られる出力と，単純に ss の各行をそれぞれ別のコマンドに入力として与えた場合に得られる出力は同一にはならない．今回の場合であれば，得られた統計情報を各列ごとにその合計を計算すれば，入力を分割しない場合とした場合とで同一の出力を得ることができる．ここでも，$で始まる行はコマンドプロンプトを示す．

```
$ { echo foo; echo bar; } | wc
      2       2       8
$ { echo foo | wc;
    echo bar | wc; }
      1       1       4
      1       1       4
```

3.2.3 Non-Parallerizable Pure の例

Non-Parallerizable Pure に分類されるコマンドの性質を，Non-Parallerizable Pure に分類されるコマンドである md5sum コマンドによって例示する．md5sum コマンドは，標準入

力あるいはコマンドライン引数で指定されたファイルから読み込んだデータに対して，ハッシュ値を計算して出力するコマンドである．ただ，ハッシュ値を計算する場合，Parallerizable Pure に分類されるコマンドと同様に，Stateless に分類されるコマンドとは異なり，入力を分割して処理した場合と入力を分割しないで処理した場合の出力は異なる．得られる出力は，分割した入力それぞれに対するハッシュ値になる．そして，それらは一方向に計算されることから，個別のハッシュ値同士を使用して，その元となる入力を結合して得られたデータから計算したハッシュ値を求めることは不可能である．このように，Non-Parallerizable Pure に分類されるコマンドは，Parallerizable Pure に分類されるコマンドとは異なり，入力を分割して処理した際に得られた出力を結合するなどして分割前の入力を使用した計算結果を求めることは不可能である．よって，Non-Parallerizable Pure に分類されるコマンドは，入力を分割して個別に処理することができない．ここでも，$で始まる行はコマンドプロンプトを示す．

```
$ { echo foo; echo bar; } | md5sum
f47c75614087a8dd938ba4acff252494  -
$ { echo foo | md5sum;
    echo bar | md5sum; }
d3b07384d113edec49eaa6238ad5ff00  -
c157a79031e1c40f85931829bc5fc552  -
```

3.2.4 Stateful の例

Stateful に分類されるコマンドの性質を，Stateful に分類されるコマンドである date コマンドによって例示する．date コマンドは，ユーザからの入力は受け取らず，コマンド実行時のシステムの時刻情報を出力する．内部的には，システムから時刻情報を入力として受け取っていると言える．時刻情報は刻一刻と変化していくため，コマンドをいつ実行するかによって，date コマンドの出力は変化する．このように，Stateful に分類されるコマンドは，ユーザの入力に依存せずにコマンドの出力が決定される．ここでも，$で始まる行はコマンドプロンプトを示す．

```
$ date
Thu Jan 12 00:00:00 PM JST 2023
$ date
Fri Jan 13 12:24:56 PM JST 2023
```

第 4 章

手法提案

4.1 概要

本論文では，シェルスクリプトに対してキャッシュを使った増分計算を適用する手法を提案する．具体的には，シェルスクリプトに含まれる各コマンドあるいはパイプラインに対して，その入出力をキャッシュすることで，同一の入力に対してキャッシュから出力を返し，また入力が変更された場合にのみコマンドあるいはパイプラインを実行することによることで，必要な計算のみを実行するという方法で増分計算を実現する．

この手法による増分計算を適用できるのは，3.2 における UNIX コマンドの分類のうち，少なくとも Non-Parallerizable Pure に属している UNIX コマンドである．これは，UNIX コマンドの入出力をキャッシングする都合上，入力が同一であれば，出力も同一になる必要があるからである．この性質を満たさない場合，入力が同一でもコマンドを実行するたびに，出力が変化する可能性があるということになる．これは，入力から出力が一意に決まらないということである．そのようなコマンドに対して入出力のキャッシュを行なうと，特定の入力に対するコマンドの出力が最初にキャッシュした際の出力に固定されてしまうことになる．これは，入出力のキャッシングを行わない場合のコマンド実行と比較して，そのコマンド本来の動作を変化してしまうということである．その結果，そのコマンドの実行を含んだシェルスクリプトの本来の処理の文脈を破壊してしまうことになる．そのような性質を持つ UNIX コマンドは Stateful に分類される．逆に，Stateful 以外の分類である Stateless，Parallerizable Pure そして Non-Parallerizable Pure の 3 つの分類に属するコマンドは，本手法に必要な入力が同一であるなら出力も同一になる性質を満たす．そして，Non-Parallerizable Pure は Stateless と Parallerizable Pure のスーパークラスである．よって，本手法を適用するためには，対象となる UNIX コマンドが少なくとも Non-Parallerizable Pure に属している必要がある．

前述のように，本手法を適用可能な UNIX コマンドは，少なくとも Non-Parallerizable Pure の性質を有している．よって，その出力は入力から一意に決定されると仮定できる．これは，入力が一定であれば，その UNIX コマンドをいつ実行しても同一の出力になるという

ことである．つまり，事前にある入力に対する出力が決定されていると言える．この性質を利用すると，事前に入力に対する出力を計算しておくことで，コマンドを実行する際に，実際にコマンドを実行することなく出力を得ることが可能である．このようにすることで，特に実行時間が長いようなコマンドについて，そのコマンド実行をすることなく出力を得ることで，その実行時間を短縮可能である．ただし，一般に，コマンドは無数に存在し，かつコマンドに対してどのような入力が与えられるかも不明である．よって，コマンドについて，事前に入力に対する出力を計算しておくことは困難であると言える．そこで，代わりに，実際にコマンドが実行される際に，その入力と出力をキャッシングしておく．これは，無数に存在するコマンドを入力の組み合わせについて事前に出力を計算しておくことに比べれば，十分に実現可能な方法である．特にコマンドの再実行時には，一度入力したデータと同じデータが再度入力となる可能性が高い．その場合，事前にキャッシュしておいた入出力の関係から，そのコマンドの出力を得ることが可能である．

このような UNIX コマンドに対する入出力のキャッシングをシェルスクリプト内の各 UNIX コマンドに適用することで，シェルスクリプト内の処理のうち，新規に計算する必要がある箇所のみを計算することが可能になる．このことは，必要な計算のみを行い，不要な再計算をスキップすることで計算を高速化されるという増分計算であると見なせる．これにより，特に，シェルスクリプト内の処理時間が長いコマンドの実行をスキップすることで，シェルスクリプト全体の処理時間の高速化が期待できる．

ただし，実際には，本手法を適用する過程で通常のシェルスクリプトの実行時には存在しないオーバーヘッドが発生する．そのようなオーバーヘッドによる処理時間の遅延が，コマンドの通常実行よりも小さい場合にのみ，本手法は効果的であると言える．場合によっては，シェルスクリプトの実行速度を高速化できないこともありえる．例えば，元から実行時間が短いコマンドに本手法を適用しても，高速化の効果は小さいか，あるいは逆に処理時間が増える可能性もある．

本手法において重要な役割を持つ UNIX コマンドの分類であるが，本手法においては，それをどのようにして判定するかは規定しない．それは，あくまで本手法をどのように実装するかに依存する．判定する方法としては，例えば，PaSh のようなコマンドアノテーションフレームワークによって機械的に決定する，ユーザが注意深く適用可能なコマンドかどうかを判断するなどが考えられる．

本論文では，本手法を CaSh と incrementalize という 2 つの UNIX コマンドとして実装している．この二つの UNIX コマンドは，ともにシェルスクリプトに増分計算を適用することを目的としている．しかし，両者は増分計算を適用する UNIX コマンドに対して期待する性質が異なる．また，想定される利用方法も異なるため，両者は単純に比較できるような関係にはない．

4.2 CaSh

CaSh は，UNIX コマンドの一つであり，本手法の実装の一つでもある．CaSh では，入力をシェルスクリプトを想定している．そして，そのシェルスクリプトに含まれる対象となる各 UNIX コマンドに対して，本手法による増分計算を適用するために入出力のキャッシュを適用していく．具体的には，シェルスクリプト内に含まれるコマンドを実行する際に，コマンドラインとそれに対する入出力を特定して，その出力をキャッシュとして保存するようスクリプトに変形を加えていく．CaSh は入力となったシェルスクリプトを実行する代わりに，変形されたスクリプトを実行する．こうすることで，シェルスクリプト内に含まれる対象となる各 UNIX コマンドは，処理を実行しながらもその入出力をキャッシングしていく．さらに，もし処理を実行する段階でその入力がキャッシュされているなら，処理を実行することなく，代わりに出力としてキャッシュから読み出したデータを返す．これにより，各 UNIX コマンドについて，キャッシュが保存されていない，つまり一度も実行していない処理についてのみ実行していくことで，シェルスクリプトに対して増分計算を適用し，過去に実行した処理の再実行をスキップできる．また，同様の処理はシェルスクリプト内の各コマンドだけでなく，パイプラインに対しても適用される．このような処理を入力となるシェルスクリプトに適用することで，その処理速度の高速化を図る．

4.3 incrementalize

incrementalize は，CaSh と同様に，UNIX コマンドの一つであり，本手法の実装の一つでもある．incrementalize 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．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．

4.3.1 キャッシュ行数

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．

第 5 章

CaSh

PaSh[4] のソースコードを修正して CaSh を実装する．

5.1 設計

PaSh は，入力となるシェルスクリプトに対して並列化を適用する．PaSh では，これをスクリプトに対する最適化の処理として実装している．そこで本論文では，PaSh のソースコードを修正して，その最適化処理時に，並列化処理の代わりに提案手法を適用するように変更する．

提案手法であるキャッシュを使ったパイプラインの増分計算は，並列化に関するコマンドの分類のうち，Stateless，Parallerizable Pure あるいは Non-Parallerizable Pure に分類されるコマンドにのみ適用可能である．入力となるシェルスクリプト内のコマンドの特定およびそれらのコマンドの並列化に関するコマンドの分類の特定に関する処理は，PaSh に元から実装されている．

提案手法では，キャッシュはファイルシステム上にファイルとして保存される．ファイルシステム上のファイルを特定するファイルパスは，そのファイルシステム上で一意である．よって，その一意性に着目し，キャッシュを一意に特定するためのキーとして利用する．キーは，キャッシュを配置するルートディレクトリへのパス，増分計算を適用するコマンドを特定するハッシュ値，そのコマンドに対する入力を特定するハッシュ値の三つの要素から構成される．キャッシュとして保存されたファイルの内容は，キャッシングしたコマンドの出力である．そのため，そのファイルの内容を出力することで，キャッシングしたコマンドの出力を，そのコマンドを再実行することなく出力可能である．

提案手法によってシェルスクリプトに増分計算を適用する場合，そのシェルスクリプト内の各コマンドのうち，提案手法を適用可能なコマンドかどうかとその入力を特定し，適用可能である場合は，そのコマンドの入力に対応するキャッシュが存在するかどうかを確認する．もしキャッシュが存在する場合は，そのコマンドの実行を，対応するキャッシュを読み出すコマンドによって置き換える．また，この処理はパイプラインにおいても，そのパイプライ

ンを構成する各コマンドに対して適用される．そして，パイプライン経由で入力を受け取るコマンドの入力を特定するために，その入力を出力するコマンドとそのコマンドに対する入力を特定するために算出した値を利用する．よって，パイプラインで結ばれた二つのコマンドは，一つのキャッシュ読み出しによって置換される．もしキャッシュが存在しない場合，そのコマンドは通常通り実行されるが，その出力はキャッシングされる．ただし，出力先にデータを出力する前に，その出力をキャッシュファイルに保存するだけで，本来に出力先には通常通り出力される．

5.2 パラメータ

環境変数 PASH_CACHE_ROOT によって，提案手法において利用するキャッシュを配置するルートディレクトリを設定する．定義されていない場合は，デフォルトではユーザのホームディレクトリ直下の.cache/pash ディレクトリにする．その他のパラメータについては，PaSh の実装を使用している都合上，PaSh のそれと同様である．ただし，PaSh における最適化処理の部分の実装を，提案手法を実装するために変更しているため，そこに関するパラメータを指定しても動作に影響を与えない．

5.3 処理

PaSh は，入力となるスクリプトを解析して，スクリプト内の各コマンドやパイプラインを，スクリプトと相互変換が可能な内部表現に変換する．その中間表現に最適化処理を適用し，再度スクリプトに書き戻す．よって，提案手法の実装では，その中間表現に対して適切な変更を適用することになる．中間表現は，各コマンドをノード，入出力関係をエッジとしたグラフである．また，中間表現では，パイプラインはシェルの制御オペレータである|ではなく，UNIX の機能の一つである名前付きパイプとシェルの制御オペレータである&を使ったバックグラウンド実行を使った表現に変換される．よって，その内部表現では，パイプラインではなく UNIX コマンドの単体実行のみを考慮すればよい．

中間表現からは，スクリプト内に含まれる各コマンドと，それぞれのコマンドに対する変数割り当て，コマンドライン引数，リダイレクト，入出力先を取得できる．キャッシュのキーを構成するためのハッシュ値の内，コマンドを特定するハッシュ値を計算するために，まず，中間表現から対象とするコマンドについて，変数割り当て，コマンドライン引数，リダイレクトを配列として取得する．これは，コマンドを実行する際のコマンドラインを構築を構築することを想定している．その配列を JSON 文字列として出力した文字列を，UTF-8 でエンコードしてから MD5 ハッシュ関数でハッシュ値に変換することで算出される．得られるハッシュ値は 128bit 長であり，これを 16 進数で表現した文字列を，キャッシュのキーを構成するための値として利用する．同様に，キャッシュのキーを構成するためのハッシュ値の内，コマンドの入力を特定するハッシュ値を計算するために，まず，入力となるファイルパ

スあるいはパイプライン経由で入力を受け取るコマンドを特定するハッシュ値をそれぞれ配列として取得する．その二つの配列を，それぞれキーとして input と command を指定した JSON 文字列に変換してから，コマンドを特定するハッシュ値の場合と同様に，UTF-8 でエンコードしてから MD5 ハッシュ関数でハッシュ値に変換することで算出される．得られるハッシュ値は 128bit 長であり，これを 16 進数で表現した文字列を，キャッシュのキーを構成するための値として利用する．

このようにして得られた二つのハッシュ値と，パラメータとして取得したキャッシュを保存するルートディレクトリを使用して，ファイルシステム上に存在するキャッシュファイルを特定するためのキーとして使用するファイルパスを構成する．ファイルパスは，**キャッシュを保存するルートディレクトリ/コマンドを特定するハッシュ値/コマンドの入力を特定するハッシュ値**のように構成される．算出されたキャッシュファイルへのファイルパスを使用して，それぞれのコマンドに対して対応するキャッシュが存在するかを確認する．キャッシュの探索は，ファイルシステム上に，算出されたハッシュ値によって構成されるファイルパスの先にキャッシュファイルが存在するかを確認することによって行われる．確認は，シェルスクリプトを変換する段階で実行される．存在する場合は，中間表現内の対応するコマンド実行の表現を，キャッシュファイルを読み出すコマンド実行の表現で置換する．置換にするためには，該当するコマンド実行を表現するノードを中間表現から削除して，新たにキャッシュファイルを読み出すコマンド実行を表現するノードを中間表現に追加する．その際，追加するキャッシュファイルの読み出しを行うコマンド実行を表現するノードに出力先は，削除したコマンド実行を表現するノードと同一にしておく．キャッシュファイルの読み出しは cat コマンドによって実行する．キャッシュが存在しない場合は，該当するコマンド実行を表現するノードの出力先を，キャッシングを行うコマンドを表現するノードを入力先として，そして該当するコマンド実行を表現するノードの元々の出力先を出力先に，それぞれ設定した名前つきパイプに変更する．キャッシングは特定されたキャッシュファイルへのファイルパスをコマンドライン引数に設定した tee コマンドによって実行する．

これらの変換は，中間表現内のコマンドを表現する各ノードに対して適用される．変換が適用された中間表現は，PaSh によって最終的にシェルスクリプトに変換される．

5.4 スクリプトの変形例

以下に，提案手法によってシェルスクリプトが変形される過程を例示する．

始めに，次のようなパイプラインを考える．

```
grep a < file_a |
  grep -i b |
  grep c
```

このパイプラインは，file_a の内容に対して，grep コマンド (`grep a`) で a が含まれる行のみを取り出す．その出力をシェルの機能であるパイプラインで次の grep コマンド (`grep -i b`) に渡し，b または B が含まれる行のみを取り出す．再度，その出力をシェルの機能であるパイプラインで次の grep コマンド (`grep c`) に渡し，c が含まれる行のみを取り出す．

このパイプラインに提案手法を適用した場合，かつこのパイプラインに含まれる各コマンドについてキャッシュが保存されていない場合，次のように変形されていく．

```
grep a < file_a |
  tee ~/.cache/pash/e29311f6f1bf/02e6c6f4596c49f9 |
  grep -i b |
  grep c
```

まず，このパイプラインの先頭のコマンドである，file_a を入力にした `grep a` に対応するキャッシュが存在するかを確認する．しかし，ここでは各コマンドのキャッシュは保存されていないため，その出力を tee コマンドでディスクに書き込みながら実行する．その出力は，通常通り後続のコマンドである `grep -i b` に渡される．次に，`grep -i b` について，file_a を入力にした `grep a` の出力を入力にした場合の出力に対応するキャッシュが存在するかを確認する．しかし，やはりここでは各コマンドのキャッシュは保存されていないため，その出力を tee コマンドでディスクに書き込みながら実行する．その後続のコマンドである `grep c` についても，同様に処理される．最終的に，このパイプラインは以下のように変形される．

```
grep a < file_a |
  tee ~/.cache/pash/e29311f6f1bf/02e6c6f4596c49f9 |
  grep -i b |
  tee ~/.cache/pash/9ffbf43126e3/25e9299ba805cccf |
  grep c |
  tee ~/.cache/pash/9a8ad92c50ca/348d7f91a81f2382
```

このパイプライン内の各 grep コマンドに対応するキャッシュは存在しない．よって，各 grep コマンドを通常通り実行するが，その出力を tee コマンドでディスクに書き込みながら，つまりキャッシングしながら実行していく．

この直後に，同様のパイプラインを実行した場合，このパイプラインは次のように変形されていく．

```
cat ~/.cache/pash/e29311f6f1bf/02e6c6f4596c49f9 |
  grep -i b |
```

```
  grep c
```

やはり，このパイプラインの先頭のコマンドである，file_a を入力にした grep a に対応するキャッシュが存在するかを確認する．今回は，直前に対応するキャッシュが存在するため，そのコマンドを，対応するキャッシュファイルの中身を出力する cat コマンドで置換する．次に，後続のコマンドである grep -i b について，file_a を入力にした grep a の出力を入力にした場合の出力に対応するキャッシュが存在するかを確認する．こちらについても，直前に対応するキャッシュが存在するため，そのコマンドを，対応するキャッシュファイルの中身を出力する cat コマンドで置換する．その後続のコマンドである grep c についても，同様に処理される．最終的に，このパイプラインは以下のように変形される．

```
cat ~/.cache/pash/9a8ad92c50ca/348d7f91a81f2382
```

このように，パイプラインを構成する各コマンドに対応するキャッシュが存在する場合，そのパイプラインは，最終的に単一のキャッシュファイルを読み出すコマンドに置換される．

ここで，先のパイプラインのキャッシュが存在する場合において，次のパイプラインを考える．

```
grep a < file_a |
  grep b |
  grep c
```

このパイプラインは，先のパイプラインのうち，二番目のコマンドである grep -i b のコマンドライン引数を変更して grep b にしたものである．このパイプラインは，次のように変形されていく．

```
cat ~/.cache/pash/e29311f6f1bf/02e6c6f4596c49f9 |
  grep b |
  grep c
```

やはりここでも，このパイプラインの先頭のコマンドである，file_a を入力にした grep a に対応するキャッシュが存在するかを確認する．依然として，直前に対応するキャッシュが存在するため，そのコマンドを，対応するキャッシュファイルの中身を出力する cat コマンドで置換する．その後，後続のコマンドである grep b について，file_a を入力にした grep a の出力を入力にした場合の出力に対応するキャッシュが存在するかを確認する．grep -i b であれば対応するキャッシュが存在するが，ここでは grep b である．よって，対応するキャッシュは存在しないため，コマンドをそのまま実行しつつ，その出力を tee コマンドに

パイプライン経由で渡してキャッシングする．同様に，後続のコマンドである `grep c` についても対応するキャッシュが存在しないため，コマンドをそのまま実行しつつ，その出力を tee コマンドにパイプライン経由で渡してキャッシングする．最終的に，パイプラインは次のように変形される．

```
cat ~/.cache/pash/e29311f6f1bf/02e6c6f4596c49f9 |
  grep b |
  tee ~/.cache/pash/ddd5752b5fe6/25e9299ba805cccf |
  grep c |
  tee ~/.cache/pash/12f54a96f644/6bffab30acbc90c4
```

5.5 PaSh へのコントリビュート

CaSh を実装するために PaSh のソースコードを読む必要があった．その過程で，タイポ修正やソースコードの実態にそぐわないコメントの削除といった軽微な修正 [14][15][16][17]，および発見したバグの修正 [18][19] を行った．修正したコードは，PaSh のリポジトリがホストされている Web サービスである GitHub のプルリクエスト機能を使って提出した．いずれの修正も既に PaSh のリポジトリ本体に取り込まれている．

第 6 章

incrementalize

incrementalize は，補助用の UNIX コマンドである 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．

6.1 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．

6.1.1 設計

chash の役割は，入力の読み込みとそのハッシュ値の計算である．そして，読み込んだ入力とハッシュ値は，chash 上では処理せずに他プロセスで処理する．そのため，それらのデータを他プロセスと共有する必要がある．これを実現するために，ファイルシステム上のファイルとシェルのパイプラインを経由したプロセス間通信を使用する．

読み込んだ入力は一行ごとにファイルシステム上の一つのファイルに書き出しておく．書き込んだ行数が事前に決定しておいた値以上になった場合は，書き出すファイルを別のファイルに変更して，再び読み込んだ入力を書き出していく．この処理は，読み込む入力が無くなるまで継続する．この処理によって，入力をいくつか一定行数ごとに分割していく．

