ERATO セミナ 2010 - No. 15

データインテンシブな計算方法としての分散ワークフローに関する研究の紹介

柴田剛志
東京大学大学院情報学環特任研究員

2010/8/23

概要

近年、コンピュータで処理すべきデータの量がムーアの法則を超える勢いで急激に増加している。 このような大量のデータを処理するのに適し、かつ並列計算をあまり意識せず記述できる並列計算の枠組みとして、ワークフローというものが知られている。本講演では次のことについて発表予定である。

1. ワークフローとはどのようなものか
2. GXPmake というワークフローシステムの説明と実際のデモ (Povray)
3. ファイルアクセスログと可視化について
4. ワークフローの実際の応用例をいくつか
5. 広域に散らばる計算機を使った時のワークフローのための遅延隠蔽方法

データインテンシブな
分散ワークフローに関する研究の紹介

2010/8/23
ERATOセミナー
東京大学 情報理工 田浦研究室
柴田

背景

- 多くのプログラマは並列計算に伴なう煩雑な処理を考えたくない
  - データの転送や計算の同期など
  - **簡単に分散並列計算ができるフレームワーク**が必要
- 近年コンピュータが処理すべきデータの量が非常に増えている
  - ムーアの法則を超えるファイル数やファイルサイズの増加
- データ中心の並列計算の必要性
  - データの場所が計算するクラスタやクラウドの場所と異なる
  - データの場所が複数に分散されている
  - **データの局所性が重要**となる

データ中心の並列計算フレームワーク

- MapReduce (Hadoop[1])
  - googleが内部で使うフレームワークとして論文を発表したのが始まり
  - Map と Reduce の２層からなる並列計算モデル
- ワークフロー (Dryad[2], Swift[3], Pegasus, など)
  - 複数のタスクを定義してその間の依存関係を記述
  - MapReduce の一般化という見方もできる
    - MapReduce :
      - MapとReduceの２種類のタスク
      - MapからReduceへの全体全の依存関係
    - ワークフロー :
      - ユーザが様々なタスクと依存関係を定義

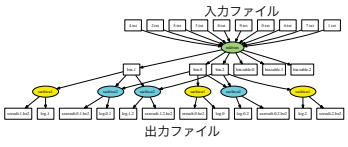
[1] http://hadoop.apache.org/
[2] http://research.microsoft.com/en-us/projects/dryad/
[3] http://www.ci.uchicago.edu/swift/index.php

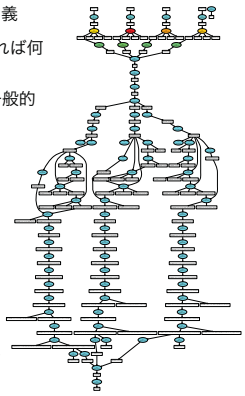
発表のアウトライン

- **ワークフローとはどのようなものか**
- デモ(Povray Animation)
- ファイルアクセスログと可視化について
- ワークフローの実際の応用例をいくつか
- 広域で使った時のワークフローに対する遅延隠蔽方法 (ongoing)

ワークフローとは

- ジョブおよびジョブ間の依存関係を自由に定義
  - ジョブは実行可能なコマンドラインであれば何でも良い
- ジョブをノード、依存関係を矢印とすると一般的に無循環有向グラフ(DAG)で表現される





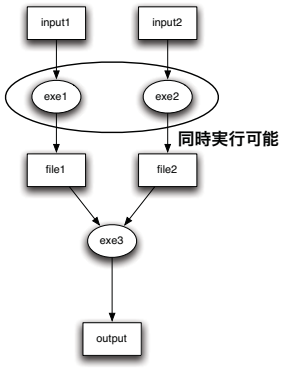
丸いのはジョブ
四角いのはファイル
依存関係が無いジョブは同時に実行できる

Makefileでワークフローを記述する

- makefileのもとの用途とは異なるが、ワークフローの記述ができる。
- make -j で並列実行もできる。

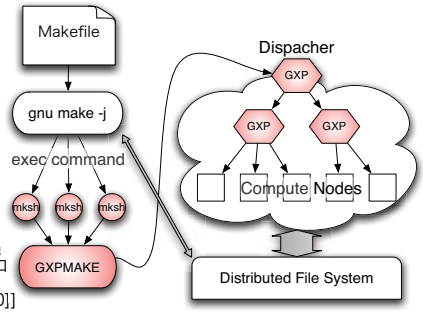
左のワークフローをmakefileで書くと：

```
output : file1 file2
    exe3 file1 file2
file1 : input1
    exe1 input1
file2 : input2
    exe2 input2
```



