Vol.2016-HPC-155 No.11
2016/8/8

ワークフローシステムPwrakeにおける耐障害機能

田中 昌宏[1,a)]　建部 修見[1]

概要：Gfarm ファイルシステムのスケーラブルな並列 I/O 性能を活用しつつ，メニータスクワークフローを効率よく実行するため，我々は Pwrake というワークフローシステムを開発している．本稿では，ワークフローシステムに求められる耐障害性を Pwrake において実現するための設計および実装について述べる．ワーカーノードの障害に対しては，Gfarm のファイル自動複製作成，および，Pwrake によるタスク再実行により，ワークフローの継続を可能にする．また，中断したワークフローを途中から再開するためのチェックポイントとして機能させるため，タスク失敗時に出力ファイルをリネーム・削除するオプションを導入する．評価実験では，Gfarm ファイル自動複製によるワークフロー実行時間への影響が 10% 以下であり，ワーカーノードにおいて擬似的に発生させた障害に対してワークフローが継続することを確認した．

1. はじめに

観測装置の進化により生み出される大量の科学データを活用したサイエンスを推進するため，大規模データの並列処理基盤が必要とされている．科学データのパイプライン処理では，大量の入力ファイルに対して，複雑な依存関係を持つ複数ステップからなる処理が必要になることが多く，こうした依存関係を持つタスクからなる処理の流れは「ワークフロー」と呼ばれる．ワークフローを並列計算機システムにおいて効率よく並列実行するために，ワークフロー実行基盤システムが必要となる．スパコンにおいて一般的に用いられている PBS などのバッチスケジューラは，ジョブ投入のオーバーヘッドがあるため，短時間のプロセスを大量にを起動する目的には向いていない．このような多数のタスクを大量に並列実行するカテゴリとして，Many Task Computing (MTC)[1] が提唱されている．MTC をターゲットとするワークフローの処理系として，Pegasus[2]，Swift[3], [4]，GXP make[5] などがある．

我々は，MTC 向けワークフローシステムとして，Pwrake[6][*1] を開発している．Pwrake は，ワークフローの記述力が高い Rake というツールをベースに，並列分散実行の機能を拡張したワークフローシステムである．特にデータインテンシブなワークフローを目的として，Gfarm ファイルシステム [7] を利用し，ローカリティを高めるスケジューリング [8] により，スケーラブルな並列 I/O 性能による処理を可能にする．

ワークフローシステムに求められる機能の 1 つに，耐障害性が挙げられる．大規模な計算機クラスタで多数の計算ノードを使用した並列処理を実行する場合，実行中に計算ノード停止，ネットワーク切断などの障害が発生することが予想され，ノード数が多いほどその確率も高くなる．大規模システムでワークフローを実行する場合，こうした障害が起きた場合でも，ワークフローを続行できることが求められる．Globus toolkit などの大規模システムにおける分散環境の統合的なミドルウエアでは，障害検知や回避を行う機能を提供しており，いろいろなワークフローシステムから利用されている．一方の Pwrake では，統合的なミドルウエアを用いるのではなく，Gfarm ファイルシステムにおける耐障害機能を活用しつつ，タスクの再実行や故障ノード脱退の機能を取り入れることにより，軽量なシステムでありながら，一定の耐障害機能を実現することを目指す．

本稿の構成は以下の通りである．2 節で Pwrake の概要について延べ，3 節で Gfarm ファイルシステムが持つ耐障害性について述べる．4 節で Pwrake の耐障害性の設計について述べ，5 節で耐障害性のための Pwrake の実装について述べる．6 節で耐障害機能が性能に与える影響について評価する実験について述べ，7 節で関連研究，8 節でまとめについて述べる．

2. Pwrake ワークフローシステムの概要

Pwrake (Parallel Workflow extension for RAKE) は，Ruby 版の Make である Rake を，並列分散実行できるよう

[1] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] tanaka@hpcs.cs.tsukuba.ac.jp

[*1] https://github.com/masa16/pwrake

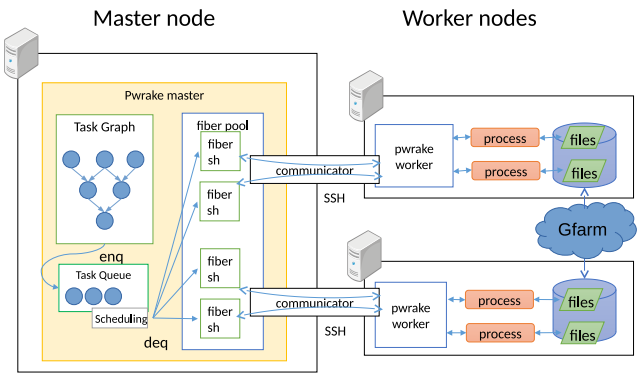

図 1 Pwrake 概要

に拡張したツールである．Pwrake 用のワークフロー記述言語として，高い記述性を持つ Rakefile の仕様をそのまま引き継いでいる．手軽に並列分散処理を実現するツールでありながら，研究室のコモディティクラスタから大規模なクラスタまでスケールする性能を目標としている．Pwrake では，次のような既存の技術を組み合わせて利用することにより，軽量なシステムを実現している．