シェルのパイプラインを経由したプロセス間通信では，読み込んだ入力から計算したハッシュ値と，それらの入力を行ごとに書き出したファイルへのパスをスペース区切りで標準出力に書き出す．ハッシュ値は，0 から 9 までの整数あるいは a から f までの 16 進数を表現する文字のみで構成される．そのため，ハッシュ値にスペースが含まれてしまうことで，ハッシュ値とファイルパスの区切りが曖昧になることはない．これは極めて単純なフォーマットであるため，この出力を受け取るプロセスは，受け取ったデータをスペースで分割するだけで，ハッシュ値とファイルへのパスを受け取り可能である．よって，JSON などのデータ交換用の構造化データフォーマットを使用したシリアライズとデシリアライズは不要である．

この処理は，先の読み込んだ入力は一行ごとにファイルシステム上の一つのファイルに書き出しておく処理において，書き込んだ行数が事前に決定しておいた値以上になった場合ごとに実行する．

以上の処理によって，chash は他プロセスに対して入力データとそのデータから計算したハッシュ値を渡すことが可能になる．

6.1.2 パラメータ

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．`lines` は任意引数であり，省略した場合の値は 4096 になる．`/path/to/tmp` は必須引数であり，省略はできない．

6.1.3 処理

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．

6.1.4 実装上の工夫

chash は C 言語によって実装されている．これは，スクリプト言語であるシェルスクリプトによる処理よりも，コンパイラ型言語である C 言語による処理のほうが単純に高速であることが期待できるためである．しかしこれは，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もある．

コマンド実行のオーバーヘッド

シェルスクリプトでハッシュ値を計算する場合，ハッシュ値を求めたい値を，ハッシュ値を計算するコマンドに渡して実行することになる．そのコマンドを実行する際に，シェルは親プロセスである自分自身を fork(2) してから，子プロセス側で exec(2) 等を実行してハッシュ値を計算するコマンドを実行する．fork(2) は，自分自身のプロセスを複製するシステムコールである．fork(2) は実行時にプロセスを複製するために，メモリ内容についても複製す

る．実装によっては，複製する際に発生するオーバーヘッドを，コピーオンライトを使用することで軽減している場合もある．ただし，子プロセス側で実行する exec(2) 等の処理は回避できない．incrementalize においては，入力は一行ないし複数行単位に分割されてキャッシュされる．そのため，それらのハッシュ値の計算をする場合，シェルスクリプトによる方法では，分割された入力の数だけハッシュ値を計算するコマンドを実行する必要がある．これは，入力行数や一つのキャッシュで扱う行数である `lines` の値が増えるほど，ハッシュ値の計算によって発生するオーバーヘッドが増大することを意味する．しかし，一つのコマンドの中でそれぞれの入力に対してハッシュ値を計算するようにすれば，ハッシュ値を計算するためのコマンド実行は，入力行数や一つのキャッシュで扱う行数である `lines` の値に関係なく 1 回で済むようになる．これにより，入力に対するハッシュ値の計算時に発生するコマンド実行のオーバーヘッドを削減できる．

シェルが扱えるデータサイズ

シェルスクリプト上で入力を読み込んでその値を保持する場合，シェル変数を使うことになる．そして，そのシェル変数に束縛されている値をコマンド等に渡す場合，シェルの変数展開を使用する．よって，シェルスクリプトで入力に対するハッシュ値を計算する際も，この変数展開を使用することになる．その際，展開されたシェル変数は，その変数に束縛されていた文字列によって置換され，シェルに解釈されることになる．このとき，展開する変数に巨大な文字列が束縛されている場合，エラーが発生することがある．変数展開によって巨大な文字列をシェルが適切に解釈できないからである．例えば，1 ギガバイト分の文字列をシェルに解釈させようとするとエラーが発生する．一般に，入力のサイズがどの程度になるかは，実際にコマンドを実行するまでわからない．しかし，incrementalize による増分計算が効果的な状況の一つに巨大な入力が与えられた場合がある．一般に，小さい入力よりも巨大な入力のほうが計算に時間がかかるからである．そういった場合に，増分計算によって一度実行した処理結果を再利用することで，処理を再実行せず処理時間を高速化できる．よって，incrementalize においては，巨大な入力は十分に想定されるべきユースケースである．だがシェルスクリプトでは，そのような巨大な入力を適切に扱うことできない．そこで，始めからそういった巨大な入力を想定した補助コマンドとして chash を作成し，上記のような状況に対応できるようにする．

ハッシュアルゴリズム

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[20] を使用する．これは，incrementalize における入力に対するハッシュ値の計算では，入力のフィンガープリントの取得が主な目的だからである．よって，できるだけ高速にハッシュ値を求めることができるだけでよく，暗号学的なハッシュ値の強度は不要である．

6.2 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．

6.2.1 設計

シェル関数 incrementalize は，コマンド出力のキャッシュをファイルシステム上のファイルとして保存する．ファイルの中身は，コマンドの出力である．また，ファイルシステム上のファイルパスは一意である．そのため，その一意性に注目して，キャッシュを特定するキーとして利用する．キーとするファイルパスは，キャッシュを保存するディレクトリのパス，キャッシュ対象のコマンドを特定する ID，入力を特定する ID の 3 つから構成される．

シェル関数 incrementalize は，実行時パラメータとして決定された，増分計算を適用するコマンドとそのコマンドを一意に特定する ID を持つ．そして，chash から入力データとそのハッシュ値を，ファイルシステムとシェルのパイプライン経由で標準入力から受け取る．chash の標準出力への出力は，各行が，分割された入力を書き込まれたファイルシステム上のファイルとそれに対するハッシュ値で構成される．よって，シェル関数 incrementalize は，標準入力から一行ずつデータを読み込んで処理していく．

読んだデータは，スペースで分割することで，分割された入力データのファイルとそのハッシュ値を取得できる．増分計算を適用可能なコマンドの出力は，そのコマンドを実行する際のコマンドラインとそのコマンドへ入力されるデータによって決定される．よって，コマンドの出力のキャッシュが存在するかは，実行時パラメータとして決定されたコマンドを一意に特定する ID と，標準入力から取得した入力のハッシュ値の 2 つをキーにして特定する．

キャッシュはファイルシステム上に，上記のキーをファイルパスの一部として保存されているため，キャッシュの存在確認は，特定のファイルパスにファイルが存在するかどうかを確認することによって実行される．ただし，上記の処理によりキャッシュが存在することが確認できたとしても，そのキャッシュが完全であるかどうかは不明である．例えば，キャッシュの作成中に処理がキャンセルされることによって，コマンドの出力の一部しか保存できていないキャッシュが作成されることが考えられる．このとき，増分計算の際に不完全なキャッシュがコマンドの出力として使用されてしまい，コマンドを通常実行した場合と比較して異なる出力になる．これを防ぐためには，存在するキャッシュが完全なものになっているかどうかを確認してから使用する必要がある．そのため，キャッシュの作成が完了したあとに，それを示すためにキャッシュごとに空のファイルを作成する．キャッシュの存在確認を行なう際，キャッシュが存在することのみを確認するのではなく，そのキャッシュに対応

する空のファイルが存在すること，空のファイルがキャッシュファイルよりも後に作成されているかどうかを確認する．この三つの条件を満たした場合にのみ，そのキャッシュは完全であり，利用可能であると判断する．適切にキャッシュが作成されたなら，空のファイルは，キャッシュファイルと比較してそれより後に作成される．これを確認するためには空のファイルとキャッシュファイルのタイムスタンプを確認すればよい．また，何らかの理由によってキャッシュファイルが変更された場合，キャッシュファイルのタイムスタンプは，空のファイルよりも先の時間を示す．そのため，2 つのファイルのタイムスタンプの比較により，不完全なキャッシュが作成された場合だけでなく，incrementalize 以外によってキャッシュファイルを変更された場合も含めて，キャッシュファイルの完全性を確認できる．

キャッシュが存在し，かつそれが完全であれば，そのキャッシュの内容を読み込んで出力する．そうでない場合，増分計算を適用する対象であるコマンドを実行する．その際，そのコマンドへの標準入力として，chash からハッシュ値と一緒に受け取った入力ファイルへのパスを指定する．そして，その出力を標準出力に書き出しながら，同時にその出力をキャッシュとしてファイルシステムにも書き出す．キャッシュを書き出す先は，前述したキャッシュを一意に特定するキーとしてのファイルパス先である．キャッシュの書き出しが完了したあとは，前述のようにキャッシュが完全であることを示すために，対応する空のファイルを作成する．

6.2.2 パラメータ

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，環境変数経由で，キャッシュファイルを保存するパスを決定する際のルートディレクトリのパスを受け取る．

6.2.3 実装

シェル関数 incrementalize は，chash の出力をシェルのパイプライン経由で標準入力から受け取ることを前提として処理を実行する．それらの入力は一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．

6.3 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．

6.3.1 設計

incrementalize は，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．なお incrementalize は，増分計算を適用するコマンドあるいはパイプラインが必要な性質を満たしていることを前提に処理する．よって，ユーザが適切に判断する，あるいは PaSh のように機械的にスクリプトを解析するなどして，コマンドが増分計算を適用するにあたって必要な性質を満たしていることを保証する必要がある．

6.3.2 実装上の工夫

並列処理

chash とシェル関数 incrementalize は，それぞれ別のコマンドとして実装されている．したがって，chash の出力をシェルのパイプラインを使ってシェル関数 incrementalize に渡すようにすることで，2 つのコマンドを容易に並列実行させることができる．

chash が実行する入力の分割およびその分割された入力ごとのハッシュ値の計算と，その結果を受けてのシェル関数 incrementalize が実行するキャッシュの探索，読み出しおよび書き込みは，論理的には，chash が入力の分割とハッシュ値計算を行い，その結果を順次キューにエンキューしていき，シェル関数 incrementalize がワーカーとしてそのキューを監視し，自身が処理を実行していない状態でキューに値がエンキューされているなら，その値をデキューして処理し，その後再びキューを監視するといった一連の処理として解釈することができる．このとき，chash とシェル関数 incrementalize は，キューを介して独立して並列に処理を実行できることがわかる．つまり，シェル関数 incrementalize が処理を実行している間に，chash が次にエンキューする値を計算することが可能であり，それによって chash による入力の分割とそのハッシュ値の計算によって発生するオーバーヘッドを緩和することが可能である．そのことから incrementalize では，そのキューとしてシェルのパイプラインを

使用して chash とシェル関数 incrementalize を通信することで，この 2 つの処理を並列に実行する．このようにパイプラインを使ったコマンドの並列実行は，シェルの基本的な機能であり，2 つのコマンドを制御オペレータである|で繋げるだけで実現可能である．他のプログラミング言語のように専用のライブラリを使用するなどして，複雑な並列実行を実装することなく，極めて簡潔に並列実行を表現かつ実装できる．

シェルスクリプトによる実装

増分計算において，実際にコマンドあるいはパイプラインを実行する部分を，別の言語で同等の処理を実現するのではなく，純粋にシェルスクリプトで実装したことで，増分計算を適用する前の文脈を破壊することなくコマンドを実行可能である．

第 7 章

実験と評価

7.1 概要

提案手法の実装である CaSh と incrementalize について，実際にシェルスクリプトを高速化できるかを，それぞれ実験を行って確認する．実際にシェルスクリプトを高速化できたかは，提案手法を適用していないと言える通常実行時における実行時間と，提案手法を適用したと言えるそれぞれのコマンドを使ってシェルスクリプトやコマンドを実行した場合を比較によって行う．もし後者のほうが高速になっていれば，シェルスクリプトやコマンドを提案手法によって高速化できたと判断する．ただし，本手法は，どのコマンドと入力についてキャッシュが作成されているかをパラメータとする計算であると言える．よって，実験に際してはそのパラメータに考慮する必要がある．ここでは，キャッシュが実験で想定されるコマンドと入力について完全に作成されている状態と，逆にまったく作成されていない状態の二つの状況を考える．前者は，提案手法上は最もシェルスクリプトの高速化の効果を期待できる状況である．逆に後者は，キャッシュ作成時のオーバーヘッドによりシェルスクリプトの実行を低速化される可能性が最も高い状況である．

7.2 CaSh

実装した CaSh を実験によって評価する．

7.2.1 実験目的

実験では，次の二つを確認する．

- CaSh によってコマンド/パイプラインを高速化できるかを確認
- コマンドの初回実行時におけるキャッシングによるオーバーヘッドの度合いを確認

7.2.2 実験内容

いくつかのスクリプト，実際には 1 つのパイプラインのみを含むいくつかのスクリプトを，そのまま実行した場合と，CaSh を使って実行した場合との実行時間を比較する．これは，キャッシュによってスクリプト実行速度が向上するかを確認するためである．また，CaSh を使って実行する場合は，キャッシュが存在する場合とそうでない場合に分けて計測する．これは，キャッシングによって発生するオーバーヘッドがどの程度のものなのかを確認するためである．

実行時間は，それぞれ 10 回実行して計測した時間の平均値を採用する．念のため，平均値以外にも，最大値，最小値，標準偏差も算出する．実行するスクリプトは，実行するパイプラインの属性，PaSh では，Stateless，Paraleliizable Pure，Non-Paralelizable Pure，Stateful の 4 つに分類されるが，ここでは ‘Stateful かそうでないかの 2 つのみ，と予測される計算時間の長さによって特徴付けられる．

ディスクキャッシュの影響を回避するために，スクリプトの実行前にキャッシュをクリアする．キャッシュにクリアは以下のコマンドによって実行する．

```
sync
echo 3 | tee /proc/sys/vm/drop_caches
```

実行するスクリプトは以下の四つである．

スクリプト 1

次のスクリプトは，Stateful に分類され，瞬時に実行が完了するパイプラインである．

```
ls -ABFohv --time-style=iso | tr 'a-z' 'A-Z'
```

スクリプト 2

次のスクリプトは，Stateful ではなく，瞬時に実行が完了するパイプラインである．

```
echo foo | tr 'a-z' 'A-Z'
```

スクリプト 3

次のスクリプトは，Stateful に分類され，ネットワークアクセスが発生するパイプラインである．

```
#!/bin/bash

FROM=2015
TO=2015
IN='http://ndr.md/data/noaa/'
fetch='curl -L -s'

seq $FROM $TO |
  sed "s;^;$IN;" |
  sed 's;$;/;' |
  xargs -r -n 1 $fetch |
  grep gz |
  tr -s ' \n' |
  cut -d ' ' -f9 |
  sed 's;^\(.*\)\(20[0-9][0-9]\).gz;\2/\1\2\.gz;' |
  sed "s;^;$IN;" |
  head -n144 |
  xargs -n1 $fetch |
  gunzip |
  cut -c 89-92 |
  grep -v 999 |
  sort -rn |
  head -n1
```

スクリプト 4

次のスクリプトは，Stateful ではなく，ある程度巨大な入力を受け取るパイプラインである．

```
#!/bin/bash

grep -v '[1-6]' <random_number_100GB.txt |
  sed -E 's/^0+//g' |
  wc
```

表 7.1 実験環境 1

OS	GNU/Linux (5.18.9-arch1-1)
CPU	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
メモリ	(16GB, LPDDR3)
ディスク	WDS500G2B0B (500GB, SATA III, SSD)

表 7.2 スクリプト 1 についての実験結果

条件	平均 (sec)	最大値 (sec)	最小値 (sec)	標準偏差 (sec)
通常実行	0.0412	0.045	0.037	0.00274064
提案手法	0.4669	0.613	0.362	0.0648203

7.2.3 実験環境

実験で使用した環境のスペックを表 7.1 に示す.

7.2.4 実験結果

スクリプト 1

実行するパイプラインは stateful であるため，その出力はキャッシュされない．そのため，ここでは CaSh を使用した実行をキャッシュの有無で区別しない.

実験結果を表 7.2 に示す．PaSh を使用した場合の実行時間が，PaSh なしで実行した場合と比較して，平均時間で 11 倍以上 (11.332524 = 0.4669/0.0412) になっている．これは，変換前のスクリプトに含まれるコマンドが一瞬で実行が完了するために，PaSh によるスクリプトの解析と変換によって発生したオーバーヘッドが，元の処理時間と比較して相対的に大きくなるからである．また，変換前のスクリプトに含まれるコマンドが Stateful であるために，その出力がキャッシュされない．よって，そのようなスクリプトでは PaSh を使用する意味はない.

スクリプト 2

実験結果を表 7.3 に示す．CaSh をキャッシュありで使用した場合の実行時間が，CaSh なしで実行した場合と比較して，平均時間で 13 倍以上 (13.99403 = 0.4688/0.0335) になっている．これは，スクリプト 1 と同様に，変換前のスクリプトに含まれるコマンドが一瞬で実行が完了するために，CaSh によるスクリプトの解析と変換によって発生したオーバーヘッドが，元の処理時間と比較して相対的に大きくなるからである．このスクリプトに含まれるコマンドは，スクリプト 1 と異なり，Paraleliizable Pure であるため，CaSh によってその

表 7.3 スクリプト 2 についての実験結果

条件	平均 (sec)	最大値 (sec)	最小値 (sec)	標準偏差 (sec)
通常実行	0.0335	0.039	0.022	0.00479004
提案手法/キャッシュなし	0.4783	0.532	0.417	0.0411422
提案手法/キャッシュあり	0.4688	0.542	0.372	0.0585279

表 7.4 スクリプト 3 についての実験結果

条件	平均 (sec)	最大値 (sec)	最小値 (sec)	標準偏差 (sec)
通常実行	413.099	505.101	328.822	50.6157
提案手法/キャッシュなし	414.215	456.012	357.75	34.2353
提案手法/キャッシュあり	0.5104	0.559	0.449	0.0352427

出力がキャッシュされる.

スクリプト 3

実験結果を表 7.4 に示す. スクリプトに含まれるコマンドがネットワークアクセスをするためか, 結果のばらつきが大きい. CaSh をキャッシュありで使用した場合の実行時間が, CaSh なしで実行した場合と比較して，平均時間で 800 倍以上 (809.363245 = 413.099/0.5104) になっている．これは，CaSh によるスクリプトの解析と変換によって，最終的に実行されるスクリプトが，ただ cat でキャッシュを読み出すだけになるからである．このスクリプトについては，CaSh によるキャッシングの効果が顕著に表われている．また，CaSh をキャッシュなしで使用した場合の実行時間が，CaSh なしで実行した場合と比較して，平均時間でほぼ差がない (1.002702 = 414.215/413.099)．CaSh によるコマンド出力のキャッシングでは，ディスクへの書き込みが発生する．これは，CaSh によるスクリプトの解析と変形と同様に，CaSh を使用しない場合には発生しないオーバーヘッドである．だが，前述の結果を見ると，ディスクへの書き込みのオーバーヘッドは無視できるほど十分に小さい.

スクリプト 4

実験結果を表 7.5 に示す．スクリプトに含まれるコマンドがローカルのファイルシステムにアクセスするだけであるためか，結果のばらつきが，スクリプト 4 に比べて小さい．CaSh をキャッシュありで使用した場合の実行時間が，CaSh なしで実行した場合と比較して，平均時間で 530 倍以上 (539.133081 = 327.739/0.6079) になっている．これは，スクリプト 3 と同様に，CaSh によるスクリプトの解析と変換によって，最終的に実行されるスクリプトが，ただ cat でキャッシュを読み出すだけになるからである．このスクリプトについても，CaSh によるキャッシングの効果が顕著に表われている．また，CaSh をキャッシュなしで使

表 7.5 スクリプト 4 についての実験結果

条件	平均 (sec)	最大値 (sec)	最小値 (sec)	標準偏差 (sec)
通常実行	327.739	344.153	323.659	6.03161
提案手法/キャッシュなし	328.945	335.727	322.734	4.88641
提案手法/キャッシュあり	0.6079	0.749	0.547	0.0616287

用した場合の実行時間が，CaSh なしで実行した場合と比較して，平均時間でほぼ差がない (1.00368 = 328.945/327.739)．これもスクリプト 3 と同様に，ディスクへの書き込みのオーバーヘッドは無視できるほど十分に小さいと言える．

7.2.5 実験のまとめ

元々の実行時間がある程度長いスクリプトであれば，CaSh によるキャッシングの効果があることが確認できた．一方で，元々の実行時間が極めて短いスクリプトの場合，CaSh によるスクリプトの解析と変換によるオーバーヘッドが実行時間の大半を占めてしまい，むしろ実行時間が遅くなることを確認できた．そして，少なくとも SSD であれば，CaSh でのコマンドの初回実行時のキャッシュの書き込みが実行時間に与える影響は，十分に小さく無視できる程度のものであることが確認できた．

7.3 incrementalize

実装した incrementalize を実験によって評価する．

7.3.1 実験目的

実験では，次の三つを確認する．

- incrementalize によってコマンド/パイプラインを高速化できるかを確認
- キャッシュ行数の値が incrementalize の実行時間にどう影響を与えるかを確認
- ワーストケース (実行時点でキャッシュが一切存在しない場合) でのオーバーヘッドの度合いを確認

7.3.2 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 incrementalize 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 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．

```
sync
echo 3 | tee /proc/sys/vm/drop_caches
```

また，各コマンドの標準出力を /dev/null にリダイレクトして，出力の描画のオーバーヘッドを排除する．

入力データは，以下のシェルスクリプトによって現在の作業ディレクトリ内の inputs ディレクトリに生成される．

```
mkdir -pv inputs
cd inputs

curl -sfL 'http://pac-n4.csail.mit.edu:81/pash_data/small/unix50.zip' >
↪  unix50.zip
unzip unix50.zip

for small in small/*; do
  cat "$small" "$small" "$small" "$small" "$small" >"${small##*/}"
done
```

表 7.6 に実行する UNIX コマンド/コマンドライン及び入力ファイル名とそのファイルサイズ，行数を示す．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．

表 7.6: 実行するコマンド/コマンドライン及びメタデータ

No.	Command/Command Line	Input	Filesize (Byte)	Lines
1	`cut -d ' ' -f 2`	1.txt	1073754000	85159800
2	`fmt -w 1`	10.txt	1074566000	13561000
3	`sed 's/[^[:alnum:]]/ /g'`	8.txt	1075953000	5114500
4	`grep '[opq]'`	11.txt	1091209300	8233600
5	`tr a-z A-Z`	2.txt	1073770500	25311000
6	`cut -d ' ' -f 4 | tr -d ','`	2.txt	1073770500	25311000
7	`cut -d ' ' -f 2 | cut -c 1-1 | tr -d '\n ' | tr '[A-Z]' '[a-z]'`	3.txt	1073745200	73209900
8	`tr ' ' '\n ' | grep '\.'`	4.txt	1073778000	2334300
9	`tr ' ' '\n ' | grep 'x' | grep '\.'`	4.txt	1073778000	2334300
10	`tr ' ' '\n ' | grep 'x' | grep '\.' | cut -d '.' -f 2 | grep -v '[KQRBN]'`	4.txt	1073778000	2334300
11	`tr ' ' '\n ' | grep 'x' | grep '\.' | cut -d '.' -f 2 | grep '[KQRBN]' | cut -c 1-1`	4.txt	1073778000	2334300
12	`tr ' ' '\n ' | grep 'x' | grep '\.' | cut -d '.' -f 2 | cut -c 1-1 | tr '[a-z]' 'P'`	4.txt	1073778000	2334300
13	`tr ' ' '\n ' | grep '\.' | cut -d '.' -f 2 | cut -c 1-1 | tr '[a-z]' 'P'`	4.txt	1073778000	2334300
14	`grep 'print' | cut -d "\"" -f 2 | cut -c 1-12`	5.txt	1073744900	104755600
15	`awk "{print $2, $0}" | cut -d ' ' -f 2`	6.txt	1073758400	82201600
16	`cut -f 1 | grep AT&T`	7.txt	1073743200	28933200
17	`tr ' ' '\n ' | grep 1969`	8.txt	1075953000	5114500

18	grep 'Bell' | awk 'length <= 45' | cut -d ',' -f 2 | awk "{\$1=\$1};1"	8.txt	1075953000	5114500
19	grep '(' | cut -d '(' -f 2 | cut -d ')' -f 1	8.txt	1075953000	5114500
20	tr -c "[a-z][A-Z]" '\n ' | awk "length >= 16"	8.txt	1075953000	5114500
21	tr ' ' '\n ' | grep '[A-Z]' | tr '[a-z]' '\n ' | grep '[A-Z]' | tr -d '\n ' | cut -c 1-4	9.1.txt	1073743200	59652400
22	cut -c 1-1 | tr -d '\n '	9.2.txt	1073749600	48806800
23	cut -c 1-2 | tr -d '\n '	9.3.txt	1073742400	58835200
24	tr ' ' '\n ' | grep "\"" | sed 4d | cut -d "\"" -f 2 | tr -d '\n '	9.4.txt	1073750100	46182800
25	tr ' ' '\n ' | grep '[A-Z]' | sed 1d | sed 3d | sed 3d | tr '[a-z]' '\n ' | grep '[A-Z]' | sed 3d | tr -c '[A-Z]' '\n ' | tr -d '\n '	9.6.txt	1073753600	33554800
26	sed 2d | sed 2d | tr -c '[A-Z]' '\n ' | tr -d '\n '	9.7.txt	1073748100	39403600
27	tr -c '[a-z][A-Z]' '\n ' | grep '[A-Z]' | sed 1d | sed 2d | sed 3d | sed 4d | tr -c '[A-Z]' '\n ' | tr -d '\n '	9.8.txt	1073750100	46182800
28	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 '	9.9.txt	1073747200	31580800
29	sed 1d | grep 'Bell' | cut -f 2	10.txt	1074566000	13561000
30	grep 'UNIX' | cut -f 1	11.txt	1091209300	8233600

1 つのキャッシュで扱う入力の行数は以下の通りである.