ワークフローシステム GXP Make

- GXP Make
  - gnu makefile で記述
  - 同時で実行出来るところは実行
  - GXPというフレームワークを使ってジョブを遠隔実行
- GXP :
  - 簡単な操作で複数の計算機を扱う仕組み
  - 例：x000 - x010の計算機を確保し、hogeというプログラムを動かしたい:
    gxpc explore x[[000-010]]
    gxpc e hoge

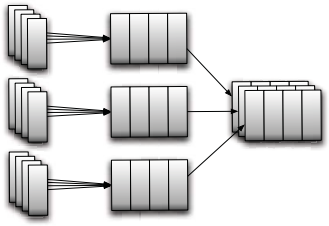


発表のアウトライン

- ワークフローとはどのようなものか
- **デモ(Povray Animation)**
- ファイルアクセスログと可視化について
- ワークフローの実際の応用例をいくつか
- 広域で使った時のワークフローに対する遅延隠蔽方法 (ongoing)

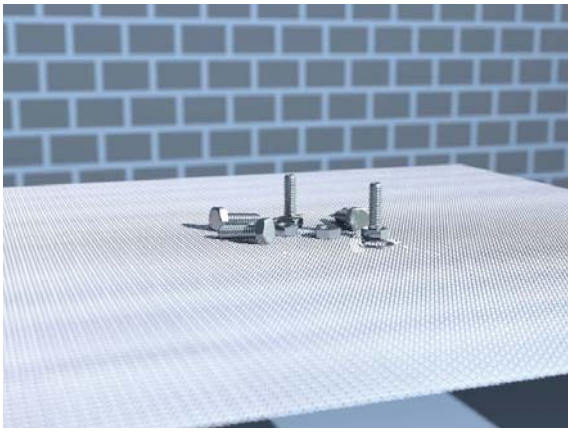
デモ：POVRAYアニメーション

- POVRAY : レイトレーシング
  - リアルだが１枚の映像を作るのに数時間以上かかる。
  - 視点を動かして１００枚の絵のアニメーションを作る。
    （逐次では数百時間かかる。）
  - ⇒1枚を20個の断片に分割して、合計2000個の断片に分割し、並列にレイトレーシングして、最後にひっつける処理をしたい。
- makefileで記述後、GXP Make で実行。