1. $2^{10} = 1024$

表 7.7 実験環境 2

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)

2. $2^{12} = 4096$
3. $2^{14} = 16384$
4. $2^{16} = 65536$
5. $2^{18} = 262144$
6. $2^{20} = 1048576$
7. $2^{22} = 4194304$
8. $2^{24} = 16777216$
9. $2^{26} = 67108864$
10. $2^{28} = 268435456$
11. $2^{30} = 1073741824$

7.3.3 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 7.7 に示す．

7.3.4 実験結果

実行した各 UNIX コマンドあるいはパイプラインごとに，通常実行，キャッシュあり，キャッシュなりの場合の平均実行時間の比較を図 7.1 から図 7.30 に示す．これらのよると，通常実行時と比較して，キャッシュする行数にもよるが，概ねキャッシュありの場合の実行時間は大差がないかあるいはかなり高速になっっていることがわかる．ただ，全 30 件のなかで明確に高速になった例は，図 7.3，図 7.21，図 7.25，図 7.27，図 7.28 の 5 件のみである．

ここで，個別の結果に注目してみる．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 7.3 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 7.4 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 7.28 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 7.8 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．単純に考えれば，キャッシュなしの場合の実行時間は，通常実行よりも遅くなる．だが，その想定に合致しない結果も散見される．これは，実験に使用した入力が，特定のデータの複製をいくつも連結したものによって構成されているからであると思われる．それにより，キャッシュがない状態で増分計算を適用したとしても，途中の計算において，それ以前の計算中に作成されたキャッシュにヒットするということが発生し得る．その場合，キャッシュなしという条件であっても，途中からキャッシュありと同様の増分計算が適用される．そして，その増分計算が有効に作用した結果，例えキャッシュなしの場合でも通常実行よりも実行速度が高速になることが期待できる．

一部の結果では，キャッシュなしの場合と比べて，キャッシュありの場合のほうが実行速度が遅くなることがある．そのほか，図 7.4 では，キャッシュありの場合でも，通常実行と比較して明確に低速になっている．これは，増分計算を適用するコマンド/パイプラインが非常に高速に動作するために，増分計算の適用によって発生するオーバーヘッドが通常実行時の実行時間を上回ることが原因であると考えられる．

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あるいはパイプラインにおいても，一度にキャッシュする行数は多いほうがが処理速度が低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

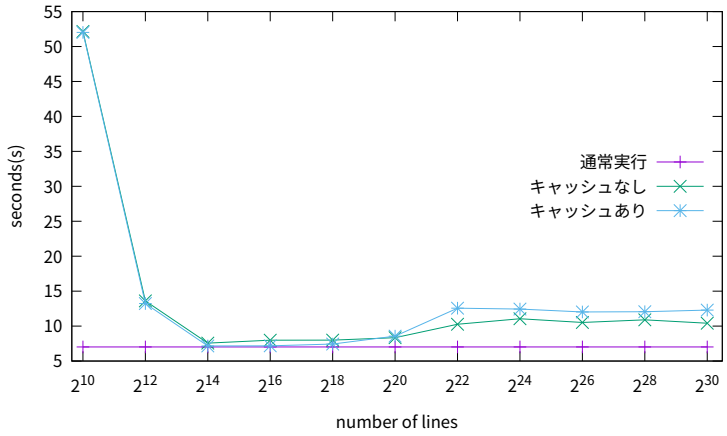


図 7.1 test-unix50-1-cut

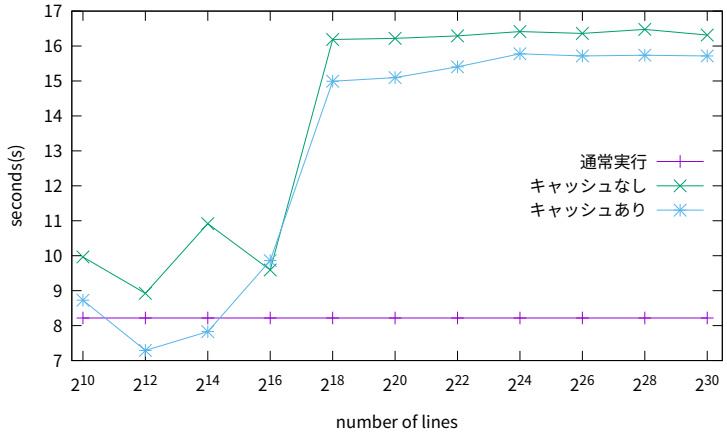


図 7.2 test-unix50-10-fmt

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．前述のように，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とすると，例えキャッシュが再利用されず，新規に作成されたとしても少なくともその処理時間に与える影響は少ない．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．