デモ：できるもの

デモ：できるもの



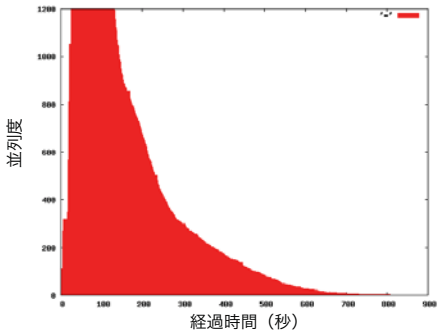
デモ：POVアニメのmakefile

- これだけの処理でも分散処理系で書こうとすると大変
- makefileでかくと数十行 ⇒ 簡単！

```
define create_cut_crop   # 1: cut , 2: crop
cut_$(1)/$(2).crop : cut_$(1)
        $(POVRAY) +Q9 +A +L$(LDIR) +Icut_$(1)/cut.pov +O- +W$(WIDTH) +H$(HEIGHT) +SC$(CALC_SC) +EC$(CALC_EC) 2>$(LOG) | \
        $(CONVERT) - -crop $(PADDING)x$(HEIGHT)+$(CALC_PD)+0 cut_$(1)/$(2).crop
endef

define mkdir_cut
cut_$(1) :
        mkdir -p cut_$(1)
        sed "s/%TIME%/`expr 50 - $(1)`/g" $(SOURCE) > cut_$(1)/cut.pov
endef

define create_cut
cut_$(1).gif : $(PARTS_IN_ACUT)
        $(MONTAGE) $(PARTS_IN_ACUT) -tile $(PARTITION)x1 -geometry $(PADDING)x$(HEIGHT)+0+0 +frame cut_$(1).png
        $(CONVERT) cut_$(1).png cut_$(1).bmp # GXP_MAKE: on=huscs
endef

all: $(CUTS)
        ffmpeg -b 5M -y -i cut_%04d.bmp boltstill.mpg # GXP_MAKE: on=huscs

$(foreach i,$(TIMES),$(eval $(call mkdir_cut,$(i))))
$(foreach i,$(TIMES),$(eval $(call create_cut,$(i))))
$(foreach i,$(TIMES),$(foreach j,$(PARTS),$(eval $(call create_cut_crop,$(i),$(j)))))
```

デモ：並列実行の様子

- 実行の様子



発表のアウトライン

- ワークフローとはどのようなものか
- デモ(Povray Animation)
- **ファイルアクセスログと可視化について**
- ワークフローの実際の応用例をいくつか
- 広域で使った時のワークフローに対する遅延隠蔽方法 (ongoing)

ワークフローによるファイルアクセスログ

- ファイルを通してジョブ同士のデータのやりとりを行う。
- DAGをつくったり、計算時間とI/O時間の比率を見たりするためにはファイルアクセスのログが必要
- Filesystem in Userspace (FUSE) というカーネルモジュールを用いてログ専用の薄いラッパーファイルシステムを作成
- 各ファイルアクセス、ホスト名、ジョブID、プロセスID
  - メタデータアクセス => 実行時刻とブロック時間を記録
  - read, write => open-closeまでのあいだの、書込み回数、サイズ、ブロック時間の合計も記録

job id	GXP_WORK_IDX (given from GXP Make)
process id	hostname:processId
operation	read, write, open, create, getattr, mkdir, etc.
data id	filename
date	execution time of each operation in nsec. from the epoc
time	blocking time during each operation
size	size for each read and write

ログの解析

- ワークフローを実行したログから、ジョブが読み書きしたファイルをもとに、DAGを作成
- プログラムごとの統計が必要
  （一つのプログラムが異なる入力ファイルに対して実行される：実行された複数のものがジョブである）
  => ジョブをクラスタリングする必要がある。
  - ジョブが同じクラスタにはいる <=> 使われたプログラムは同じだが入力ファイルが異なっている

```
DIR := some_dir
$(DIR)/%.output : $(DIR)/%.input
cmd $< > $@
```

⇒

```
cmd some_dir/1.input > some_dir/1.output
cmd some_dir/2.input > some_dir/2.output
cmd some_dir/3.input > some_dir/3.output
.....
```

  - クラスタリングの方法：
    - makefileを構文解析することで、コマンドラインの「**標準形**」を作成。標準形をクラスタの中心として、編集距離で最も近いクラスタに分類する。

```
cmd some_dir/%.input > some_dir/%.output.
```

発表のアウトライン

- ワークフローとはどのようなものか
- デモ(Povray Animation)
- ファイルアクセスログと可視化について
- **ワークフローの実際の応用例**
- 広域で使った時のワークフローに対する遅延隠蔽方法 (ongoing)

ワークフローアプリケーション

- 現実問題のアプリケーション集
  - 自然言語処理（２つ）
    - Medline to Medie
    - Case Frames Construction
  - ウェブデータ解析
    - Similar Pages Extraction
  - 天文データの解析・加工
    - Supernovae Explosion Detection
    - Montage (よく使われる既存のワークフロー)

実行環境

- InTrigger内の1クラスタ (複数クラスタは用いていない)
- クラスタ内ネットワークはギガビットイーサで構成
- 15 ノード120 コア (120 並列)
- 分散ファイルシステム : NFS