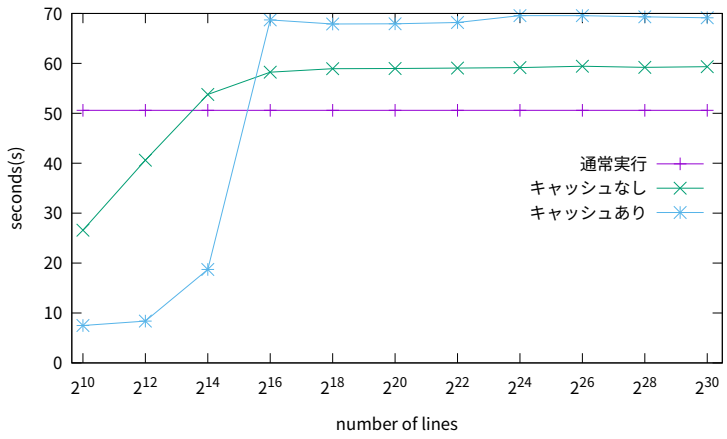


図 7.3 test-unix50-8-sed

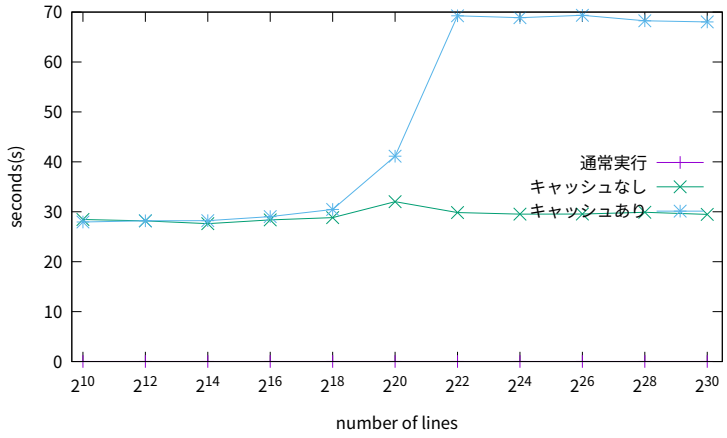


図 7.4 test-unix50-11-grep

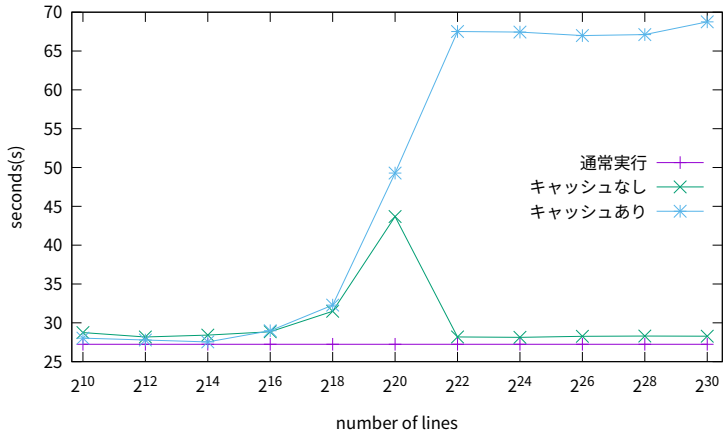


図 7.5 test-unix50-2-tr

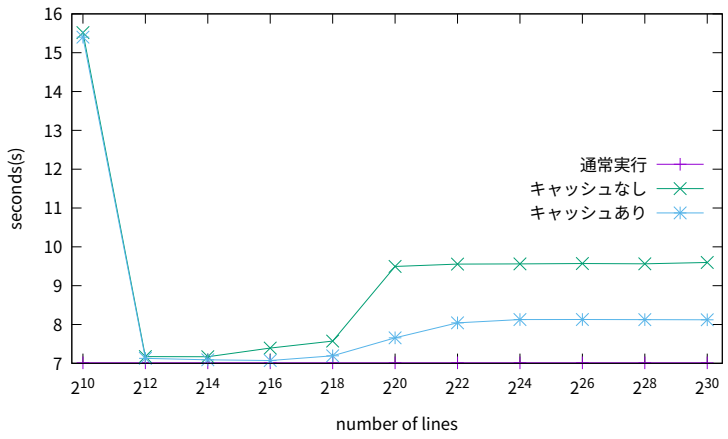


図 7.6 test-unix50-5

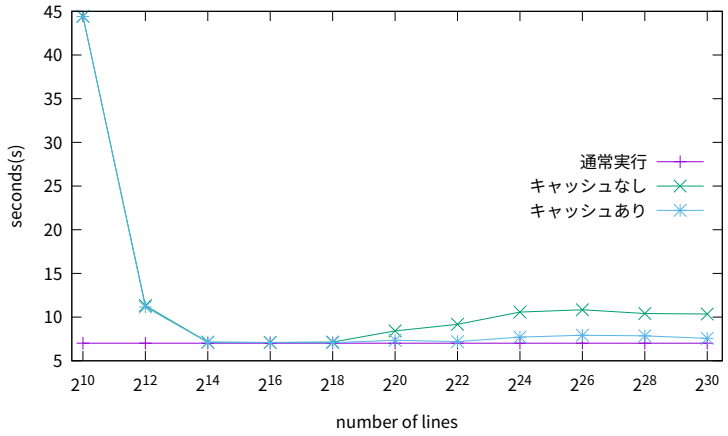


図 7.7 test-unix50-6

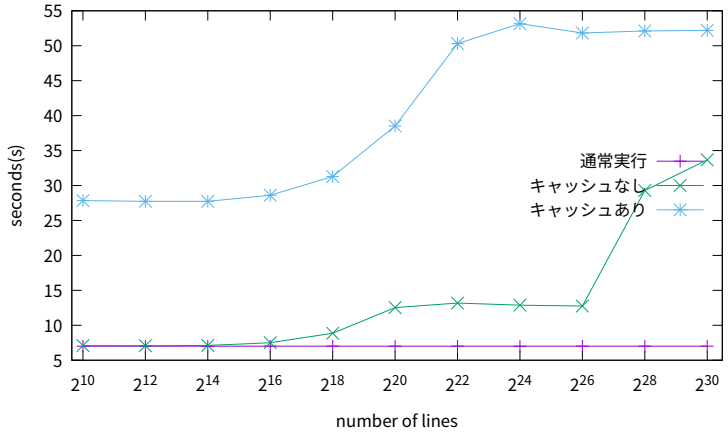


図 7.8 test-unix50-7

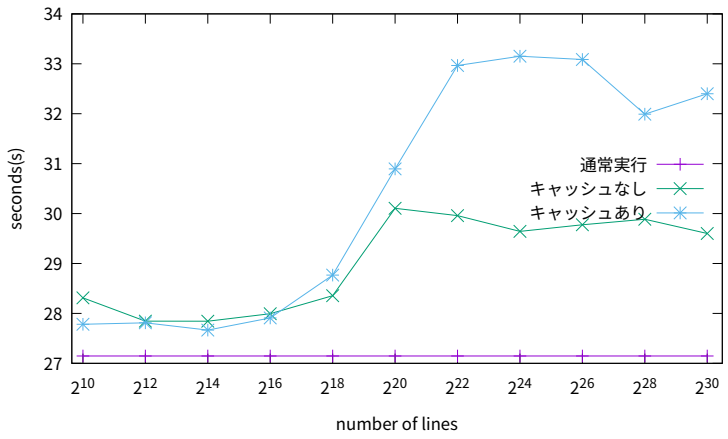


図 7.9 test-unix50-8

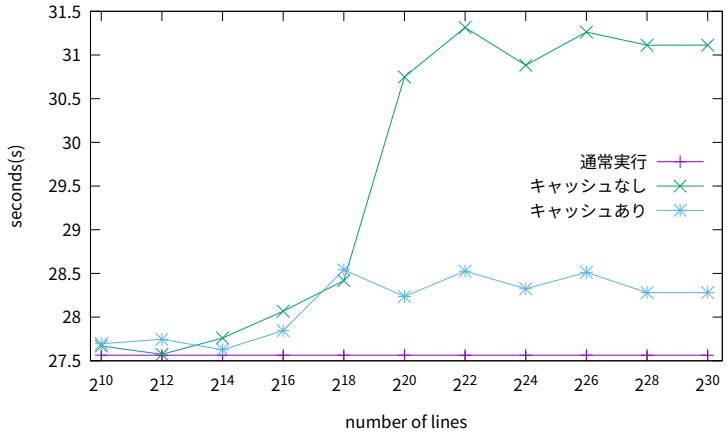


図 7.10 test-unix50-9

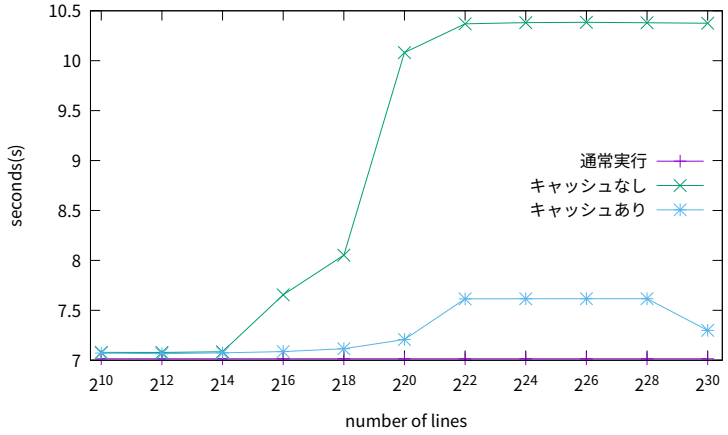


図 7.11 test-unix50-10

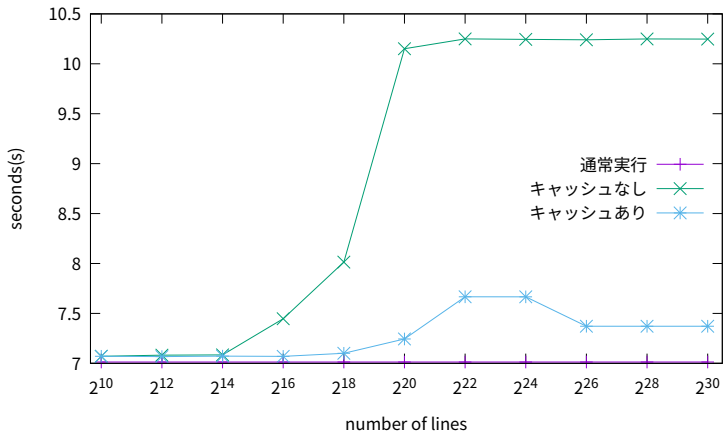


図 7.12　test-unix50-11

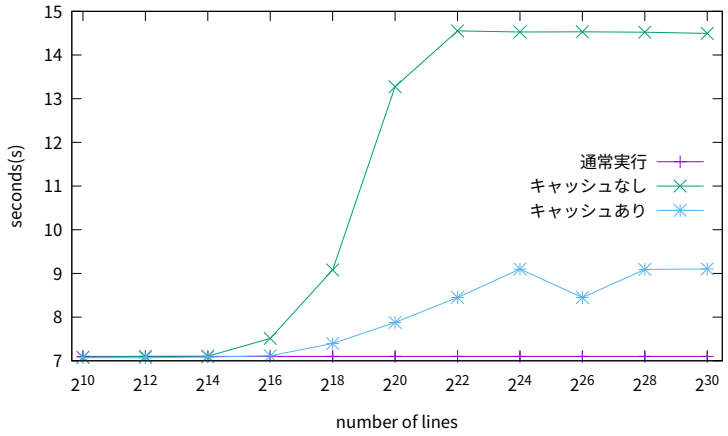


図 7.13　test-unix50-12

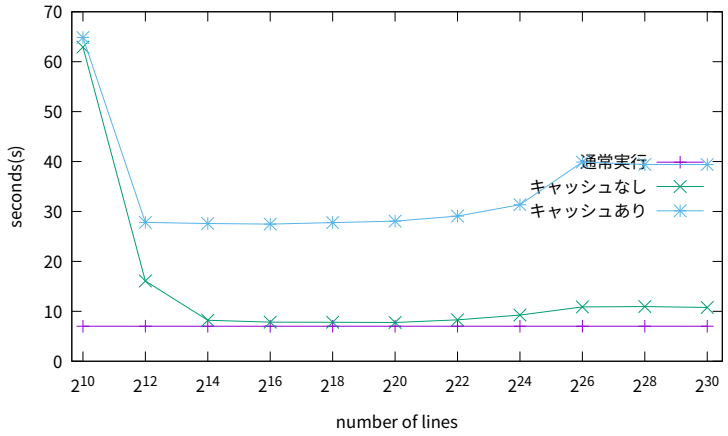


図 7.14　test-unix50-13

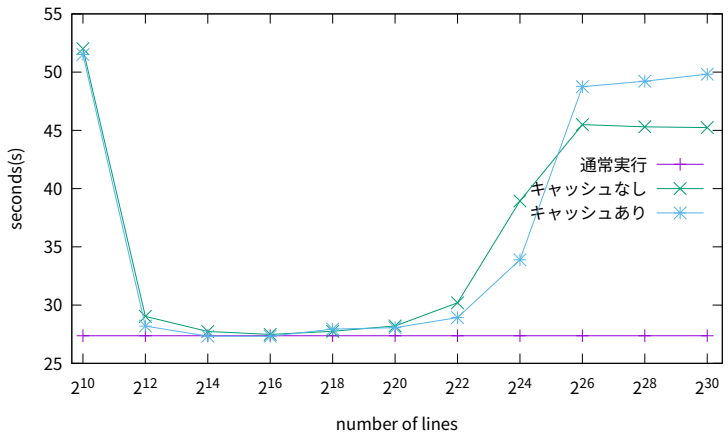


図 7.15 test-unix50-14

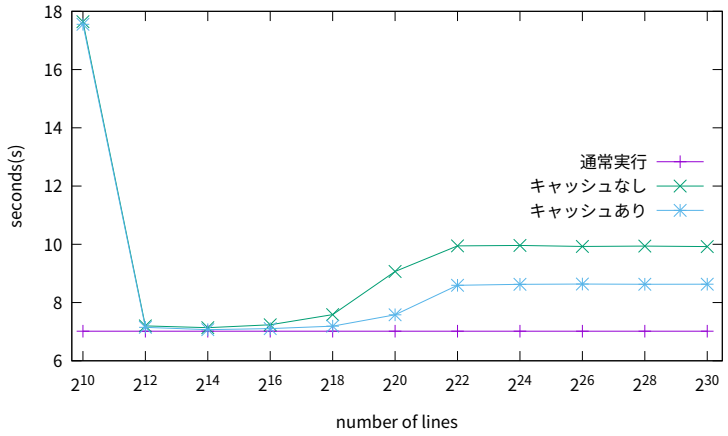


図 7.16 test-unix50-15

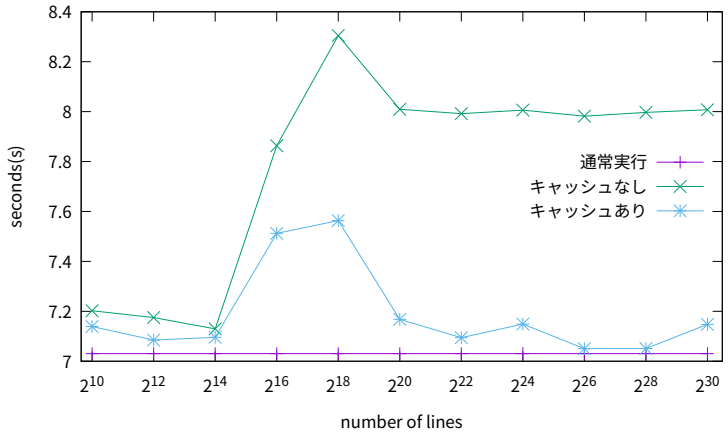


図 7.17 test-unix50-18

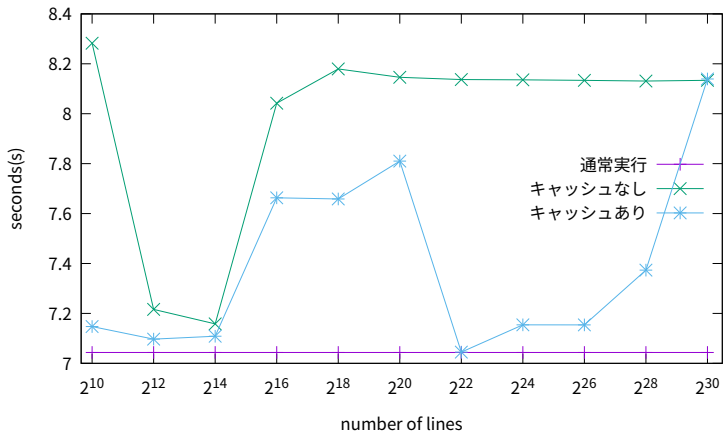


図 7.18 test-unix50-19

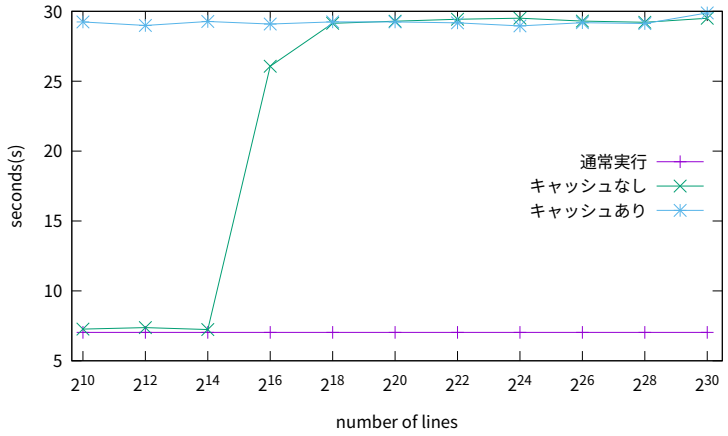


図 7.19 test-unix50-20

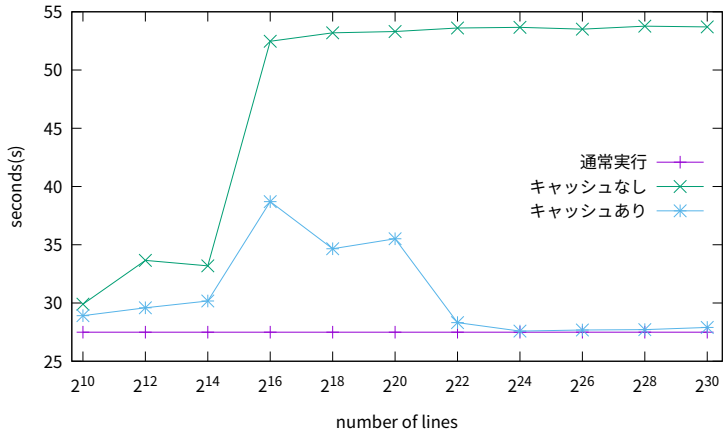


図 7.20 test-unix50-21

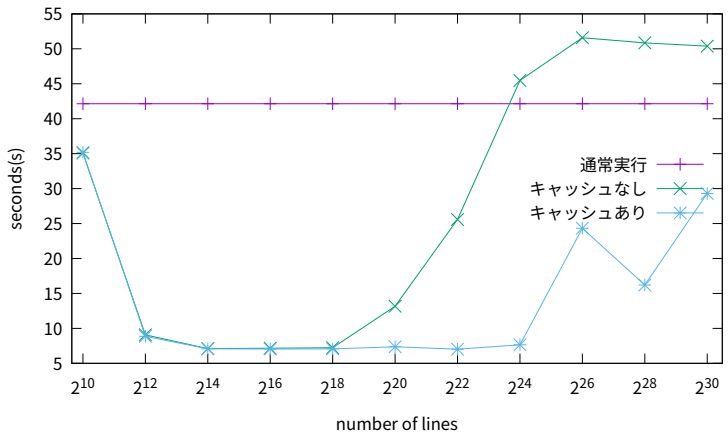


図 7.21 test-unix50-23

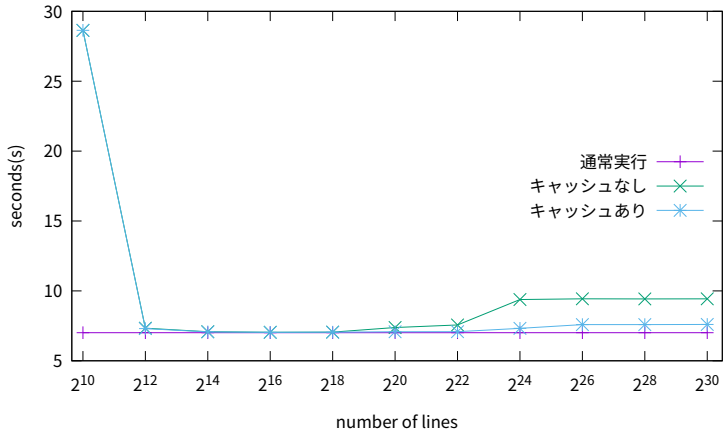


図 7.22 test-unix50-24

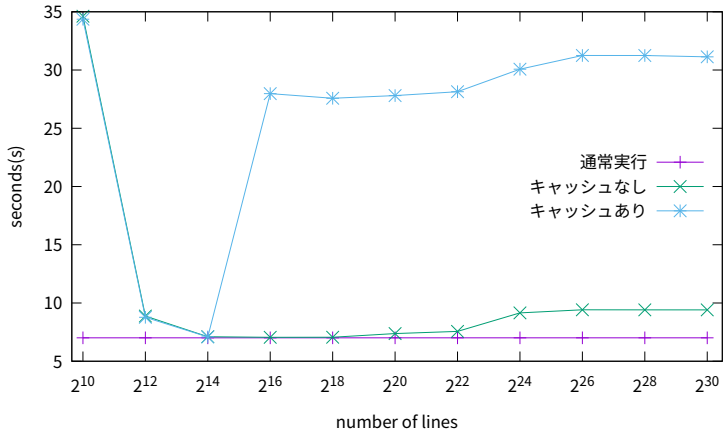


図 7.23 test-unix50-25

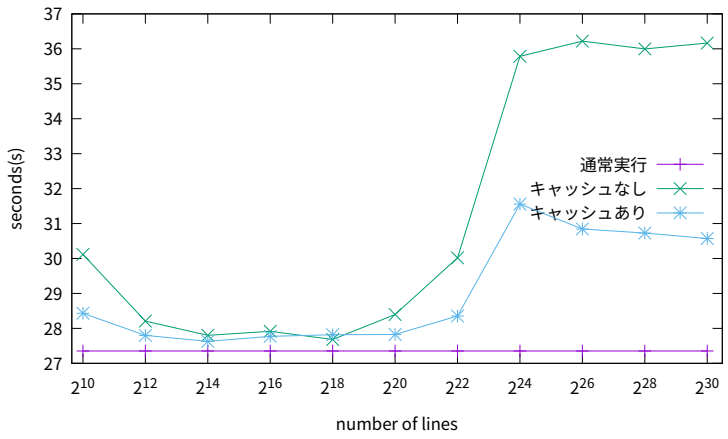


図 7.24 test-unix50-26

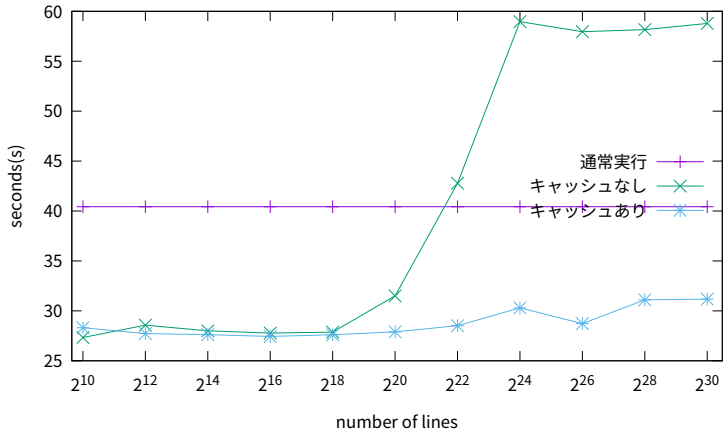


図 7.25 test-unix50-28

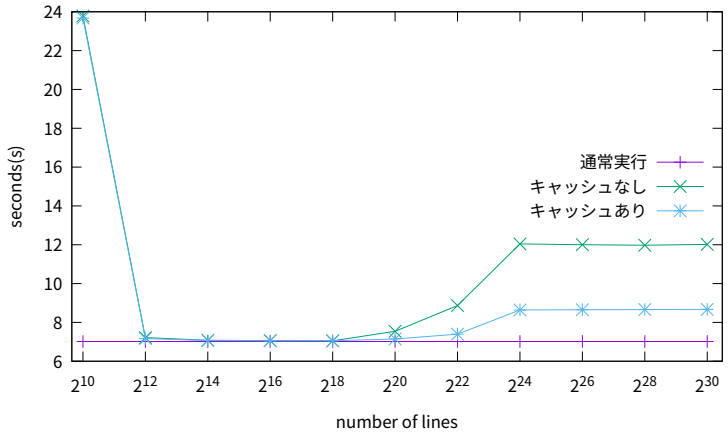


図 7.26 test-unix50-29

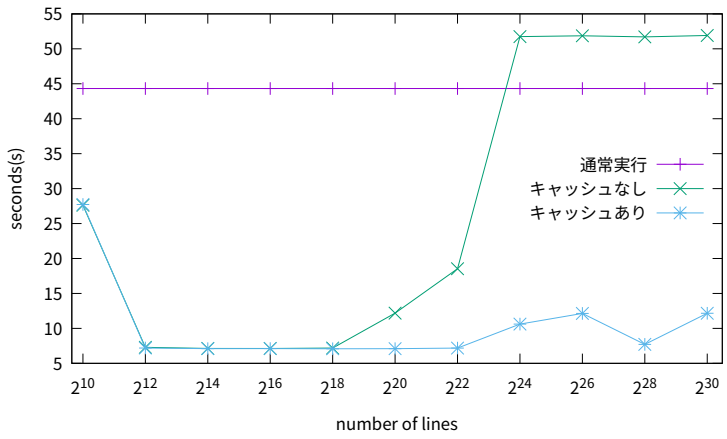


図 7.27 test-unix50-30

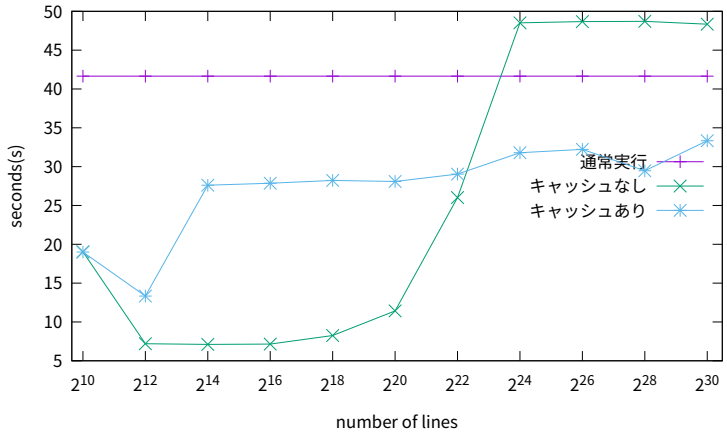


図 7.28 test-unix50-31

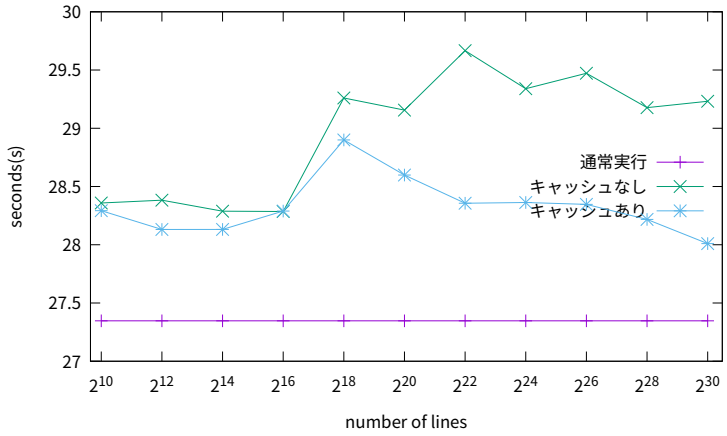


図 7.29 test-unix50-33

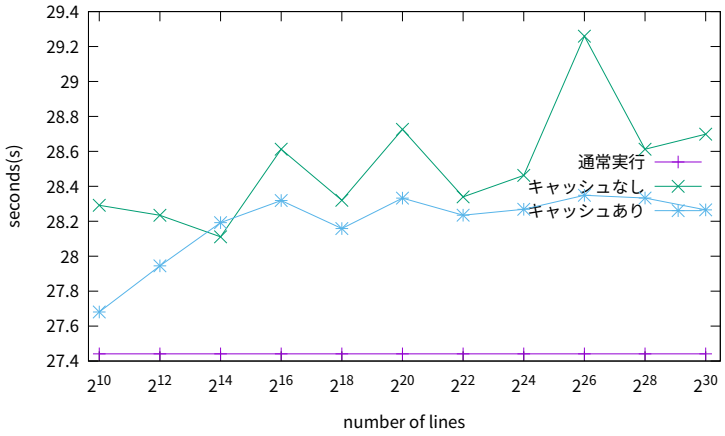


図 7.30 test-unix50-35

7.3.5 実験のまとめ

incrementalize により，シェルスクリプトを高速化可能であることを確認できた．しかし，明確に高速化できたのは計 30 個のスクリプトのうち，5 つに留まった．それ以外については，通常実行と提案手法を適用した場合とで，あまり差がないか，逆に明確に低速化する例があった．ただし，incrementalize によって極端にシェルスクリプトの処理速度が低下する例はあまりなかった．

キャッシュ行数については，全体的な傾向として少ないほうが処理速度が高速になった．逆に，キャッシュ行数が多いほど，処理速度は低速になる傾向にあった．よって，キャッシュ行数は少ないほうが incrementalize によるシェルスクリプトの処理速度の向上させる効果が見込めることがわかった．

ワーストケース (実行時点でキャッシュが一切存在しない場合) でのオーバーヘッドがシェルスクリプトの処理速度に与える影響は，それほど影響がない例が多かった．ただし，キャッシュ行数が多いと，キャッシュが一切存在しない場合に incrementalize を使用した場合の処理速度が低速化する傾向にあった．

7.4 実験で考慮しなかった事項

提案手法について，実験で考慮しなかったいくつかの事項について述べる．

7.4.1 キャッシュファイルのガベージコレクション

CaSh と incrementalize では，ファイルシステム上にキャッシュを保存している．CaSh と incrementalize が実装している現行のアルゴリズムでは，キャッシュファイルが際限無く

単調に増加していく．すると，ディスク容量の圧迫やファイルシステムにおける管理ファイルの増加によるファイルの探索性能の低下が発生することが考えられる．また，特に SSD を使用している場合，ディスク容量の圧迫に起因するディスク IO 性能の低下が発生することも考えられる．よって，所謂ガベージコレクション (以後 GC) のように，不要なキャッシュファイルを削除するような仕組みを導入することが考えられる．ただし，それによって追加の処理が発生するため，そのオーバーヘッドが CaSh と incrementalize の実行に与える影響を考慮する必要がある．ただ，必ずしも CaSh と incrementalize の実行時に GC を実行する必要はない．そのため，CaSh と incrementalize のキャッシュファイルの GC を担当するデーモンプロセスを起動しておいて，バックグランドで処理させることが考えられる．

7.4.2 増分計算の適用対象の指定

CaSh は，現行の実装では，パイプライン内も含む対象となる全てのコマンドに対して増分計算を適用する．しかし，本提案手法は，その性質上コマンドの処理が重い，つまりはコマンド実行に時間がかかるものに対してより高い効果が期待できる．逆に，コマンドの処理が軽い，つまりはコマンド実行にあまり時間がかからないものについては，通常実行のみでよく，増分計算を適用する必要性が薄い．例えば，入力を恒等写像するようなコマンドである `cat` の場合，入力の処理にはほとんど時間がかからないため，増分計算の適用対象外としても問題ない．このように必要性が少ないあるいは高速化が期待できないコマンドについては増分計算の適用対象外とすることで，その分のキャッシュファイルの作成や管理が必要なくなる．それにより，7.4.1 で述べたキャッシュファイルが増大する問題を緩和することが期待できる．

そのような増分計算の選択的適用を実現するためには，事前知識を活用する方法と，ユーザが増分計算の適用範囲を明示することが考えられる．PaSh は事前知識として UNIX コマンドの並列性に関する分類を利用していた．同様に，各 UNIX コマンドについて，どれくらい処理が重いかあるいは処理時間がかかるかという観点による分類を作成しておく．そして，入力スクリプトを解析して増分計算を適用する際，これまでの単に増分計算を適用可能かどうかの基準だけでなく，コマンドの重さに関する基準も考慮して，増分計算を適用していく．それにより，増分計算を，高速化が期待できるコマンドに対してのみ適用可能になる．そのほか，入力スクリプトに事前にユーザが増分計算の適用に関する指示コメントのようなものを記述しておき，CaSh がそれを読み取り増分計算の適用時に利用することが考えられる．これにより，ユーザがソースコードを修正する必要はあるが，事前知識だけでは対処できないような，ローカルな事情にもとづく増分計算の適用範囲の指示が可能になる．

7.4.3 一般化と最適化

本論文では，特に CaSh について，評価実験における実験対象となるスクリプトが 4 件とあまり多くない．そのため，確認された CaSh の増分計算の効果がより多くのスクリプトに対してどこまで発揮されるかは不透明である．この問題を解決するためには，より多くのスクリプトを対象として性能評価を行い，CaSh によるスクリプトの高速化の効果の有無を一般化する必要がある．また，今回の実験では，キャッシュが完全に作成されている場合か，キャッシュがまったく存在しない場合の両極端の状況についてのみ実験を行った．だが，一般には，一部のコマンドの一部の入力についてのみキャッシュが作成されている状況が考えられる．そのような状況のおける性能評価を行うことも，CaSh による増分計算の効果を一般化する上で重要である．そのほか，本論文では主に実行時間の平均値に注目して CaSh と incrementalize の評価を行った．データの分散が小さいの場合にはあまり問題にならないが，一部の実験結果については，分散が小さくないものもある．そのような場合も含め，平均値だけでなく分布や高次特徴量の分析を行うことで，より正確に CaSh と incrementalize の実行時間の傾向を確認できる．なお，本論文における評価実験では試行回数を 10 回にしていたが，分布や高次特徴量の分析を行うためには，試行回数を増やし，より多くのデータを収集する必要がある．

本論文では，CaSh と incrementalize の実行時間にしか注目しなかった．そのため，処理のボトルネックの分析や最適化ができていない．本提案手法の実装である CaSh と incrementalize は，それぞれキャッシュをファイルシステムの保存している．よって，ファイル IO やディスク IO といったより低レイヤの要因がボトルネックになることが考えられる．しかしながら，それらが処理においてどの程度の影響があるかは未確認である．そのような要因を分析と特定およびアルゴリズムや実装の改善といった最適化によって，本提案手法や実装の処理速度の向上が期待できる．

第 8 章

おわりに

8.1 まとめ

本論文では，シェルスクリプトにキャッシュを使って増分計算を適用する手法を提案した．そして，提案手法の実装として CaSh と incrementalize という二つの UNIX コマンドを実装した．CaSh は，純粋な計算を実行する UNIX コマンドやパイプラインを含むシェルスクリプトを対象に増分計算を適用して処理速度を高速化する UNIX コマンドである．incrementalize は，純粋な計算のなかでも，行指向な計算を実行する UNIX コマンドあるいはパイプラインを対象として増分計算を適用して処理速度を高速化する UNIX コマンドである．そして，この二つのコマンドを，実際に対象となる UNIX コマンドやパイプラインを高速化できるかどうかを，実験によって評価した．その結果，元々の実行時間が長いようなスクリプトであれば，CaSh によるキャッシングによって実行速度を高速化できることを確認できた．incrementalize については，対象となる UNIX コマンドやパイプラインを高速化できることを確認できた．しかし，逆に低速化させてしまう例も確認された．以上より，提案手法によってシェルスクリプトを高速化できることを確認できた．しかし，必ずしも全ての対象となる UNIX コマンドやシェルスクリプトを高速化できるわけではない．また，元々十分に高速な UNIX コマンドやシェルスクリプトについては，提案手法を適用する過程で発生するオーバーヘッドにより，逆に低速化する可能性もある．よって，提案手法を UNIX コマンドやシェルスクリプトに適用する際には，大量の入力データを扱ったり，処理に時間がかかるような，高速化の効果が見込めるかどうかを考慮することが重要である．

8.2 今後の展望

本論文では，提案手法は UNIX コマンドやパイプラインの入出力のキャッシングを行っていたが，特に出力について，暗に標準出力のみを扱っていた．これをより一般化してファイルを扱えるようにすると，増分計算の適用対象をさらに拡大できる．そうすると，本手法をより汎用的な増分計算を適用するためのフレームワークとして利用できるようになる．増分

計算を個別のツールに実装する必要がなくなるため，ツールの実装をよりシンプルに保つことができる．これは，いわゆる UNIX 哲学における “スモール・イズ・ビューティフル” や “一つのプログラムには一つのことをうまくやらせる” といった諸定理 [21] にも即している．

謝辞

論文執筆にあたって，ご指導いただきました新美礼彦教授，修士論文審査委員として，コメントおよびアドバイスをいただきました髙橋信行教授，松原克弥教授に深く感謝いたします．

The authors would like to think Enago (`www.enago.jp`) for the English Language review.

発表・採録実績

発表等

[1] 佐藤 碧，新美 礼彦，“行指向なコマンド/パイプラインへの増分計算の適用”，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2022-HPC-187，No. 10，pp. 1–7，2022 年 12 月 1 日，(予稿集のみ)

[2] 佐藤 碧, 新美 礼彦, “シェルスクリプトにおける純粋なコマンドに対する入出力キャッシングによる増分計算の適用”，情報処理学会 第 85 回全国大会　〜ダイバーシティと情報処理〜，(2023 年 3 月 2 日–4 日，発表予定)

[3] 佐藤 碧，新美 礼彦，“行指向なコマンド/パイプラインへの増分計算の適用”，第 188 回 ハイパフォーマンスコンピューティング研究発表会，(2023 年 3 月 16 日–17 日，発表予定，口頭発表のみ)

参考文献

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

[2] M. Chevalier-Boisvert, N. Gibbs, J. Boussier, *et al.*, "Yjit: A basic block versioning jit compiler for cruby," in *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 25–32, ISBN: 9781450391092. DOI: 10.1145/3486606.3486781. [Online]. Available: https://doi.org/10.1145/3486606.3486781.

[3] M. L. Nelson, "A survey of reverse engineering and program comprehension," *CoRR*, vol. abs/cs/0503068, 2005. arXiv: cs/0503068. [Online]. Available: http://arxiv.org/abs/cs/0503068.

[4] N. Vasilakis, K. Kallas, K. Mamouras, A. Benetopoulos, and L. Cvetković, "Pash: Light-touch data-parallel shell processing," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 49–66.

[5] M. Greenberg, K. Kallas, and N. Vasilakis, "Unix shell programming: The next 50 years," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 104–111, ISBN: 9781450384384. DOI: 10.1145/3458336.3465294. [Online]. Available: https://doi.org/10.1145/3458336.3465294.

[6] D. Raghavan, S. Fouladi, P. Levis, and M. Zaharia, "POSH: A Data-Aware shell," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 617–631, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/raghavan.

[7] K. Brazil, *Cli tool and python library that converts the output of popular command-line tools, file-types, and common strings to json, yaml, or dictionaries. this allows piping of output to tools like jq and simplifying automation scripts.* 2019-2022. [Online]. Available: https://github.com/kellyjonbrazil/jc.

[8] G. Ramalingam and T. Reps, "A categorized bibliography on incremental computation," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93, Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 502–510, ISBN: 0897915607. DOI: 10.1145/158511.158710. [Online]. Available: https://doi.org/10.1145/158511.158710.

[9] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89, Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 315–328, ISBN: 0897912942. DOI: 10.1145/75277.75305. [Online]. Available: https://doi.org/10.1145/75277.75305.

[10] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster, "Adapton: Composable, demand-driven incremental computation," *SIGPLAN Not.*, vol. 49, no. 6, pp. 156–166, Jun. 2014, ISSN: 0362-1340. DOI: 10.1145/2666356.2594324. [Online]. Available: https://doi.org/10.1145/2666356.2594324.

[11] S. Erdweg, M. Lichter, and M. Weiel, "A sound and optimal incremental build system with dynamic dependencies," *SIGPLAN Not.*, vol. 50, no. 10, pp. 89–106, Oct. 2015, ISSN: 0362-1340. DOI: 10.1145/2858965.2814316. [Online]. Available: https://doi.org/10.1145/2858965.2814316.

[12] S. Handa, K. Kallas, N. Vasilakis, and M. C. Rinard, "An order-aware dataflow model for parallel unix pipelines," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473570. [Online]. Available: https://doi.org/10.1145/3473570.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[14] a5ob7r, *Fix typo in a comment*, 2022. [Online]. Available: https://github.com/binpash/pash/pull/511 (visited on 01/29/2023).

[15] a5ob7r, *Add a missing gitignore target*, 2022. [Online]. Available: https://github.com/binpash/pash/pull/556 (visited on 01/29/2023).

[16] a5ob7r, *Remove an outdated todo comment*, 2022. [Online]. Available: https://github.com/binpash/pash/pull/580 (visited on 01/29/2023).

[17] a5ob7r, *Remove an invalid trailing comma in an annotation*, 2022. [Online]. Available: https://github.com/binpash/pash/pull/605 (visited on 01/29/2023).

[18] a5ob7r, *Fix the eager runtime intermediate filenames*, 2022. [Online]. Available: https://github.com/binpash/pash/pull/526 (visited on 01/29/2023).

[19] a5ob7r, *Fix field splittings by missing double quotations*, 2022. [Online]. Available: https://github.com/binpash/pash/pull/550 (visited on 01/29/2023).

[20] Y. Collet, *Xxhash: Extremely fast hash algorithm*, 2016. [Online]. Available: https://github.com/Cyan4973/xxHash.

[21] M. Gancarz, *UNIX* **という考え方: その設計思想と哲学**. オーム社, 2001, ISBN: 9784274064067. [Online]. Available: https://books.google.co.jp/books?id=5XzfIACY0G8C.

[22] IEEE and T. O. Group, *Utility conventions*, 2001-2018. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/basedefs/V1_chap12.html (visited on 01/22/2023).

図目次

7.1 test-unix50-1-cut . . . 41
7.2 test-unix50-10-fmt . . . 41
7.3 test-unix50-8-sed . . . 42
7.4 test-unix50-11-grep . . . 42
7.5 test-unix50-2-tr . . . 42
7.6 test-unix50-5 . . . 43
7.7 test-unix50-6 . . . 43
7.8 test-unix50-7 . . . 43
7.9 test-unix50-8 . . . 44
7.10 test-unix50-9 . . . 44
7.11 test-unix50-10 . . . 44
7.12 test-unix50-11 . . . 45
7.13 test-unix50-12 . . . 45
7.14 test-unix50-13 . . . 45
7.15 test-unix50-14 . . . 46
7.16 test-unix50-15 . . . 46
7.17 test-unix50-18 . . . 46
7.18 test-unix50-19 . . . 47
7.19 test-unix50-20 . . . 47
7.20 test-unix50-21 . . . 47
7.21 test-unix50-23 . . . 48
7.22 test-unix50-24 . . . 48
7.23 test-unix50-25 . . . 48
7.24 test-unix50-26 . . . 49
7.25 test-unix50-28 . . . 49
7.26 test-unix50-29 . . . 49
7.27 test-unix50-30 . . . 50
7.28 test-unix50-31 . . . 50

7.29 test-unix50-33 50
7.30 test-unix50-35 51

B.1 cut コマンドについての平均実行時間の比較 81
B.2 fmt コマンドについての平均実行時間の比較 82
B.3 sed コマンドについての平均実行時間の比較 82
B.4 grep コマンドについての平均実行時間の比較 82
B.5 tr コマンドについての平均実行時間の比較 83

表目次

3.1 各 UNIX コマンドの並列化可能性の分類 10

7.1 実験環境 1 33
7.2 スクリプト 1 についての実験結果 33
7.3 スクリプト 2 についての実験結果 34
7.4 スクリプト 3 についての実験結果 34
7.5 スクリプト 4 についての実験結果 35
7.6 実行するコマンド/コマンドライン及びメタデータ 37
7.7 実験環境 2 39

B.1 実行する UNIX コマンドとそのコマンドライン及び入力ファイルと行数 . . 73
B.2 実験環境 3 74
B.3 cut コマンドについての実験結果 1 75
B.4 fmt コマンドについての実験結果 1 75
B.5 sed コマンドについての実験結果 1 76
B.6 grep コマンドについての実験結果 1 76
B.7 tr コマンドについての実験結果 1 77
B.8 cut コマンドについての実験結果 2 77
B.9 fmt コマンドについての実験結果 2 78
B.10 sed コマンドについての実験結果 2 78
B.11 grep コマンドについての実験結果 2 79
B.12 tr コマンドについての実験結果 2 79

C.1 test-unix50-1-cut-without-incrementalize 84
C.2 test-unix50-1-cut-with-incrementalize-without-cache 84
C.3 test-unix50-1-cut-with-incrementalize-with-cache 85
C.4 test-unix50-10-fmt-without-incrementalize 85
C.5 test-unix50-10-fmt-with-incrementalize-without-cache 85
C.6 test-unix50-10-fmt-with-incrementalize-with-cache 86

C.7 test-unix50-8-sed-without-incrementalize 86
C.8 test-unix50-8-sed-with-incrementalize-without-cache 86
C.9 test-unix50-8-sed-with-incrementalize-with-cache 87
C.10 test-unix50-11-grep-without-incrementalize 87
C.11 test-unix50-11-grep-with-incrementalize-without-cache 87
C.12 test-unix50-11-grep-with-incrementalize-with-cache 88
C.13 test-unix50-2-tr-without-incrementalize 88
C.14 test-unix50-2-tr-with-incrementalize-without-cache 88
C.15 test-unix50-2-tr-with-incrementalize-with-cache 89
C.16 test-unix50-5-without-incrementalize 89
C.17 test-unix50-5-with-incrementalize-without-cache 89
C.18 test-unix50-5-with-incrementalize-with-cache 90
C.19 test-unix50-6-without-incrementalize 90
C.20 test-unix50-6-with-incrementalize-without-cache 90
C.21 test-unix50-6-with-incrementalize-with-cache 91
C.22 test-unix50-7-without-incrementalize 91
C.23 test-unix50-7-with-incrementalize-without-cache 91
C.24 test-unix50-7-with-incrementalize-with-cache 92
C.25 test-unix50-8-without-incrementalize 92
C.26 test-unix50-8-with-incrementalize-without-cache 92
C.27 test-unix50-8-with-incrementalize-with-cache 93
C.28 test-unix50-9-without-incrementalize 93
C.29 test-unix50-9-with-incrementalize-without-cache 93
C.30 test-unix50-9-with-incrementalize-with-cache 94
C.31 test-unix50-10-without-incrementalize 94
C.32 test-unix50-10-with-incrementalize-without-cache 94
C.33 test-unix50-10-with-incrementalize-with-cache 95
C.34 test-unix50-11-without-incrementalize 95
C.35 test-unix50-11-with-incrementalize-without-cache 95
C.36 test-unix50-11-with-incrementalize-with-cache 96
C.37 test-unix50-12-without-incrementalize 96
C.38 test-unix50-12-with-incrementalize-without-cache 96
C.39 test-unix50-12-with-incrementalize-with-cache 97
C.40 test-unix50-13-without-incrementalize 97
C.41 test-unix50-13-with-incrementalize-without-cache 97
C.42 test-unix50-13-with-incrementalize-with-cache 98

C.43 test-unix50-14-without-incrementalize 98
C.44 test-unix50-14-with-incrementalize-without-cache 98
C.45 test-unix50-14-with-incrementalize-with-cache 99
C.46 test-unix50-15-without-incrementalize 99
C.47 test-unix50-15-with-incrementalize-without-cache 99
C.48 test-unix50-15-with-incrementalize-with-cache 100
C.49 test-unix50-18-without-incrementalize 100
C.50 test-unix50-18-with-incrementalize-without-cache 100
C.51 test-unix50-18-with-incrementalize-with-cache 101
C.52 test-unix50-19-without-incrementalize 101
C.53 test-unix50-19-with-incrementalize-without-cache 101
C.54 test-unix50-19-with-incrementalize-with-cache 102
C.55 test-unix50-20-without-incrementalize 102
C.56 test-unix50-20-with-incrementalize-without-cache 102
C.57 test-unix50-20-with-incrementalize-with-cache 103
C.58 test-unix50-21-without-incrementalize 103
C.59 test-unix50-21-with-incrementalize-without-cache 103
C.60 test-unix50-21-with-incrementalize-with-cache 104
C.61 test-unix50-23-without-incrementalize 104
C.62 test-unix50-23-with-incrementalize-without-cache 104
C.63 test-unix50-23-with-incrementalize-with-cache 105
C.64 test-unix50-24-without-incrementalize 105
C.65 test-unix50-24-with-incrementalize-without-cache 105
C.66 test-unix50-24-with-incrementalize-with-cache 106
C.67 test-unix50-25-without-incrementalize 106
C.68 test-unix50-25-with-incrementalize-without-cache 106
C.69 test-unix50-25-with-incrementalize-with-cache 107
C.70 test-unix50-26-without-incrementalize 107
C.71 test-unix50-26-with-incrementalize-without-cache 107
C.72 test-unix50-26-with-incrementalize-with-cache 108
C.73 test-unix50-28-without-incrementalize 108
C.74 test-unix50-28-with-incrementalize-without-cache 108
C.75 test-unix50-28-with-incrementalize-with-cache 109
C.76 test-unix50-29-without-incrementalize 109
C.77 test-unix50-29-with-incrementalize-without-cache 109
C.78 test-unix50-29-with-incrementalize-with-cache 110

C.79 test-unix50-30-without-incrementalize 110
C.80 test-unix50-30-with-incrementalize-without-cache 110
C.81 test-unix50-30-with-incrementalize-with-cache 111
C.82 test-unix50-31-without-incrementalize 111
C.83 test-unix50-31-with-incrementalize-without-cache 111
C.84 test-unix50-31-with-incrementalize-with-cache 112
C.85 test-unix50-33-without-incrementalize 112
C.86 test-unix50-33-with-incrementalize-without-cache 112
C.87 test-unix50-33-with-incrementalize-with-cache 113
C.88 test-unix50-35-without-incrementalize 113
C.89 test-unix50-35-with-incrementalize-without-cache 113
C.90 test-unix50-35-with-incrementalize-with-cache 114

付録 A

試作型 incrementalize

A.1 概要

試作型 incrementalize は，本文中の incrementalize と同様に，本提案手法の実装の一つである．本文中の incrementalize，この試作型 incrementalize の実装によって発覚した問題点を解決した改良型 incrementalize である．両者の違いは，あくまで別の実装という点のみであり，シェルスクリプトに対して増分計算を適用する際の戦略は基本的には同様である．なお，特に断りがなければ，本文中の incrementalize は改良型 incrementalize を指している．

試作型 incrementalize は，改良型 incrementalize とは異なり，シェルスクリプトにおける単一のコマンドを対象にして増分計算を適用する．これは，試作型 incrementalize が，CaSh による増分計算のように，シェルスクリプトにおける単一のコマンドに対して適用されることを想定していたからである．そのため，試作型 incrementalize を使ってパイプラインに増分計算を適用するのなら，パイプラインを構成する各 UNIX コマンドに対して個別に試作型 incrementalize を適用していくことで，間接的にパイプライン全体に増分計算を適用することになる．ただし，試作型 incrementalize は，改良型 incrementalize と同様に，あくまで指定された UNIX コマンドが，必要な性質である Stateless を満していることを前提にして処理するだけの UNIX コマンドである．増分計算を適用する対象である UNIX コマンドが必要な性質である Stateless を満しているかどうかを判別する機構や，シェルスクリプトを入力として受け取って，それの含まれる各 UNIX コマンドやパイプラインに対して増分計算を適用するような機構を持たない．試作型 incrementalize は，これまた改良型 incrementalize と同様に，そういった処理をする機構と組み合せて利用されることを前提としている．

A.2 設計

試作型 incrementalize は，コマンドライン引数あるいは環境変数から必要なパラメータを取得して，対象となる UNIX コマンドに増分計算を適用する．このとき，前述のように，増分計算を適用する対象である UNIX コマンドが必要な性質を満していることを前提として処

理する．そのため，なんらかの方法で，例えば，ユーザが適切に判断する，あるいは PaSh のように機械的にコマンドの性質を特定する，などといった方法で，前述の性質を満していることを保証する必要がある．

試作型 incrementalize は，標準入力からデータを一行ずつ読み出していく．このとき，読み出したデータはメモリ上に保持しておく．そのようにして標準入力から読み出した行の総数が，事前に決定しておいた値以上になるか，それ以上読めなくなった場合，それまで読んだ全入力行のハッシュ値を計算する．それによって得られたハッシュ値と，事前に取得しておいた増分計算を適用する UNIX コマンドを特定するハッシュ値をもとにして，対応するキャッシュが存在するかを確認する．キャッシュが存在する場合，そのキャッシュを読み出すことで，UNIX コマンドを実行することなくその出力を返す．キャッシュが存在しない場合，それまでに読んだ全入力行を標準入力として対象となる UNIX コマンドを実行しながら，その出力をキャッシュしながら標準出力に出力していく．いずれかに処理を実行したあと，それまでに読んだ入力行とその総数をリセットしてから，再び標準入力からデータを読み出していく．これを，標準入力からデータを読めなくなるまで続ける．

A.3 パラメータ

試作型 incrementalize におけるパラメータは，増分計算を適用するコマンド `words` と，`words` を一意に特定する値 `words_key`，コマンド出力のキャッシュを保存するディレクトリのパス`/path/to/cachedir`，一度のキャッシュする入力行数の上限 `lines` の 4 つである．`words` と `words_key` は必須引数としてコマンドライン引数として指定する．`lines` は環境変数 `BUFFER_LINE_SIZE` として指定できる．環境変数 `BUFFER_LINE_SIZE` が宣言されていない場合は，`lines` は 4096 が指定されたものとして処理する．`/path/to/cachedir` は，環境変数 `INCREMENTALIZE_CACHE_ROOT` として指定できる．環境変数 `INCREMENTALIZE_CACHE_ROOT` が宣言されていない場合は，`/path/to/cachedir` は`~/tmp/incrementalize` が指定されたものとして処理する．

A.4 処理

A.4.1 コマンドライン引数

試作型 incrementalize は，増分計算を適用するコマンド `words` を，コマンドライン引数として受け取る．その際，増分計算を適用するコマンドを構成するコマンドラインの各トークンは単一の引数としてではなく，個別の引数として渡されることを想定している．これは，前述のように，試作型 incrementalize は増分計算を適用する対象が，単体の UNIX コマンドのみだからである．このように個別の引数として受け取るようにしてインターフェースに制限を設けることで，強引にパイプラインを渡すといったことが難しくなる．具体的には，次

の場合が単一の引数として渡される場合である．

```
incrementalize -- 'grep -i a'
```

一方で，次の場合が個別の引数として渡される場合である．

```
incrementalize -- 'grep' '-i' 'a'
```

なお，引数--以降の引数は，厳密に引数として解釈され，オプションおよびその引数とは解釈されなくなる．この仕様は，POSIX のコマンドラインの文法におけるガイドライン [22] に規定されているが，あくまでも実装に依存する．

A.4.2 キャッシュファイル

試作型 incrementalize によって作成されるコマンド出力のキャッシュは，シェル関数 incrementalize(6.2) の実装と同様に，ファイルシステム上にファイルとして保存される．また，そのキャッシュファイルのファイルパスやその内容が持つ役割も同様である．ファイルシステム上のファイルの論理的な位置を特定するファイルパスは，一つのファイルシステム上で一意である．その一意性を利用して，コマンド出力のキャッシュを特定するためのキーとして利用する．キーとなるファイルパスは，キャッシュを保存するディレクトリのパス，増分計算を適用する UNIX コマンドを一意に特定するハッシュ値，入力内容を特定するハッシュ値の 3 つから構成される．

A.4.3 入力の読み込み

試作型 incrementalize は，標準入力からの入力を Bash の組み込みコマンドである `mapfile` あるいは `readarray` で読み込む．なお，`readarray` は `mapfile` のエイリアスである．この組み込みコマンドは，標準入力等から複数行を読み込んでシェル変数に配列として格納する．また，`-n` オプションによって，読み込む行数の上限を指定することができる．ただし，この組み込みコマンドはデータを読み込むことができなくなってもエラーを返さない．つまり，この組み込みコマンドの実行だけでは，それ以上データを読み込むことができなくなったかどうかを判断することができないということである．よって，読み込んだデータを格納したシェル変数に束縛されている配列の要素数を確認して，1 以上であるなら読み込みが成功として処理を継続し，そうでないならこれ以上読み込むことが不可能であると判断して処理を終了する．

A.4.4 キャッシュの読み書き

シェル変数に配列として束縛されている読み込まれた入力データは，配列の各要素を改行コードで連結した文字列として，xxHash アルゴリズム [20] による 64bit 長のハッシュ値を計算するコマンドである xxh64sum に入力として渡される．それによって計算された入力データのハッシュ値と，事前に取得している UNIX コマンドを特定するハッシュ値，キャッシュを保存するディレクトリへのパスから，対象となるキャッシュファイルを特定する．特定したキャッシュファイルが存在するなら，それをコマンドの出力として cat コマンドで出力する．存在しないなら，入力データを格納した配列の各要素を改行コードで連結した文字列を入力として，対象となる UNIX コマンドを実行する．その際，その出力を，算出されたキャッシュファイルへのパスを引数とした tee コマンドにパイプ経由で渡す．tee コマンドはその出力を引数として渡されたファイルパス先のファイルに書き込みながら，同時に標準出力にも書き出す．これらの処理は，標準入力から読み込んだデータの行数が，事前に指定された一定の値になるか，あるいはデータを読み込むことが無くなるたびに繰り返し処理を実行される．

A.5 実装上の工夫

A.5.1 入力データの読み込み

試作型 incrementalize は，Bash の組み込みコマンドである `mapfile` を使って標準入力から入力データを読み込む．この組み込みコマンドは，一度のコマンド実行で標準入力から複数行を読み込むことが可能である．これに対して，POSIX に定義されている `read` は，一回のコマンド実行で一行を読むことしかできない．あるいは，区切り文字を改行文字以外に設定することで，複数行を読み込むことは可能であるが，行数を指定して読み込むことができない．これを実現するためには，そのような処理をシェルスクリプトによって記述する必要がある．しかし，シェルスクリプトによって記述された処理は，インタプリタがスクリプトを解釈する必要がある都合上，組み込みコマンド一つを実行することに比べて，低速になりがちである．そこで，Bash の組み込みコマンドである `mapfile` を使って標準入力からデータを読み込む処理をできるだけ高速に実行できるようにする．