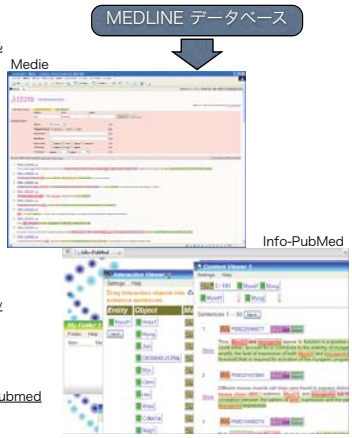
各ワークフローの規模

		Medline to Medie	Case Frames Construction	Similar Pages Extraction	Supernovae Explosion Detection	Montage
ジョブの種類の数 (クラスタリング結果)		62	29	3	9	11
ジョブの数		9821	22483	466	10512	17585
総cpu時間 [hours]		251	172	5.8	144	1.99
総I/O時間 [hours]		241	252	6.1	91	341
総I/Oサイズ[GB]	read	1514	168	77	301	339
	write	110	116	3	181	94

Medline to Medieとは?

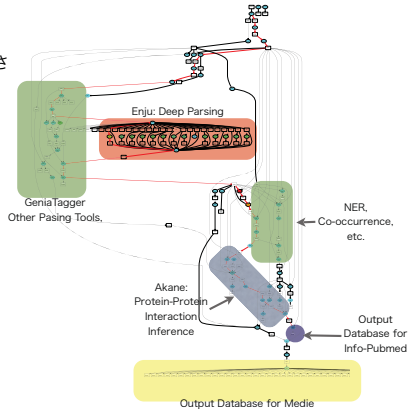
- 生物医学論文データからの自然言語処理を用いた高度な情報抽出
- 入力 : MEDLINEデータベース
  - 生物医学論文のデータベース
  - 約1800万の論文がXMLになっている
  - アブストラクトだけを用いる
- 出力 : 二つのサーチエンジンのためのデータベースを作成
  - Medie : 自然言語の深いパージングによる、意味に近い検索
    http://www-tsujii.is.s.u-tokyo.ac.jp/medie/
  - Info-PubMed : プロテイン、タンパク質名から、それぞ相互作用するプロテイン、タンパク質名を検索

http://www-tsujii.is.s.u-tokyo.ac.jp/info-pubmed



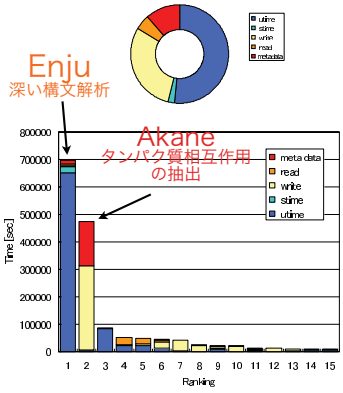
Medline to Medieのワークフロー

- 自然言語処理や機械学習のたくさんの手法を組み合わせている
  - 品詞 (POS)タグ付け
  - 固有表現抽出 (NER)
  - GeniaTagger :生物医学用語に特化したタグ付け
  - Enju : 深い構文解析
  - Akane : タンパク質相互作用の検出
- 62 種類のジョブ



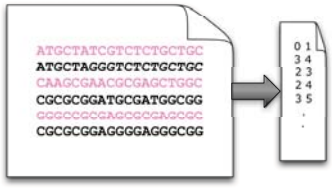
Medline to Medieの実行結果

- 入力 : 300万 (30K x 99) アブストラクト (gzipped 500MB)
- enju のみ 3584 回, 他のものは 99回
- EnjuはCPUインテンシブ
- Akane は I/O インテンシブ
- ファイルアクセスログから、次のことがわかる
  - 多すぎる1byte単位での“write”
  - 不必要な “open” と “close”
- **実行ログから改善すべき点**がわかる



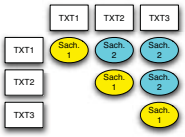
SimilarPagesExtractionとは

- 入力 : ウェブから集めた文
- 出力 : 全てのテキスト中の「似ている」文のペアすべて
  - 「似ている」 <=> 2つの文のハミング距離がしきい値以下
- Sachica (T.Uno, PAKDD2008)
  http://sites.google.com/site/yasuotabei/sachica
  - 固定長の文字列の集合から、しきい値以下のハミング距離を持つペアを全て列挙（高速に）