A.5.2 入力データの保持

読み込まれた入力データはシェル変数としてメモリ上に保持される．提案手法においては，入力データを特定するためのハッシュ値を計算し，また出力のキャッシュが存在しない場合は，再度その入力データを入力としてコマンドを実行する必要がある．このように，同じ入力データを最大二回利用する可能性があり，当然そのデータは両方の利用において同一の内

容でなければならない．例えば，入力のソースがファイルであり，且つその内容が不変であることを仮定できるのなら，入力を保持せず，単に二回入力を読めば済む．しかし，一般にそのような仮定は成立しない．また，入力のソースがパイプラインを経由した別のコマンドの出力である場合もある．その場合，二回入力を読むためにはコマンドを二度実行する必要がある．同様のコマンドを複数回実行するのは，単純にコマンドの実行時間が二倍になるため効率が悪い．理論的には，提案手法による増分計算によって二度目以降のコマンド実行はキャッシュファイルの読み込みに置換される．だが，そのためのオーバーヘッドは発生することは考慮する必要がある．

そこで，一度読み込んだ入力データをシェル変数としてメモリ上に保持しておく．こうすることで，同一のデータを参照でき，ファイルの読み書きやコマンド実行によるオーバーヘッドも削減できる．シェル変数として保持されるため，シェルスクリプトで実装された試作版 incrementalize で扱うことが容易でもある．

A.5.3 ハッシュ値計算

入力データの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[20] による 64bit 長のハッシュ値を計算するコマンドである xxh64sum を使用する．これは 6.1.4 の場合と同様に，できるだけ高速に入力データのフィンガープリントを求めることが目的であるためである．そのために，暗号学的ハッシュアルゴリズムと比べると計算コストが小さいことが期待できる非暗号学的ハッシュアルゴリズムであり，かつそのなかでも高速な xxHash を使用する．

付録 B

試作型/改良型 incrementalize の比較

B.1 実験目的

シェルスクリプトを使用して実装した試作版 incrementalize と，そのうちの入力のハッシュ値計算を chash によって高速化した改良型 incrementalize の性能を比較する．これにより，改良型 incrementalize が試作版 incrementalize と比較してどれだけ高速になったかを確認する．また，それぞれの incrementalize を使ってシェルスクリプトにおいてキャッシュを利用した複数行単位の増分計算を実現する上で，一度にキャッシュする行数の違いが，増分計算のパフォーマンスにどのような影響を与えるのかを調査する．

B.2 実験内容

試作版 incrementalize と改良型 incrementalize を用いて，いくつかの UNIX コマンドに増分計算を適用して実行する．このとき，1 つのキャッシュで扱う入力の行数の違いが，増分計算を適用した際の UNIX コマンドの実行時間に与える影響を確認するために，1 つのキャッシュで扱う入力の行数をパラメータとする．よって，事前に決定しておいた行数をそのパラメータとして指定してそれぞれの incrementalize を利用して各 UNIX コマンドに増分計算を適用し，その処理時間を計測する．なお，改良型 incrementalize は，UNIX コマンドだけではなく，パイプラインに対しても増分計算を適用可能である．しかし，試作型 incrementalize は，UNIX コマンドに対してしか増分計算を適用できない．よって，それぞれの incrementalize で増分計算を適用する対象を合わせるために，本実験では UNIX コマンドのみを対象とする．

処理時間は，Bash の予約語である time によるコマンド実行から終了までの経過時間を使って計測する．試作版 incrementalize と改良型 incrementalize について，対象となるそれぞれの UNIX コマンドと 1 つのキャッシュで扱う入力の行数の組み合わせについて 10 回計測を行い，それらの平均時間を処理時間とする．実際には，time コマンドの出力は標準エラー出力に出力されるため，それぞれの incrementalize を実行する際に，標準エラー出力を

ファイルにリダイレクトして記録し，それを生データとする．

増分計算はキャッシュの有無を状態に持つ状態つき計算であると考えられる．そのため，incrementalize による増分計算の処理時間を公平に計測するためには，キャッシュの作成状況を考慮する必要がある．よって，本実験では，incrementalize による UNIX コマンドへの増分計算の適用の前に，事前に対象となるそれぞれの UNIX コマンドと 1 つのキャッシュで扱う入力の行数の組み合わせについて，出力のキャッシュを作成しておく．その出力は，実験で想定される入力全てについて完全に作成される．キャッシュの作成は，対象となるそれぞれの UNIX コマンドと 1 つのキャッシュで扱う入力の行数の組み合わせについて，事前にそれぞれの incrementalize を実行しておくことで行う．

各コマンド実行前に，ディスクキャッシュの影響を排除するために，ディスクキャッシュを開放しておく．キャッシュにクリアは以下のコマンドによって実行する．

```
sync
echo 3 | tee /proc/sys/vm/drop_caches
```

また，各コマンドの標準出力を/dev/null にリダイレクトして，出力の描画のオーバーヘッドを排除する．

表 B.1 に実行する実行する UNIX コマンドとそのコマンドライン及び入力ファイルと行数を示す．なお，入力ファイルは表 7.6 と共通である．従って，作成方法もまた同様である．

表 B.1　実行する UNIX コマンドとそのコマンドライン及び入力ファイルと行数

コマンド	コマンドライン	入力ファイル	行数
cut	`cut -d ' ' -f 2`	1.txt	85159800
fmt	`fmt -w 1`	10.txt	13561000
sed	`sed 's/[^[:alnum:]]/ /g'`	8.txt	5114500
grep	`grep '[opq]'`	11.txt	8233600
tr	`tr a-z A-Z`	2.txt	25311000

1 つのキャッシュで扱う入力の行数は以下の通りである．

1. $2^{10} = 1024$
2. $2^{12} = 4096$
3. $2^{14} = 16384$
4. $2^{16} = 65536$
5. $2^{18} = 262144$
6. $2^{20} = 1048576$
7. $2^{22} = 4194304$

表 B.2　実験環境 3

コンポーネント	スペック
AMI イメージ	Ubuntu 22.04
インスタンスタイプ	t2.large (vCPU: 2, メモリ: 8(GiB))
EBS	20GB (SSD)

8. $2^{24} = 16777216$
9. $2^{26} = 67108864$
10. $2^{28} = 268435456$
11. $2^{30} = 1073741824$

B.3　実験環境

実験環境として，AWS が提供する Amazon EC2 のインスタンスを使用した．インスタンスの具体的なスペックを表 B.2 に示す．

B.4　実験結果

B.4.1　各コマンドの実行時間

一度にキャッシュする行数を変化させて実行した試作型 incrementalize の各コマンドごとの実行時間について，cut コマンドについての実行時間を表 B.3 に，fmt コマンドについての実行時間を表 B.4 に，sed コマンドについての実行時間を表 B.5 に，grep コマンドについての実行時間を表 B.6 に，tr コマンドについての実行時間を表 B.7 にそれぞれ示す．同様に改良型 incrementalize の各コマンドごとの実行時間について，cut コマンドについての実行時間を表 B.8 に，fmt コマンドについての実行時間を表 B.9 に，sed コマンドについての実行時間を表 B.10 に，grep コマンドについての実行時間を表 B.11 に，tr コマンドについての実行時間を表 B.12 にそれぞれ示す．

B.4.2　巨大な文字列を扱う際のエラー

コマンドによっては，実験内容で述べたキャッシュ行数のうち，一部の大きなキャッシュ行数について結果が示されていないものがある．これは，エラーが発生したことで正常に実行することができなかったことを示している．エラー内容は，十分なメモリを確保できなかった旨を示していた．同様のエラーは次のようなパイプラインによって再現できる．

表 B.3　cut コマンドについての実験結果 1

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	392.248	420.347	385.677	11.6107
4096	270.269	273.425	269.070	1.22249
16384	395.511	480.954	239.568	68.6731
65536	404.755	455.128	362.783	35.5512
262144	417.101	457.860	369.630	33.317
1048576	418.769	470.514	376.079	35.0205
4194304	419.722	464.949	371.604	35.591
16777216	422.528	473.458	370.572	33.671
67108864	-	-	-	-
268435456	-	-	-	-
1073741824	-	-	-	-

表 B.4　fmt コマンドについての実験結果 1

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	104.875	127.279	99.988	9.39427
4096	81.6691	83.126	79.040	1.18427
16384	75.029	76.573	73.849	1.07086
65536	82.1854	83.066	80.893	0.614155
262144	88.4878	88.864	88.010	0.317629
1048576	89.2686	90.446	88.526	0.576619
4194304	94.9852	121.608	87.369	11.4551
16777216	25.3565	37.803	19.229	7.61603
67108864	25.7865	39.406	19.346	8.21554
268435456	25.1795	35.765	19.278	7.24728
1073741824	25.8281	37.813	19.207	8.26489

```
$ od /dev/urandom | { read -N $(( 2**30 )); : "$REPLY"; }
bash: xrealloc: cannot allocate 18446744071562068096 bytes
```

このパイプラインは，まず od /dev/urandom によって巨大な文字列を作成する．/dev/urandom はランダムなバイトストリームを返すデバイスファイルである．これを，入力データの 8 進数表現をファイルのオフセット付きで出力する od コマンドに入力として

表 B.5　sed コマンドについての実験結果 1

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	63.373	65.995	60.237	1.64015
4096	53.2132	57.434	51.619	1.8248
16384	58.0468	61.116	53.763	2.47039
65536	61.2494	67.461	56.833	3.22255
262144	64.7053	67.978	59.530	2.55985
1048576	64.9743	67.831	60.500	2.36271
4194304	66.6727	69.117	64.446	1.35827
16777216	-	-	-	-
67108864	-	-	-	-
268435456	-	-	-	-
1073741824	-	-	-	-

表 B.6　grep コマンドについての実験結果 1

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	81.9555	99.237	77.830	6.82824
4096	69.9012	70.846	69.237	0.436189
16384	71.2048	75.892	69.041	1.83887
65536	80.2595	81.378	79.361	0.628133
262144	81.2897	82.602	80.272	0.745446
1048576	84.2832	85.317	83.306	0.655502
4194304	86.9369	88.120	85.967	0.637125
16777216	-	-	-	-
67108864	-	-	-	-
268435456	-	-	-	-
1073741824	-	-	-	-

渡すことで，できるだけ効率よく ASCII に含まれる印字可能な文字のみで構成される任意の巨大な文字列を構築する．なお，そのような文字列を構築できるのなら，ほかの方法を用いても構わない．そのようにして構築された文字列を Bash の組み込みコマンドである read で読み込んでシェル変数に格納する．変数名を指定しない場合，デフォルトで `REPLY` という変数名が使用される．`-N` オプションは，オプション引数によって指定された数だけ文字を読むかあるいは入力を読めなくなるまで入力を読み込む．ここでは，ある程度大きな文字数と

表 B.7　tr コマンドについての実験結果 1

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	141.055	142.938	139.993	0.88036
4096	107.042	107.403	106.441	0.291714
16384	99.1935	101.589	98.087	1.08692
65536	100.878	103.467	99.731	1.08896
262144	148.393	215.378	108.070	43.117
1048576	116.278	117.414	115.269	0.690078
4194304	172.314	220.159	121.600	37.1411
16777216	175.929	220.292	124.100	36.9352
67108864	28.4415	39.331	21.095	8.15731
268435456	28.9535	43.541	21.212	9.29702
1073741824	28.813	40.817	21.207	8.42389

表 B.8　cut コマンドについての実験結果 2

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	85.409	85.750	85.024	0.266687
4096	22.1155	22.415	21.956	0.151608
16384	13.9702	16.977	12.541	1.86549
65536	13.5394	16.184s	12.283	1.37002
262144	12.9199	15.590	12.256	1.07286
1048576	13.6834	14.389	13.471	0.259717
4194304	18.3422	19.667	17.034	1.07846
16777216	19.4116	22.438	18.498	1.09293
67108864	18.937	20.691	17.236	0.842933
268435456	19.6866	22.354	18.858	0.994545
1073741824	19.6854	22.133	18.490	1.35723

して 1 ギガ ($2^{30} = 1073741824$) を指定している．これにより，`REPLY` というシェル変数に 1 ギガ分の文字列が束縛されることになる．その後，シェル変数 `REPLY` に束縛されている文字列を変数展開で，何もしない Bash の組み込みコマンドである : の引数として指定して実行する．すると，上記のような，必要な量のメモリを確保できないというエラーが発生する．これは，巨大な文字列を束縛している変数を変数展開しようとすることで発生するエラーであると考えられる．このことは，シェルの実装にもよるが，少なくとも Bash をインタープ

表 B.9　fmt コマンドについての実験結果 2

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	14.558	14.639	14.490	0.0485112
4096	10.4058	12.849	9.416	1.52585
16384	10.1872	13.234	9.674	1.07995
65536	12.0985	12.363	11.931	0.120796
262144	20.0602	27.729	17.485	3.10958
1048576	21.8957	28.421	19.546	2.55834
4194304	23.8319	30.552	22.140	2.40142
16777216	22.2522	29.782	19.476	2.94058
67108864	20.1265	29.350	17.433	3.471
268435456	19.3987	28.790	17.345	3.38147
1073741824	19.2394	28.352	17.350	3.25322

表 B.10　sed コマンドについての実験結果 2

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	9.7048	9.815	9.630	0.061097
4096	10.6949	11.255	10.509	0.21333
16384	15.6483	26.855	13.997	3.9413
65536	27.0624	89.700	17.108	22.0482
262144	27.13	94.111	17.579	23.5609
1048576	24.2361	69.440	17.659	15.9472
4194304	26.2507	99.433	17.601	25.7204
16777216	19.8091	22.124	18.559	1.25246
67108864	26.83	100.564	17.737	25.9347
268435456	26.4675	100.161	17.646	25.9079
1073741824	26.6866	99.609	17.718	25.6361

リタとして使用する場合，シェルスクリプトにおいてシェル変数により巨大な文字列を扱うことは難しいことを示している．

B.4.3　試作型 incrementalize

試作型 incrementalize を使用した場合のいずれのコマンドの実験結果についても，キャッシュ行数を X 軸，実行時間を Y 軸として $2^{12} = 4096$ か $2^{14} = 16384$ を軸とする二次曲線の

表 B.11　grep コマンドについての実験結果 2

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	9.7239	9.774	9.660	0.0374595
4096	9.4161	9.464	9.363	0.0369066
16384	9.4366	9.561	9.369	0.0587182
65536	9.677	9.766	9.605	0.0525907
262144	10.2182	10.514	9.998	0.197332
1048576	12.5435	13.424	12.096	0.473821
4194304	18.5162	22.250	17.963	1.31374
16777216	18.7898	19.443	17.940	0.646605
67108864	18.7832	21.904	17.955	1.19751
268435456	19.3144	22.652	17.972	1.57909
1073741824	20.7305	24.258	17.903	1.96689

表 B.12　tr コマンドについての実験結果 2

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	26.3047	26.494	26.036	0.157079
4096	9.3834	9.456	9.328	0.0417378
16384	9.2591	9.312	9.144	0.0630845
65536	9.5069	10.013	9.406	0.181033
262144	10.379	10.479	10.175	0.0953345
1048576	14.6545	17.641	13.805	1.33401
4194304	21.0367	23.711	17.653	1.72615
16777216	18.7039	22.543	17.524	1.93907
67108864	91.0819	110.462	19.937	35.8001
268435456	111.596	140.600	106.443	10.2886
1073741824	169.827	211.910	144.283	19.2522

ようになっている．このことは，単純にキャッシュ行数を増やしていっても実行時間が短くなるわけではないことを示している．ただし，一部のコマンドについては，キャッシュ行数を大きくすることで実行時間が短くなっているところがある．例えば，fmt コマンドについては，キャッシュ行数が $2^{24} = 16777216$ のところから実行時間が短くなっている．これは，キャッシュ行数が fmt コマンドの入力行数である 13561000 を超えるため，一度のキャッシングで全入力行を読み込むためであると考えられる．実際，それ以上の行数の場合において

も，$2^{24} = 16777216$ の場合の実行時間とほとんど違いがない．これにより，キャッシングの処理が一度しか実行されず，それに関連したオーバーヘッドも少なくなり，結果として実行時間が短くなったと考えられる．これは，tr コマンドにおいても同様の傾向が見られる．

B.4.4 改良型 incrementalize

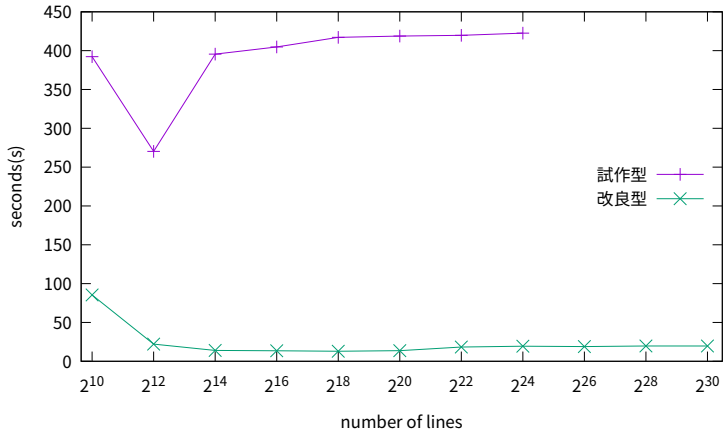
改良型 incrementalize を使用した場合のいずれのコマンドの実験結果についても，試作型 incrementalize の場合の実験結果 (B.4.3) と同様に，一度にキャッシュする入力行数とコマンドの平均実行時間の関係は二次関数に近いものになっており，一定の行数に向かって平均実行時間が減少し，その後は増加していく傾向にある．改良型 incrementalize の場合，入力行を一行ずつ読み込んでいく実装になっている都合上，一度にキャッシュする入力行数が変化しても，そこに係る標準入力からの読み出しとハッシュ値の計算の負荷は変化しない．変化するのは，読み出した入力行を書き出すファイルのサイズとキャッシュされている出力のサイズである．ただし，書き出すファイルはインメモリファイルシステム上に配置されていたり，仮にそうでなくとも，ファイルのディスクキャッシュが作成されたりする．このとき，書き込みの総量がメモリサイズに収まる限りはメモリ上での書き込みに終始できるため，処理にそれほど大きな影響はない．それよりも，明確にディスクアクセスが発生するキャッシュの検索と読み出しのほうが，処理に影響が大きいと思われる．本実験の設定上，事前に実行するコマンドと一度にキャッシュする行数の組み合わせについてのキャッシュを作成しているために，測定しているコマンド実行は，全てキャッシュがヒットすることになる．すると，指定されている行数を標準入力から読むごとにディスクアクセスが発生することになる．このときのオーバーヘッドが，一度にキャッシュする入力行数とコマンドの平均実行時間との関係について前述のように二次関数上になっており，かつ，それがもっとも支配的であるために，今回の実験結果のようになるのではないかという推測ができる．

B.4.5 試作型 incrementalize と改良型 incrementalize の比較

試作型 incrementalize と改良型 incrementalize によるそれぞれの実験結果のうち，cut コマンドについての平均実行時間の比較を図 B.1 に，fmt コマンドについての平均実行時間の比較を図 B.2 に，sed コマンドについての平均実行時間の比較を図 B.3 に，grep コマンドについての平均実行時間の比較を図 B.4 に，tr コマンドについての平均実行時間の比較を図 B.5 に，それぞれ示す．

いずれのコマンドについても，試作型 incrementalize と比べて，改良型 incrementalize は平均実行時間が大きく改善されていることがわかる．理由としては，最大のボトルネックとなっていた，入力に対応するハッシュ値の計算時間を短縮できたであろうことが考えられる．単純にハッシュ値の計算といっても，実際には，ハッシュ値を計算するコマンドの実行である．以前であれば，xxh64sum というコマンドに標準入力経由で入力値を渡してハッシュ値

図 B.1　cut コマンドについての平均実行時間の比較



を算出していた．だが，この方法では毎回コマンドを実行することになる．これは，毎回 fork が発生するということであり，その分のオーバーヘッドが発生するということである．また，一度標準入力から読んだ入力を再度標準入力経由で読むことになるため，そのオーバーヘッドもそれなりの影響があったであろうことが予想される．しかし，新しい実装では，これを専用の補助コマンド内で，標準入力からの読み出しとハッシュ値の計算を一括に実行することで，先のオーバーヘッドを解消することができた．そのほか，以前の実装では，それらの処理と，キャッシュの読み出し処理とキャッシング処理を 1 つのコマンド内で直列に実行していた．だが，実際にはこの 2 つの処理は並列に実行することが可能である．そこで，改良型 incrementalize では，その 2 つの処理をそれぞれ別のコマンド上で実行し，シェルのパイプラインを経由して通信している．そのため，2 つのコマンドは並列に動作することが可能になっており，入力における 1 つのハッシュ値の計算が完了したのち，キャッシュ関連の処理の完了を待つことなく，即座に次のハッシュ値の計算を開始できる．この 2 つが，試作型 incrementalize の場合と比べて，改良型 incrementalize では平均処理時間が大幅に改善された理由であると考えられる．

B.5　実験のまとめ

今回の実験で，基本的なアルゴリズムは変更せずに実装を変更することで，incrementalize の性能を大幅に改善できたことを確認した．また，試作型と改良型の 2 つの incrementalize についても，シェルスクリプトに対してキャッシュを利用した複数行単位の増分計算を実現する上で，一度にキャッシュする行数を増やしても，処理速度は単純に高速化されるわけではないことがわかった．

しかし，incrementalize による増分計算によって，本実験で扱った単体の UNIX コマンドの実行速度を高速化することは難しい．これは，行指向という，概念的には常にパイプラ