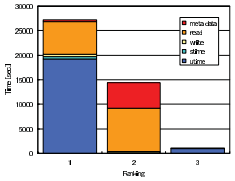
SimilarPagesExt.のワークフロー

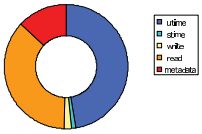
- 3種類のジョブ
  - Mkbins : 全ての入力ファイルを一度一つにまとめて、N個のファイルに分割する。
  - Sachica1 : 一つのファイル中の類似文を全て列挙
  - Sachica2 : 二つのファイル間の、類似文を全て列挙
- N x Sachca1,
  N(N-1)/2 x Sachca2



SimilarPagesExtractionの実行結果

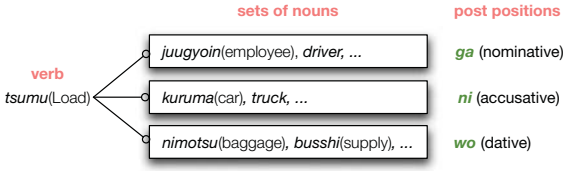
- 100万個の入力ファイル (5GB)
- 分割数 N=30;
  - 30 個の中間ファイルがそれぞれ30回読まれる
- 実験環境では全体の半分がI/O時間だった
- Mkbins (gather and split files) が総消費時間リストの２番目
  - 100万ファイルに対して 1.5万秒 => 60 ファイル/秒の処理速度
- 'read' がI/O時間の主要であることが右円グラフからわかる
  - 各中間ファイルがN回よまれることから、中間ファイルを**キャッシュ**し、**局所性を考慮したスケジューリング**が有効と考えられる





CaseFramesConstructionとは

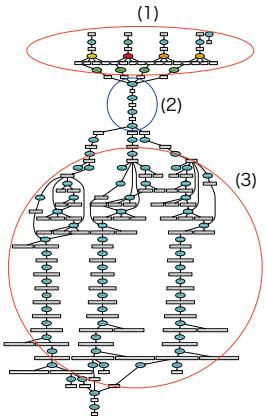
- 「格フレーム(case frame)」と呼ばれるデータ構造を、ウェブの大量文章から構築する自然言語処理のタスク
- 格フレームとは、 頻出する名詞と動詞の格ごとの共起を記述したもの



- 川原らはウェブコーパスから並列計算で9万の動詞に対して格フレームを構築(LREC2006)

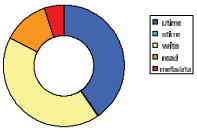
CaseFramesConst.のワークフロー

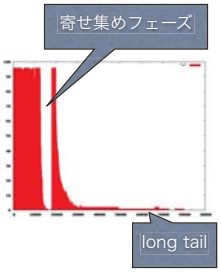
- 概要 :
  - (1: 構文解析フェーズ)
    入力ファイルは分割され、並列に構文解析される（品詞タグ付け、係り受け解析）
  - (2: 寄せ集めフェーズ)
    上記で出力された動詞と名詞の係り受けのリストを全て集めて、動詞の辞書順にソートして一つのファイルに書き込む。
  - (3: 格フレーム作成フェーズ)
    上記のファイルを適当なサイズに分割して、各動詞ごとに格フレームを作成する。
- 右図から、(3)は枝分かれはあまりしないパスが複数集まっていることがわかる。
  => (3)ではファイルの局所性を考慮するスケジューリングがしやすいことがわかる。



CaseFramesConst.の実行結果

- 実験環境では半分以上がファイルアクセスだった
  - 'write' が主要なI/O時間
- 並列度と経過時間の表（右下）
  - 寄せ集めフェーズで並列度が落ちている
  - 格フレーム作成フェーズ(3)ではロングテールとなっている
    - ファイルの局所性やクリティカルパスを考慮したスケジューリング（いまはランダムスケジューリング）が求められる。