図 B.2　fmt コマンドについての平均実行時間の比較

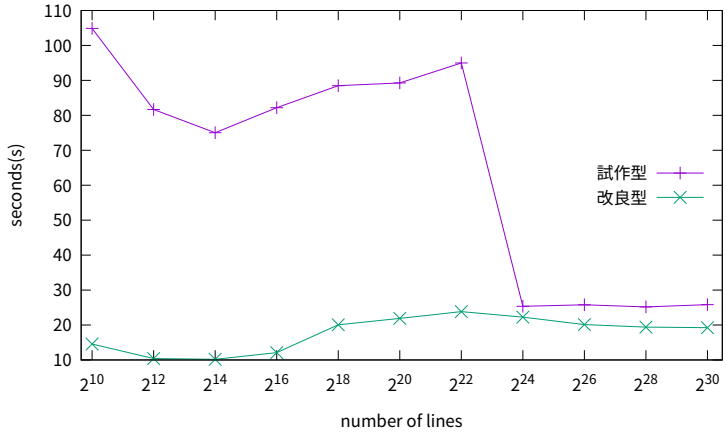


図 B.3　sed コマンドについての平均実行時間の比較

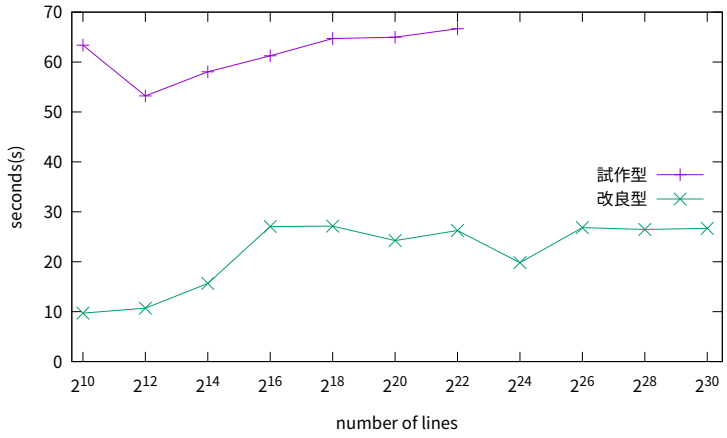


図 B.4　grep コマンドについての平均実行時間の比較

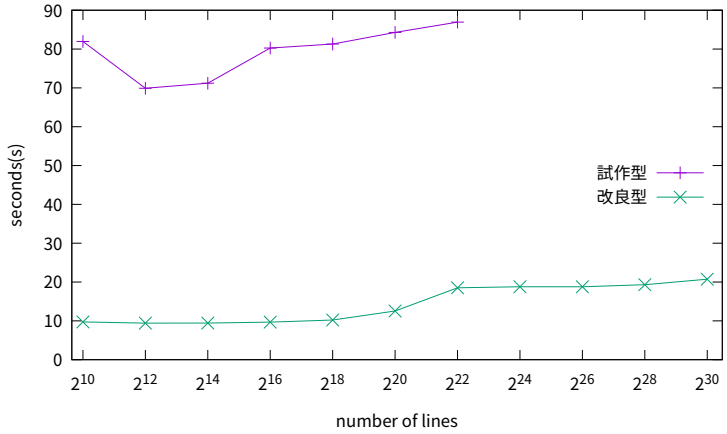
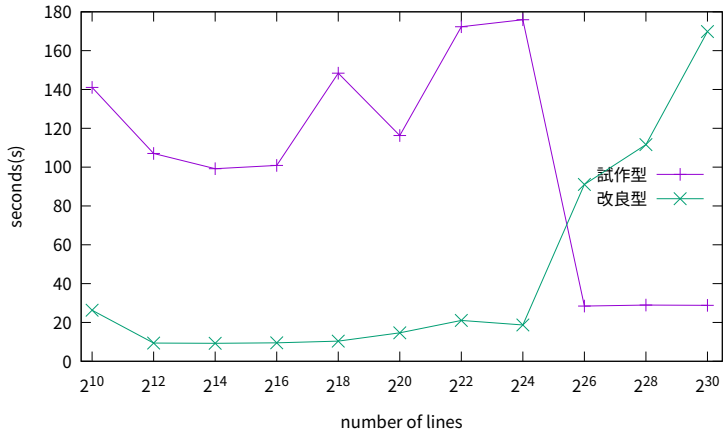


図 B.5　tr コマンドについての平均実行時間の比較



イン内をデータストリームが流れるような性質をもったコマンドに対して，キャッシュを扱う incrementalize はどうしてもその流れを堰き止める必要があることによる．つまり，そのオーバーヘッドが，例えキャッシュがヒットすることによってコマンドの実行自体をスキップすることができたとしても，コマンドの通常実行よりも大きくなってしまう可能性があるということである．実装を見直すことによって多少の改善はできるかもしれないが，incrementalize で実装したアルゴリズムを使用している限り，根本的な問題であるストリームの堰き止めは解決できない．ただ，そのようなコマンドは，行指向という都合上，通常実行の時点で十分に高速であることも多い．そのため，今回の実験で扱ったような単一の UNIX コマンド実行については，適用可能であるものの，主な適用対象ではないとすることも考えられる．

一方で，複数の UNIX コマンドからなるパイプラインについてはこの限りではない．なぜなら，複数の UNIX コマンドを同時に実行する都合上，単体の UNIX コマンドよりも実行速度が低速になる可能性が高いからである．ただし，今回の実験では，試作型 incrementalize と改良型 incrementalize の性能を比較することが目的であったため，実験の対象としたコマンドも両者が増分計算を適用可能な単体の UNIX コマンドのみを扱った．

付録 C

incrementalize の実験結果

以下のグラフ群に，7.3 についての実験結果のうち，各コマンドとキャッシュ行数の組み合わせについて incrementalize で増分計算を適用した際の実行時間を示す．これらのグラフ群は，本来は 7.3 で扱うべきであるが，量が多いために付録として記載する．

表 C.1　test-unix50-1-cut-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0163	7.026	7.012	0.00507828

表 C.2　test-unix50-1-cut-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	52.1542	52.457	51.340	0.333244
4096	13.6027	14.563	13.305	0.35398
16384	7.5655	8.176	7.352	0.259521
65536	7.9823	8.294	7.683	0.187972
262144	7.9939	8.242	7.733	0.141053
1048576	8.3404	8.461	8.129	0.107146
4194304	10.2638	10.405	10.227	0.0525416
16777216	11.0509	12.000	10.220	0.845827
67108864	10.5204	11.754	10.209	0.50259
268435456	10.8953	11.951	10.195	0.771194
1073741824	10.4122	11.798	10.152	0.491517

表 C.3　test-unix50-1-cut-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	52.0199	52.448	51.713	0.21049
4096	13.2285	13.360	13.144	0.069975
16384	7.16	7.211	7.141	0.0219038
65536	7.1759	7.187	7.167	0.00726407
262144	7.4334	7.442	7.428	0.00497103
1048576	8.565	9.291	8.482	0.255102
4194304	12.5627	13.178	12.270	0.285821
16777216	12.4374	12.799	11.847	0.374496
67108864	12.0231	13.352	11.849	0.469335
268435456	12.0567	13.864	11.848	0.63504
1073741824	12.2897	13.913	11.846	0.662064

表 C.4　test-unix50-10-fmt-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	8.2177	8.459	8.101	0.112092

表 C.5　test-unix50-10-fmt-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	9.9649	10.272	9.583	0.205668
4096	8.9238	9.309	8.530	0.29279
16384	10.9171	11.289	10.417	0.252052
65536	9.5922	9.766	9.357	0.116582
262144	16.1875	16.213	16.144	0.0233963
1048576	16.2193	16.331	16.145	0.0628137
4194304	16.2901	16.477	16.130	0.122956
16777216	16.414	16.761	16.241	0.178491
67108864	16.3609	16.998	16.113	0.284965
268435456	16.476	17.157	16.192	0.281239
1073741824	16.3139	16.663	16.197	0.137432

表 C.6　test-unix50-10-fmt-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	8.7266	8.767	8.677	0.0291936
4096	7.2893	7.304	7.283	0.00700872
16384	7.8273	8.500	7.734	0.238488
65536	9.8601	9.892	9.840	0.0143562
262144	14.9924	16.241	14.846	0.438745
1048576	15.097	17.286	14.839	0.769177
4194304	15.4051	20.389	14.842	1.75117
16777216	15.7778	24.119	14.841	2.93081
67108864	15.7166	23.478	14.842	2.72709
268435456	15.7366	23.704	14.842	2.79947
1073741824	15.7163	23.501	14.839	2.73528

表 C.7　test-unix50-8-sed-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	50.5962	51.069	50.415	0.193892

表 C.8　test-unix50-8-sed-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	26.5856	28.110	25.281	0.995794
4096	40.6061	41.080	40.063	0.313506
16384	53.7739	54.451	52.923	0.390298
65536	58.2364	58.295	58.206	0.0310383
262144	58.9358	59.280	58.809	0.148609
1048576	58.9731	59.203	58.768	0.148736
4194304	59.0609	59.308	58.851	0.179317
16777216	59.1573	59.441	58.848	0.201714
67108864	59.4271	61.374	58.855	0.706916
268435456	59.2037	59.824	58.635	0.299126
1073741824	59.3406	59.705	59.013	0.211683

表 C.9　test-unix50-8-sed-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.4932	7.754	7.454	0.0918922
4096	8.3776	8.399	8.352	0.0198001
16384	18.7201	54.982	12.066	13.5645
65536	68.704	89.761	58.965	7.84905
262144	67.8902	95.636	50.169	11.0915
1048576	67.93	93.944	50.988	10.4441
4194304	68.1887	89.001	59.505	7.69437
16777216	69.5665	83.477	67.782	4.89421
67108864	69.5611	83.798	67.843	5.00425
268435456	69.3254	83.731	66.578	5.09937
1073741824	69.1335	83.586	66.239	5.13003

表 C.10　test-unix50-11-grep-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	0.0176	0.023	0.016	0.00206559

表 C.11　test-unix50-11-grep-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.4526	28.694	28.011	0.190967
4096	28.1749	28.642	27.349	0.413917
16384	27.6196	27.853	27.293	0.232518
65536	28.3737	28.800	27.764	0.333205
262144	28.8303	29.229	28.462	0.189716
1048576	32.022	32.417	31.593	0.305664
4194304	29.8458	30.442	29.303	0.453725
16777216	29.5279	30.052	28.789	0.38154
67108864	29.5471	30.131	28.943	0.411973
268435456	29.9011	30.394	29.286	0.316626
1073741824	29.4724	30.305	28.934	0.419408

表 C.12　test-unix50-11-grep-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.9528	28.614	27.760	0.248604
4096	28.1939	28.659	27.266	0.454884
16384	28.2194	28.827	27.809	0.279827
65536	29.0313	29.646	28.713	0.338283
262144	30.4645	30.654	30.314	0.123124
1048576	41.1324	42.829	40.302	0.977277
4194304	69.2529	77.142	66.921	3.19215
16777216	68.8788	69.955	66.790	0.968747
67108864	69.3549	69.833	68.969	0.301932
268435456	68.2577	71.902	66.990	1.49053
1073741824	68.0284	69.535	66.815	1.09167

表 C.13　test-unix50-2-tr-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.2381	27.972	26.667	0.494817

表 C.14　test-unix50-2-tr-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.7446	29.219	28.368	0.271386
4096	28.1772	28.998	27.801	0.410524
16384	28.4206	29.180	28.034	0.312397
65536	28.8492	29.726	28.240	0.444294
262144	31.4877	32.015	30.988	0.33342
1048576	43.686	46.634	40.660	2.42016
4194304	28.1826	28.659	27.916	0.269965
16777216	28.1306	28.991	27.551	0.54874
67108864	28.2641	28.942	27.652	0.459949
268435456	28.2977	28.953	27.435	0.499874
1073741824	28.2771	29.147	27.659	0.53053

表 C.15　test-unix50-2-tr-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.0288	28.511	27.723	0.301759
4096	27.787	28.126	27.426	0.233207
16384	27.5415	28.102	27.086	0.308946
65536	28.9804	29.481	28.581	0.256362
262144	32.2803	32.775	31.906	0.3346
1048576	49.2836	51.714	48.715	0.923019
4194304	67.5112	74.571	65.595	2.68124
16777216	67.4391	74.533	65.733	2.67128
67108864	66.9893	68.355	65.555	1.06236
268435456	67.116	68.228	63.348	1.48448
1073741824	68.7563	78.068	64.456	4.29009

表 C.16　test-unix50-5-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0136	7.021	7.009	0.00377712

表 C.17　test-unix50-5-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	15.5175	15.741	15.285	0.124272
4096	7.1734	7.440	7.135	0.0938973
16384	7.1701	7.197	7.123	0.0239186
65536	7.3936	7.492	7.310	0.0610886
262144	7.5731	7.662	7.487	0.0647447
1048576	9.495	9.613	9.384	0.080774
4194304	9.5556	9.602	9.516	0.0325037
16777216	9.5611	9.708	9.506	0.0574523
67108864	9.5703	9.622	9.510	0.0457093
268435456	9.563	9.644	9.510	0.0443772
1073741824	9.5975	9.675	9.559	0.034805

表 C.18　test-unix50-5-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	15.4003	15.480	15.341	0.0399918
4096	7.1285	7.137	7.124	0.00485913
16384	7.0906	7.098	7.088	0.00295146
65536	7.0723	7.079	7.067	0.00374314
262144	7.1939	7.344	7.173	0.052834
1048576	7.6568	8.228	7.573	0.205608
4194304	8.0427	8.858	7.943	0.286559
16777216	8.1276	9.743	7.943	0.567604
67108864	8.1289	9.752	7.942	0.570324
268435456	8.1256	9.721	7.943	0.560589
1073741824	8.1215	9.690	7.943	0.551136

表 C.19　test-unix50-6-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.014	7.023	7.009	0.00394405

表 C.20　test-unix50-6-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	44.4484	44.723	43.524	0.341739
4096	11.323	11.990	11.159	0.242614
16384	7.1405	7.317	7.109	0.0644364
65536	7.0989	7.167	7.086	0.0244061
262144	7.1437	7.271	7.076	0.0532751
1048576	8.4204	10.073	7.513	0.790588
4194304	9.1713	9.701	8.548	0.415941
16777216	10.5753	11.399	10.282	0.380116
67108864	10.8421	11.641	10.194	0.522105
268435456	10.4107	10.867	10.232	0.191747
1073741824	10.3521	10.610	10.156	0.162306

表 C.21　test-unix50-6-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	44.4101	44.669	44.114	0.163914
4096	11.1587	11.221	11.072	0.0498176
16384	7.1133	7.121	7.109	0.00405654
65536	7.0849	7.092	7.081	0.00341402
262144	7.0754	7.082	7.071	0.00383551
1048576	7.3415	9.204	7.105	0.655232
4194304	7.1969	7.375	7.163	0.0677503
16777216	7.7111	8.980	7.561	0.445869
67108864	7.9223	10.857	7.568	1.03369
268435456	7.8543	10.388	7.570	0.890257
1073741824	7.5677	7.574	7.562	0.00374314

表 C.22　test-unix50-7-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0136	7.020	7.010	0.00320416

表 C.23　test-unix50-7-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0998	7.144	7.074	0.0177501
4096	7.0836	7.113	7.072	0.0119369
16384	7.1195	7.143	7.104	0.016071
65536	7.5057	7.575	7.362	0.0958935
262144	8.8641	9.227	8.300	0.419633
1048576	12.5339	14.261	10.113	1.95881
4194304	13.1827	16.678	12.541	1.23581
16777216	12.8784	13.081	12.590	0.171312
67108864	12.754	13.059	12.533	0.205921
268435456	29.2852	34.029	12.678	8.68725
1073741824	33.6684	34.479	32.960	0.472258

表 C.24　test-unix50-7-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.8423	28.186	27.442	0.234773
4096	27.7357	28.089	27.330	0.219528
16384	27.7374	28.139	27.392	0.265004
65536	28.6058	28.966	28.072	0.260608
262144	31.2817	31.647	31.021	0.214595
1048576	38.5171	39.328	37.869	0.42813
4194304	50.3025	52.235	49.268	1.04115
16777216	53.1509	64.869	49.908	4.18982
67108864	51.8219	52.774	49.969	1.01742
268435456	52.1105	52.268	51.408	0.264212
1073741824	52.1923	52.855	51.454	0.427993

表 C.25　test-unix50-8-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.1473	27.952	26.786	0.354632

表 C.26　test-unix50-8-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.3115	33.474	27.383	1.82605
4096	27.8452	28.229	27.469	0.269932
16384	27.8438	28.237	27.240	0.331441
65536	27.9937	28.393	27.518	0.336713
262144	28.3549	28.847	27.867	0.319105
1048576	30.1053	30.937	29.223	0.636789
4194304	29.9587	30.511	29.537	0.395374
16777216	29.6431	30.145	29.127	0.384637
67108864	29.7755	30.125	29.121	0.295116
268435456	29.8864	30.463	29.111	0.409577
1073741824	29.601	30.287	29.098	0.39403

表 C.27　test-unix50-8-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.7824	28.101	27.314	0.209084
4096	27.8117	28.172	27.398	0.247801
16384	27.6659	28.146	27.335	0.291295
65536	27.9093	28.561	27.341	0.470603
262144	28.7674	29.133	28.181	0.307783
1048576	30.8948	31.927	30.527	0.411591
4194304	32.966	33.208	32.688	0.171201
16777216	33.1518	33.448	32.908	0.1504
67108864	33.0863	33.235	32.908	0.121447
268435456	31.9907	32.408	31.828	0.160689
1073741824	32.4002	32.887	31.973	0.368004

表 C.28　test-unix50-9-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.5639	31.648	26.713	1.46639

表 C.29　test-unix50-9-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.6691	28.241	27.008	0.367627
4096	27.5738	28.021	26.964	0.318494
16384	27.7604	28.475	27.312	0.378446
65536	28.0678	29.212	27.101	0.588441
262144	28.4177	29.058	27.678	0.445674
1048576	30.7489	32.392	29.269	1.11294
4194304	31.3147	31.730	30.735	0.375962
16777216	30.883	31.309	30.248	0.338912
67108864	31.2626	31.681	30.685	0.339983
268435456	31.1124	31.587	30.644	0.331143
1073741824	31.1126	31.887	30.477	0.409487

表 C.30　test-unix50-9-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.6957	27.852	27.258	0.20355
4096	27.7453	28.442	27.322	0.387983
16384	27.6263	28.510	27.170	0.407465
65536	27.8447	28.580	27.132	0.466695
262144	28.5412	32.802	27.472	1.5324
1048576	28.2363	28.651	27.752	0.348033
4194304	28.5258	29.113	27.959	0.363612
16777216	28.3245	28.872	27.833	0.355656
67108864	28.512	31.331	27.506	1.03751
268435456	28.2805	28.692	27.719	0.336958
1073741824	28.2803	28.799	27.773	0.357494

表 C.31　test-unix50-10-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0144	7.025	7.010	0.0044771

表 C.32　test-unix50-10-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0788	7.137	7.068	0.0206656
4096	7.0782	7.104	7.069	0.0104009
16384	7.0849	7.101	7.079	0.00688719
65536	7.6554	9.017	7.198	0.590323
262144	8.0524	8.139	7.733	0.165013
1048576	10.0813	10.434	8.875	0.634858
4194304	10.3693	10.403	10.282	0.0398694
16777216	10.3813	10.560	10.260	0.0792016
67108864	10.3846	10.421	10.367	0.0143465
268435456	10.3801	10.412	10.276	0.0388028
1073741824	10.3751	10.403	10.264	0.0405146

表 C.33　test-unix50-10-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0715	7.079	7.059	0.0051908
4096	7.0658	7.074	7.059	0.00505085
16384	7.0741	7.087	7.066	0.00815748
65536	7.0867	7.118	7.052	0.0296388
262144	7.1151	7.413	7.076	0.104792
1048576	7.208	7.495	7.171	0.100873
4194304	7.6146	10.442	7.295	0.993453
16777216	7.6156	10.455	7.296	0.997668
67108864	7.6161	10.457	7.297	0.998199
268435456	7.6163	10.460	7.293	0.999185
1073741824	7.2992	7.308	7.295	0.00376534

表 C.34　test-unix50-11-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0133	7.022	7.009	0.0034657

表 C.35　test-unix50-11-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0705	7.081	7.045	0.00948976
4096	7.0819	7.134	7.066	0.0194391
16384	7.0846	7.103	7.079	0.00787683
65536	7.4466	8.568	7.195	0.416082
262144	8.0139	8.114	7.708	0.158669
1048576	10.1509	10.424	8.805	0.474996
4194304	10.2493	10.277	10.232	0.0164725
16777216	10.2446	10.267	10.233	0.0100022
67108864	10.2404	10.269	10.210	0.0180813
268435456	10.2488	10.274	10.220	0.0170411
1073741824	10.2474	10.267	10.221	0.0147964

表 C.36　test-unix50-11-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0707	7.088	7.063	0.00784644
4096	7.0674	7.076	7.062	0.00518973
16384	7.0723	7.083	7.063	0.00791693
65536	7.0697	7.112	7.056	0.0224601
262144	7.1006	7.109	7.095	0.00445222
1048576	7.244	7.522	7.210	0.0977343
4194304	7.6665	10.334	7.368	0.937266
16777216	7.6661	10.319	7.366	0.93214
67108864	7.3716	7.380	7.367	0.00422164
268435456	7.3714	7.379	7.367	0.00380643
1073741824	7.3715	7.379	7.367	0.00359784

表 C.37　test-unix50-12-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.1005	7.249	7.018	0.06416

表 C.38　test-unix50-12-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0843	7.141	7.070	0.0210452
4096	7.1015	7.153	7.073	0.0258811
16384	7.1048	7.121	7.094	0.00768548
65536	7.5115	7.646	7.390	0.120604
262144	9.0806	9.544	8.624	0.465461
1048576	13.2742	14.736	11.121	1.8243
4194304	14.5522	14.802	14.462	0.103005
16777216	14.5269	14.600	14.472	0.0379223
67108864	14.5309	14.638	14.437	0.0598451
268435456	14.5221	14.604	14.467	0.0449727
1073741824	14.4943	14.560	14.429	0.0421216

表 C.39　test-unix50-12-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.0706	7.081	7.066	0.00424788
4096	7.0693	7.076	7.067	0.00258414
16384	7.0849	7.107	7.078	0.00825227
65536	7.1148	7.232	7.098	0.0413462
262144	7.3955	8.000	7.324	0.212423
1048576	7.8784	8.503	7.803	0.219495
4194304	8.4484	8.457	8.445	0.00392145
16777216	9.0997	14.922	8.445	2.04576
67108864	8.4484	8.456	8.442	0.00392145
268435456	9.0914	14.869	8.445	2.03005
1073741824	9.1027	14.990	8.442	2.06859