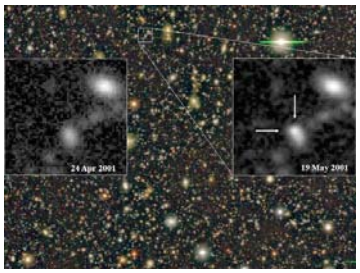




SupernovaeExplosionDetectionとは

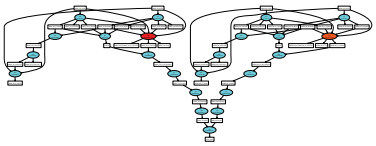
- 天文データから超新星(supernova)と思われる星の座標を列挙
- 異なる日に撮られた２枚の写真から推定
- IEEE Cluster/Grid 2008でデータ解析チャレンジ問題として出題されたもの

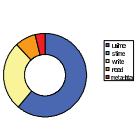
http://www.cluster2008.org/challenge/

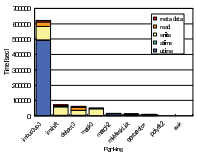


Supernovaeのワークフローと結果

- 入力 : たくさんの天文画像ファイルのペア
- 出力 : 超新星と思われる場所のリスト
- 各ファイルペアごとに一つの主要なプログラムが存在
- 実験結果では、 他のものと比べてI/OよりもCPUが主な消費時間
- 一つのペアごとにローカリティを考慮した計算ができるようなスケジューリングがよい





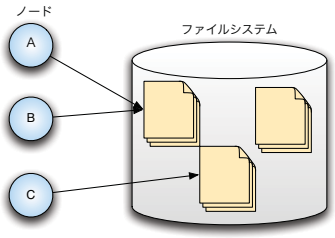


発表のアウトライン

- ワークフローとはどのようなものか
- デモ(Povray Animation)
- ファイルアクセスログと可視化について
- ワークフローの実際の応用例をいくつか
- **広域で使った時のワークフローに対する遅延隠蔽方法 (ongoing)**

分散ファイルシステムとは

- 計算機によらず、共通のディレクトリやファイルがみえるようなファイルシステム。
- 各計算機からのアクセスは普通のファイルと全く同じ。
- NFS, Lustre, gfarm などがよく知られている。



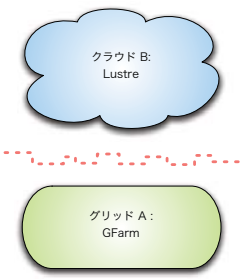
ワークフローと分散ファイルシステム

- GXPMakeのようなワークフローシステムには分散ファイルシステムが不可欠
- ジョブ間のデータの受け渡しは主にファイルを通じて行われる
  - 依存関係 : A -> B
    - Aが出力したファイルをBが入力として受け取る
    - Aがファイルやディレクトリに変更を加え、そのことを情報としてBが利用する
- 入力ファイルと出力ファイルを明示的にユーザーが書く必要がない
- 全ての計算ノードにまたがる分散ファイルシステムが必要
- e.g. Pwrake[4], GXPMake[5]

[4] http://github.com/masa16/pwrake
[5] http://www.logos.ic.i.u-tokyo.ac.jp/gxp/

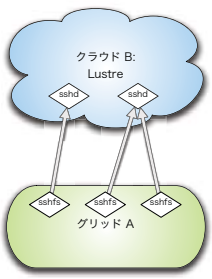
複数のクラウドやクラスタを使う

- 一般的に複数の分散ファイルシステムをつなぎあわせて使うことは難しい
  - 異なるファイルシステムと協調することは考えられていない
  - 管理者権限が必要
  - 複雑なインストールの手間
  - ファイアーウォールの問題
- 複数のクラスタやクラウドを使うときに問題



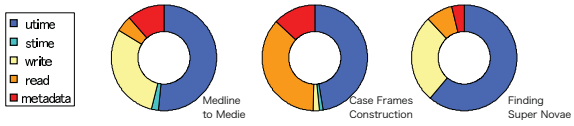
遠隔マウントという方法

- 遠隔マウント :
  - FTPやSFTPなどのプロトコルを通して遠隔のファイルシステムをマウントする
- 遠隔マウントの利点
  - 遠隔のクラスタやクラウドでどのようなファイルシステムを使っていても良い
  - そのプロトコルで接続可能なノードが少なくとも一つ存在すれば良い
- 例）SSHFS : 必要となる条件
  - リモート :
    - SSHサーバがインストールされている
    - SSHのポートが開いている
  - ローカル :
    - FUSEが使用可能な設定になっている