表 C.40　test-unix50-13-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0133	7.025	7.009	0.0046916

表 C.41　test-unix50-13-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	62.9242	63.297	62.097	0.380814
4096	16.0797	17.016	15.802	0.347933
16384	8.2137	8.657	7.961	0.206723
65536	7.83	7.922	7.721	0.0710493
262144	7.8068	8.398	7.674	0.213494
1048576	7.7601	7.828	7.703	0.0389999
4194304	8.2944	8.869	8.061	0.263052
16777216	9.2493	9.770	8.767	0.240347
67108864	10.8793	11.718	10.515	0.464157
268435456	10.9557	12.030	10.422	0.587331
1073741824	10.7739	11.611	10.452	0.450311

表 C.42　test-unix50-13-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	64.8411	75.787	61.992	4.13671
4096	27.8078	28.389	27.167	0.451011
16384	27.5882	28.285	27.292	0.351405
65536	27.4724	29.154	27.061	0.653902
262144	27.7702	28.279	27.380	0.354916
1048576	28.0605	28.515	27.574	0.26812
4194304	29.0753	29.460	28.714	0.303067
16777216	31.3587	31.700	31.022	0.196736
67108864	39.9021	41.857	39.015	0.872015
268435456	39.4098	39.802	38.835	0.371708
1073741824	39.3982	40.262	37.461	0.792414

表 C.43　test-unix50-14-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.3677	27.975	26.920	0.316502

表 C.44　test-unix50-14-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	52.0093	54.720	51.196	1.04979
4096	29.0288	35.290	27.855	2.25639
16384	27.7365	28.736	27.355	0.374958
65536	27.4809	28.182	27.031	0.281484
262144	27.7518	28.107	27.412	0.250926
1048576	28.2091	28.813	27.377	0.474028
4194304	30.1963	30.549	29.902	0.244306
16777216	38.9419	40.148	34.308	1.76722
67108864	45.4932	46.477	44.251	0.810642
268435456	45.3019	46.674	44.390	0.695649
1073741824	45.238	46.054	43.656	0.816714

表 C.45　test-unix50-14-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	51.5132	52.103	50.852	0.408793
4096	28.1973	29.060	27.687	0.394446
16384	27.3351	27.673	27.048	0.231637
65536	27.3267	27.567	27.081	0.174316
262144	27.9385	28.268	27.274	0.322969
1048576	28.0624	28.688	27.567	0.33391
4194304	28.9256	29.532	28.446	0.353129
16777216	33.8899	34.346	32.846	0.560002
67108864	48.7526	70.114	45.287	7.57051
268435456	49.2179	68.461	45.573	6.84412
1073741824	49.8219	67.874	47.033	6.35732

表 C.46　test-unix50-15-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0159	7.023	7.012	0.00284605

表 C.47　test-unix50-15-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	17.6443	17.890	17.458	0.140741
4096	7.1981	7.494	7.156	0.104115
16384	7.1367	7.184	7.116	0.026077
65536	7.2363	7.278	7.156	0.0365089
262144	7.5853	8.062	7.168	0.257915
1048576	9.0672	9.200	8.826	0.12185
4194304	9.9456	9.992	9.892	0.0407109
16777216	9.9606	10.216	9.855	0.0980943
67108864	9.9263	10.040	9.869	0.0548636
268435456	9.9387	10.247	9.847	0.116095
1073741824	9.9219	10.007	9.875	0.0520821

表 C.48　test-unix50-15-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	17.5541	17.641	17.439	0.0789028
4096	7.1488	7.156	7.146	0.00297396
16384	7.069	7.078	7.062	0.00561743
65536	7.1053	7.112	7.100	0.00383116
262144	7.1892	7.209	7.182	0.00857386
1048576	7.5793	7.635	7.565	0.0201607
4194304	8.5923	9.972	8.432	0.484799
16777216	8.6256	10.313	8.433	0.592905
67108864	8.6354	10.403	8.430	0.621086
268435456	8.6279	10.324	8.430	0.59601
1073741824	8.6299	10.341	8.430	0.60124

表 C.49　test-unix50-18-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0305	7.042	7.026	0.00455217

表 C.50　test-unix50-18-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.2023	7.234	7.156	0.0195167
4096	7.175	7.227	7.156	0.0199555
16384	7.1297	7.146	7.124	0.00616532
65536	7.8632	7.882	7.836	0.0137097
262144	8.3047	10.818	7.997	0.883314
1048576	8.0094	8.067	7.975	0.0330461
4194304	7.9919	8.092	7.973	0.0367679
16777216	8.006	8.177	7.969	0.0636606
67108864	7.9817	8.019	7.966	0.0188859
268435456	7.997	8.062	7.971	0.0266291
1073741824	8.0074	8.129	7.975	0.044888

表 C.51　test-unix50-18-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.139	7.153	7.128	0.008
4096	7.0849	7.129	7.067	0.0186396
16384	7.0959	7.121	7.075	0.0148433
65536	7.5123	7.784	7.311	0.17257
262144	7.5635	7.992	7.313	0.19236
1048576	7.1678	8.039	7.043	0.310964
4194304	7.0944	7.285	7.044	0.0936378
16777216	7.1489	7.969	7.048	0.28827
67108864	7.0504	7.056	7.045	0.00350238
268435456	7.051	7.064	7.045	0.00551765
1073741824	7.1477	8.013	7.048	0.304053

表 C.52　test-unix50-19-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0435	7.058	7.040	0.00548229

表 C.53　test-unix50-19-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	8.2824	8.759	7.592	0.34001
4096	7.2161	7.265	7.196	0.0204094
16384	7.1581	7.181	7.149	0.0106818
65536	8.0421	8.060	8.025	0.00846496
262144	8.1794	8.188	8.174	0.00492612
1048576	8.1459	8.157	8.138	0.00634998
4194304	8.1369	8.154	8.131	0.00788036
16777216	8.1357	8.145	8.130	0.00537587
67108864	8.1336	8.152	8.126	0.00857904
268435456	8.1309	8.139	8.126	0.00422821
1073741824	8.1338	8.143	8.126	0.00509466

表 C.54　test-unix50-19-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.1474	7.161	7.130	0.0106375
4096	7.097	7.138	7.083	0.0177263
16384	7.1086	7.140	7.082	0.0197101
65536	7.6631	7.903	7.456	0.151834
262144	7.6585	7.898	7.473	0.1792
1048576	7.8102	8.143	7.695	0.154582
4194304	7.045	7.053	7.039	0.00397213
16777216	7.1542	8.134	7.039	0.344288
67108864	7.154	8.136	7.039	0.345075
268435456	7.3734	8.140	7.044	0.525798
1073741824	8.1392	8.158	8.134	0.0070206

表 C.55　test-unix50-20-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0319	7.042	7.028	0.00406749

表 C.56　test-unix50-20-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	7.2669	7.292	7.226	0.0190756
4096	7.3737	7.411	7.356	0.0156918
16384	7.2347	7.251	7.225	0.00780385
65536	26.0721	28.234	8.301	6.24629
262144	29.1426	29.566	28.835	0.265015
1048576	29.2929	29.847	28.797	0.331225
4194304	29.4325	29.930	28.771	0.457381
16777216	29.5082	30.145	28.951	0.358902
67108864	29.3019	30.018	28.642	0.444814
268435456	29.2155	29.902	28.727	0.451574
1073741824	29.5044	29.955	28.890	0.366888

表 C.57　test-unix50-20-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	29.2393	30.033	28.826	0.393222
4096	28.9883	29.321	28.608	0.217122
16384	29.2734	29.526	28.847	0.285222
65536	29.091	29.470	28.490	0.312365
262144	29.2449	29.749	28.715	0.338947
1048576	29.2413	29.500	28.781	0.236853
4194304	29.1759	29.394	28.994	0.135852
16777216	28.9564	29.638	28.375	0.418286
67108864	29.1873	29.783	28.614	0.324372
268435456	29.1271	29.428	28.454	0.344977
1073741824	29.8848	35.568	29.032	1.99903

表 C.58　test-unix50-21-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.4944	28.130	26.997	0.358151

表 C.59　test-unix50-21-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	29.9007	39.258	28.440	3.30372
4096	33.6584	33.990	33.344	0.258702
16384	33.1892	33.826	32.311	0.448848
65536	52.4688	52.823	51.887	0.31743
262144	53.193	53.660	52.701	0.331694
1048576	53.3047	54.068	52.675	0.409629
4194304	53.6071	54.020	53.040	0.288587
16777216	53.6634	54.131	53.163	0.37786
67108864	53.5099	54.157	52.802	0.382601
268435456	53.7646	54.416	53.136	0.426077
1073741824	53.7006	54.363	53.015	0.440051

表 C.60　test-unix50-21-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.9119	29.252	28.499	0.228679
4096	29.588	31.885	28.547	0.911002
16384	30.1731	32.912	29.152	1.3551
65536	38.7021	49.322	31.283	5.96529
262144	34.6569	48.448	27.266	7.81734
1048576	35.5153	48.125	27.433	6.21999
4194304	28.3151	32.607	27.507	1.52325
16777216	27.5884	28.188	27.074	0.336546
67108864	27.6817	28.155	27.395	0.286631
268435456	27.7142	28.173	27.468	0.259357
1073741824	27.9018	28.300	27.481	0.267

表 C.61　test-unix50-23-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	42.1416	44.730	40.188	1.66032

表 C.62　test-unix50-23-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	35.0818	35.477	34.474	0.321682
4096	9.0729	9.465	8.834	0.211071
16384	7.1087	7.229	7.092	0.0424082
65536	7.1623	7.526	7.064	0.165981
262144	7.2328	7.945	7.129	0.250499
1048576	13.1649	23.614	8.427	5.23743
4194304	25.5882	38.096	10.557	10.7399
16777216	45.4734	52.015	25.164	8.81927
67108864	51.586	53.588	48.885	1.58183
268435456	50.8531	54.447	49.055	1.70287
1073741824	50.3799	52.456	48.556	1.26636

表 C.63　test-unix50-23-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	35.1865	35.506	34.876	0.222849
4096	8.8663	8.956	8.768	0.0605604
16384	7.0781	7.092	7.070	0.00807534
65536	7.048	7.066	7.040	0.00840635
262144	7.0688	7.154	7.020	0.0596001
1048576	7.3681	7.699	7.051	0.331871
4194304	7.0261	7.033	7.022	0.00366515
16777216	7.6517	13.342	7.016	1.99937
67108864	24.3318	53.017	7.015	22.3824
268435456	16.2079	54.551	7.014	19.3889
1073741824	29.3096	52.886	7.010	23.5127

表 C.64　test-unix50-24-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0095	7.018	7.006	0.00337474

表 C.65　test-unix50-24-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.6364	28.854	28.160	0.204349
4096	7.3333	7.672	7.268	0.1211
16384	7.0782	7.186	7.062	0.0380199
65536	7.041	7.070	7.032	0.0113627
262144	7.0509	7.067	7.045	0.00650555
1048576	7.3794	8.381	7.132	0.377153
4194304	7.5633	7.591	7.384	0.0632421
16777216	9.3817	9.516	8.585	0.281309
67108864	9.4342	9.505	9.383	0.0438046
268435456	9.4258	9.502	9.378	0.0426792
1073741824	9.4314	9.500	9.374	0.048772

表 C.66　test-unix50-24-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.6417	28.852	28.346	0.137891
4096	7.3023	7.369	7.256	0.0342898
16384	7.0672	7.073	7.065	0.00252982
65536	7.0355	7.045	7.031	0.00408928
262144	7.0475	7.058	7.045	0.00389444
1048576	7.0776	7.097	7.055	0.0150643
4194304	7.0829	7.199	7.064	0.0412619
16777216	7.3206	7.910	7.252	0.207116
67108864	7.589	9.466	7.377	0.659522
268435456	7.5895	9.466	7.378	0.659344
1073741824	7.6001	9.576	7.376	0.694273

表 C.67　test-unix50-25-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0103	7.020	7.006	0.00377271

表 C.68　test-unix50-25-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	34.6192	35.192	34.260	0.306635
4096	8.8801	9.358	8.737	0.173972
16384	7.1044	7.226	7.083	0.0429837
65536	7.0543	7.078	7.036	0.0160281
262144	7.0605	7.120	7.030	0.0350721
1048576	7.376	7.638	7.191	0.155633
4194304	7.5612	7.570	7.555	0.00458984
16777216	9.1515	9.458	8.708	0.37021
67108864	9.416	9.451	9.361	0.0263607
268435456	9.4108	9.456	9.374	0.0331086
1073741824	9.409	9.452	9.379	0.0270473

表 C.69　test-unix50-25-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	34.3619	34.540	34.196	0.12325
4096	8.7722	8.863	8.685	0.0580571
16384	7.0871	7.098	7.084	0.00406749
65536	27.9744	30.801	27.312	1.01444
262144	27.5753	28.140	27.266	0.341968
1048576	27.8061	28.154	27.327	0.261993
4194304	28.1383	28.713	27.325	0.384932
16777216	30.0712	30.474	29.406	0.311825
67108864	31.2495	31.468	31.001	0.149747
268435456	31.2466	31.454	31.141	0.0922427
1073741824	31.1322	31.467	30.914	0.153283

表 C.70　test-unix50-26-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.3536	27.873	26.880	0.345738

表 C.71　test-unix50-26-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	30.116	45.646	27.709	5.47869
4096	28.2084	30.784	27.570	0.978915
16384	27.8012	28.567	27.080	0.515125
65536	27.9169	28.315	27.476	0.2276
262144	27.6829	27.939	27.419	0.18884
1048576	28.3968	29.456	27.519	0.491246
4194304	30.0185	30.709	29.063	0.50284
16777216	35.7855	36.128	35.310	0.234115
67108864	36.2195	36.637	35.932	0.233967
268435456	35.9967	39.703	35.020	1.3456
1073741824	36.1649	36.818	35.489	0.445308

表 C.72　test-unix50-26-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.433	30.230	27.765	0.95319
4096	27.7998	28.940	27.413	0.478823
16384	27.6294	28.221	27.093	0.325063
65536	27.7723	28.195	27.295	0.309505
262144	27.8184	28.106	27.427	0.185208
1048576	27.8258	28.306	27.033	0.398388
4194304	28.3543	28.787	27.927	0.359248
16777216	31.5564	38.621	30.407	2.49794
67108864	30.8441	31.360	30.520	0.26847
268435456	30.7283	31.353	30.427	0.267738
1073741824	30.5718	30.774	30.234	0.175753

表 C.73　test-unix50-28-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	40.4301	41.413	39.653	0.582626

表 C.74　test-unix50-28-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.3324	28.661	19.745	2.68243
4096	28.5658	30.763	27.382	1.07953
16384	28.0037	28.498	27.477	0.321631
65536	27.7749	28.364	27.345	0.343504
262144	27.87	28.328	27.380	0.365526
1048576	31.5032	32.520	30.459	0.59341
4194304	42.7803	44.147	37.844	1.80593
16777216	58.9675	68.647	56.958	3.46337
67108864	57.9602	59.511	56.527	0.83149
268435456	58.1663	60.060	56.772	1.03229
1073741824	58.7865	64.032	57.405	1.94305

表 C.75　test-unix50-28-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.3101	29.639	27.849	0.549892
4096	27.7261	28.223	27.137	0.381041
16384	27.6155	27.944	27.397	0.194912
65536	27.4402	27.858	26.971	0.26441
262144	27.6073	28.029	27.275	0.293181
1048576	27.8971	29.160	27.456	0.506714
4194304	28.514	32.835	27.388	1.56793
16777216	30.2999	38.143	29.100	2.76503
67108864	28.7242	33.214	18.317	3.85361
268435456	31.1029	56.395	18.649	9.52429
1073741824	31.1747	56.731	19.166	9.55244

表 C.76　test-unix50-29-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	7.0167	7.029	7.012	0.00485455

表 C.77　test-unix50-29-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	23.7849	23.892	23.665	0.0778267
4096	7.2165	7.625	7.164	0.143614
16384	7.0794	7.184	7.062	0.0371938
65536	7.0633	7.098	7.044	0.0202158
262144	7.0501	7.060	7.046	0.00490918
1048576	7.5419	7.595	7.508	0.022343
4194304	8.8686	8.907	8.820	0.0279253
16777216	12.0442	12.207	11.899	0.0952643
67108864	12.0032	12.407	11.805	0.165318
268435456	11.9735	12.138	11.740	0.106653
1073741824	12.0194	12.441	11.668	0.216715

表 C.78　test-unix50-29-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	23.6788	23.846	23.535	0.0986856
4096	7.169	7.189	7.163	0.00749815
16384	7.0658	7.077	7.061	0.00458984
65536	7.0532	7.067	7.047	0.00561348
262144	7.0549	7.064	7.043	0.00664078
1048576	7.1458	7.300	7.125	0.054293
4194304	7.3959	7.863	7.340	0.164165
16777216	8.6404	11.974	8.261	1.17132
67108864	8.6533	12.101	8.265	1.2114
268435456	8.6629	12.178	8.264	1.23509
1073741824	8.671	12.264	8.266	1.26246

表 C.79　test-unix50-30-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	44.3203	44.716	44.062	0.203958

表 C.80　test-unix50-30-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.6175	27.736	27.310	0.12177
4096	7.2887	7.723	7.188	0.166919
16384	7.1304	7.258	7.110	0.0450905
65536	7.1267	7.169	7.108	0.0193451
262144	7.1858	7.819	7.092	0.222873
1048576	12.1821	20.255	9.178	3.32777
4194304	18.5361	19.514	15.194	1.75839
16777216	51.7478	51.921	51.457	0.153993
67108864	51.8633	52.297	51.585	0.219806
268435456	51.7072	52.148	51.349	0.227234
1073741824	51.9095	52.073	51.658	0.143026

表 C.81　test-unix50-30-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.7266	27.898	27.579	0.106112
4096	7.1889	7.214	7.181	0.00927901
16384	7.1022	7.109	7.095	0.00388158
65536	7.1101	7.124	7.065	0.0162375
262144	7.084	7.114	7.051	0.0254296
1048576	7.0994	7.140	7.085	0.0210882
4194304	7.1822	7.351	7.158	0.0595087
16777216	10.6145	36.714	7.709	9.17043
67108864	12.1402	51.966	7.710	13.9934
268435456	7.7124	7.722	7.703	0.00455095
1073741824	12.1628	52.201	7.709	14.068

表 C.82　test-unix50-31-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	41.6531	42.093	40.988	0.387052

表 C.83　test-unix50-31-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	19.0323	19.173	18.649	0.143611
4096	7.1974	7.550	7.154	0.12398
16384	7.1079	7.205	7.085	0.0357194
65536	7.1521	7.202	7.134	0.019399
262144	8.2538	9.862	7.240	1.03229
1048576	11.4224	11.512	11.357	0.052088
4194304	26.0197	26.798	20.841	1.82284
16777216	48.5183	48.989	48.029	0.304105
67108864	48.6775	49.100	48.289	0.275291
268435456	48.7088	49.220	48.417	0.312271
1073741824	48.3429	48.600	48.188	0.145374

表 C.84　test-unix50-31-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	18.9999	19.120	18.887	0.0727071
4096	13.3334	27.986	7.157	9.93308
16384	27.6076	27.973	27.280	0.291304
65536	27.8637	28.860	27.353	0.485405
262144	28.2227	32.389	27.381	1.50061
1048576	28.0895	28.820	26.994	0.55418
4194304	29.0404	30.900	28.514	0.706349
16777216	31.7908	41.106	30.581	3.2772
67108864	32.2261	58.222	19.400	9.78813
268435456	29.4724	30.968	19.099	3.64881
1073741824	33.3353	58.281	20.044	10.2115

表 C.85　test-unix50-33-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.347	27.953	26.720	0.377302

表 C.86　test-unix50-33-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.3584	28.691	27.689	0.334649
4096	28.3826	29.116	27.823	0.394218
16384	28.2882	28.662	28.006	0.17747
65536	28.2855	28.851	27.880	0.278223
262144	29.26	29.608	28.784	0.311593
1048576	29.1559	29.579	28.733	0.270027
4194304	29.6663	33.424	28.542	1.36506
16777216	29.3397	29.829	28.285	0.48945
67108864	29.4721	29.788	28.900	0.254809
268435456	29.1771	29.566	28.767	0.297565
1073741824	29.2318	29.758	28.271	0.461565

表 C.87　test-unix50-33-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.2925	28.939	28.015	0.37346
4096	28.1312	28.675	27.588	0.311213
16384	28.1315	28.720	27.734	0.321023
65536	28.2893	28.844	27.904	0.320262
262144	28.8998	29.160	28.224	0.320165
1048576	28.5985	29.290	27.454	0.517684
4194304	28.3565	28.978	27.860	0.281553
16777216	28.3628	29.360	27.775	0.460057
67108864	28.3466	28.960	27.780	0.403089
268435456	28.2175	28.581	27.573	0.329209
1073741824	28.0099	28.428	27.676	0.303221

表 C.88　test-unix50-35-without-incrementalize

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
0	27.4411	28.206	27.034	0.345964

表 C.89　test-unix50-35-with-incrementalize-without-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	28.2914	33.088	27.396	1.70219
4096	28.2342	29.203	27.706	0.469098
16384	28.1108	28.334	27.470	0.283491
65536	28.6123	30.171	27.648	0.832097
262144	28.32	28.838	27.806	0.339349
1048576	28.7259	29.105	27.800	0.366828
4194304	28.3395	28.866	27.971	0.309241
16777216	28.4624	28.884	27.997	0.32104
67108864	29.2592	34.484	28.223	1.85653
268435456	28.6122	28.937	28.105	0.249432
1073741824	28.6984	29.037	27.963	0.308525

表 C.90　test-unix50-35-with-incrementalize-with-cache

行数	平均実行時間 (s)	最大実行時間 (s)	最小実行時間 (s)	標準偏差
1024	27.6807	27.855	27.388	0.169915
4096	27.9448	28.332	27.333	0.319108
16384	28.1921	28.384	27.818	0.182083
65536	28.3186	28.738	27.882	0.279432
262144	28.1583	28.545	27.704	0.331607
1048576	28.3324	28.605	27.747	0.242493
4194304	28.2347	28.587	27.688	0.284897
16777216	28.2685	28.721	27.787	0.320464
67108864	28.3486	28.707	27.740	0.313489
268435456	28.333	28.888	27.861	0.305531
1073741824	28.2651	28.754	27.874	0.266322