問題点

- ファイルを転送するだけの場合に比べて、遅延の影響を何倍も受ける
  - メタデータサーバやリモートホストと同期を取る必要のある多数のメタデータアクセスが存在
    - stat, open, flush, create, mkdir, ... ,
    - 通常ファイルシステムの整合性を保つためには同期が必要
- 過去に実行したいくつかのワークフローの結果では、CPU時間とIOブロックの合計時間のうち 4 ~ 32% がメタデータ処理
  - 実験環境は 1 クラスタ内のNFS
- 複数のクラスタやクラウドを使うと、遅延が飛躍的に大きくなることからメタデータ処理の占める割合はかなり大きくなると予想される



ワークフローに必要な整合性

- 各操作で同期を取る必要
  - ファイルシステムの整合性の維持
    - e.g., createした直後に他のノードがそのファイルを読める
- ワークフローの正常な実行を保証したいもっと弱い整合性で十分

各ジョブBについて、Bが依存しているジョブ全て（A->B となる A 全て）が行ったファイルへの変更が、Bの実行前に正しく反映されている

  - 理由：「ジョブ間の依存関係」と「ジョブの実行順序」に関する定義から保証可能

ジョブAとジョブBは並列に実行されない

⇕

ジョブAからジョブBに依存関係がある (A->B)

⇕

Aが行ったファイルへの変更をBが情報として使う

提案手法

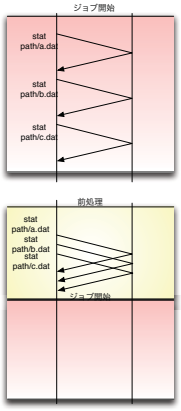
- ジョブ実行に際して次を行う
  - 事前処理：使われる可能性の高いファイルのリストを分散ファイルシステムに教える
  - 非同期化：open, flushなどのメタデータ操作を非同期に行う
  - 事後処理：非同期に行った操作の完了を待つ

```
filelist = {使われそうなファイルのパス(予想)}
fs_connection.pre_job (filelist)
job.set_week_consistency_mode ()
job.exec ()
fs_connection.post_job ()
```

ジョブを実行するときの擬似コード

提案手法：事前処理

- ワークフローシステム側：
  - 入出力として使われそうなファイルのリストをジョブの実行直前に分散ファイルシステムに教える
- 分散ファイルシステム側：
  - 事前に受け取ったパスのリストに対して、まとめてメタデータを問い合わせる（遅延が発生）
  - 存在するパスについてのみメタデータが返答されるので、それらをキャッシュしておく

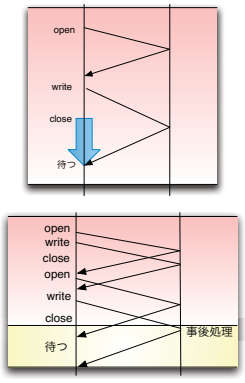


提案手法：非同期化

- ワークフローシステム側：
  - ジョブが生成する子プロセス全てに対して、次が何らかの形でわかるようにしておく。
    - 非同期化を行うべきときはそうとわかる
    - どのジョブから生成されたものかがわかる
  - e.g., 環境変数に JOBID=X, WORKFLOW_MODE=YES
- ファイルシステム側：
  - あるファイル操作が呼び出されたとき、次が満たされれば非同期に行う：
    - 呼び出したプロセスIDから非同期化を行うべきというフラグが立っている。
    - その操作が、openやflushなど、ジョブ単位の整合性では非同期に行えるものである。
    - open等はメタデータを見て挙動を決定
  - プロセスIDから得られるジョブのIDに対して、非同期に行った処理を登録しておく。

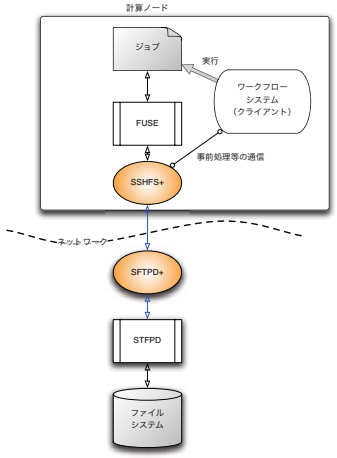
提案手法：事後処理

- ワークフローシステム側：
  - ジョブ終了時にファイルシステムにそのことを教える
  - ackをまつ
- ファイルシステム側：
  - 同期を取る、つまり、ジョブ終了時に非同期に行った処理で未完了のものがなくなるまで待つ



SSHFSに対する実装

- 提案手法を遠隔マウント用のファイルシステムであるSSHFSに対し実装
- SSHFSの仕組み：起動時にSSHを通してSFTPサーバを起動する
- 大枠部分の変更の概要：
  - ローカルから接続を受け、事前処理や事後処理に関して通信できる
  - SFTPの手前に中継サーバを設ける
    - ファイルハンドルの変換に必要

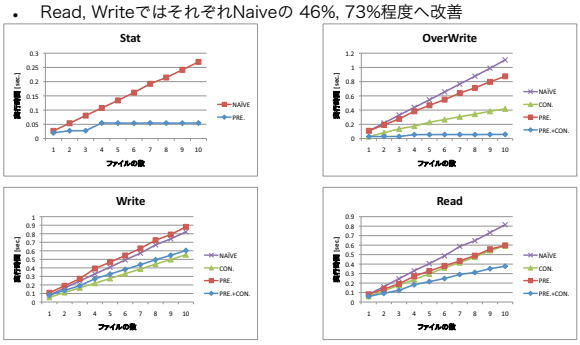


実験

- 次のような４つのジョブを用意: n個の小さなファイルに対して順次、
  - メタデータを得る (Stat)
  - 読み込む (Read)
  - 新規作成し書き込む(Write)
  - 既存のファイルに上書きする(OverWrite)
- 提案手法の関しては次の4つについて実験
  - NAIVE : 元のSSHFSと同じ
  - PRE : 事前処理において、n個のファイルのパスを全て教える
  - CON : open, flush, create等の非同期化を行う
  - PRE+CON : 上記二つを行う
- 実行時間の計測：ワークフローシステムが事前処理を行う直前から、事後処理を行った直後まで

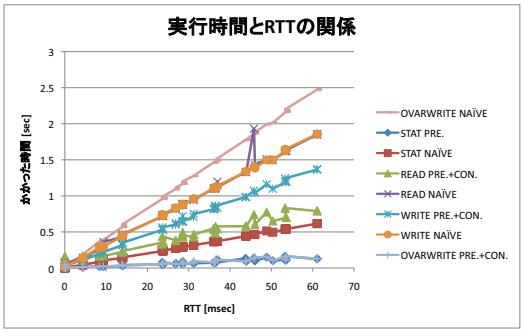
実験の結果

- RTTが27msecあるクラスタ間でSSHFSで遠隔マウント
- Stat, OverWriteではファイル数にかかわらずほぼ定数時間に
- Read, Writeではそれぞれ Naiveの 46%, 73%程度へ改善



遅延と実行時間の関係

- 九州から北海道までの７クラスタの２点間で実験(49通り)
- ファイル数は１０



まとめと今後の予定

- まとめ
  - 高遅延環境下ではファイルシステムのメタデータアクセスは大きなオーバーヘッド
  - ワークフローの正常な実行に必要とされるファイルシステム整合性はかなりゆるい
  - ワークフローのジョブからのアクセスの場合、整合性をゆるめることができるから、いくつかのメタデータアクセスは非同期に行うことが可能
  - 実験結果からは、提案手法により、遅延が支配的なときに大幅な実行時間の削減が達成できることがわかる
- 今後の予定
  - ログを用いた各JOBが入出力するファイルやディレクトリの予想
  - **実際のワークフローに適用**