- ワークフロー記述言語：Rake
- ノード間のファイル共有：Gfarm
- リモートプロセス実行：SSH

Pwrake の概要を**図 1** に示す．Pwrake のノード構成は，Pwrake プロセスを起動する 1 つのマスターノード，および，タスクの外部プロセスを起動する複数のワーカーノードからなる．Pwrake は起動時にワーカーノードに SSH で接続し，外部プロセスの実行を管理するワーカープロセスを起動する．このときの SSH の標準入出力をマスター・ワーカー間の通信路として使用する．一方，Pwrake プロセス内では，図に示すように，ワーカーノードのコアに 1 対 1 対応したワーカースレッドを起動している（Pwrake version 2 では，ワーカースレッドの実装を，Ruby の Thread ではなく，軽量スレッドである Fiber を用いた実装に変えている）．アイドルなワーカースレッドは，順次タスクキューからタスクを取り出し，タスクアクションと呼ばれるコードブロックを実行する．タスクアクション内の sh メソッドの引数には，外部プロセスとして実行するコマンドラインが与えられる．このコマンドライン文字列が SSH の通信路を通ってワーカープロセスに送られ，ワーカーノードにおいて実行される．

3. Gfarm ファイルシステムの耐障害性

Pwrake では，ワーカーノード間のファイル共有については，分散ファイルシステムの使用を想定している．特に，並列ファイル I/O に関して高い性能を得るため，Gfarm ファイルシステムに対応する機能を取り入れている．

Gfarm ファイルシステムは，メタデータサーバ（MDS）と，ファイルシステムノード（FSN）により構成される．MDS は，ファイルやディレクトリツリーの情報を集中管理するサーバである．FSN は，ファイルの内容をローカルストレージに保存するノードであり，計算ノードに設置することを想定している．Pwrake でも FSN をワーカーノードとする使い方を想定している．クライアント側からは，Gfarm コマンド群や gfarm2fs による FUSE マウントを用いてファイルシステムにアクセスする．

Gfarm ファイルシステムの耐故障性については，すでに単一障害点なしにすることが可能となるように設計・実装されている．MDS の障害に対しては，マスター MDS の他にスレーブ MDS を設置してメタデータを冗長化することにより，無停止での運用を可能にする．また，FSN の障害に対しては，障害ノードを除外して継続運用が可能である．FSN に保存したファイルについては，ファイル自動複製作成機能により，ファイル消失を防ぎ，継続的なアクセスを可能にする．

4. Pwrake 耐障害機能の設計

Pwrake の開発では，これまでファイル I/O 性能に着目したスケジューリング，スケーラビリティに着目してきており，Pwrake の耐障害性については十分に考慮されていなかった．ここではまず Pwrake に必要な耐障害機能を設計する上での方針について述べる．

分散環境の統合的なミドルウエアでは，通常，障害検知や回避を行う機能を提供する．一方，Pwrake では，軽量性を維持するため，統合的なミドルウエアを用いるのではなく，Gfarm ファイルシステムにおける耐障害機能を活用する方針とする．Pwrake には，ワークフローを実行する上で必要とされる耐障害機能のみを実装し，特殊なクラスタ監視システムなどが必要な機能は取り入れない．Pwrake の耐障害性についての設計の概要は次の通りである．

(1) ワーカーノードに障害が起きた場合，そのノードを脱退させ，残りのノードでワークフローを続行させる機能を Pwrake に実装する．生成したファイルの消失を回避するため，Gfarm のファイル自動複製作成機能を活用する．
(2) マスタノードに障害が起きた場合，Pwrake プロセスが終了しワークフローが中断する．ワークフローを再開するには，障害を避けて Pwrake を再実行する．再実行のとき，出力ファイルが生成されたタスクについてはスキップするという Pwrake の仕様により，ワークフローの途中からの再開が可能である．
(3) 障害ではなく，タスクに不具合があって実行が失敗する場合，そのタスクの結果に依存するタスクの実行は行わない．ユーザがプログラム修正などの対処を行い，ワークフローを再実行させる．

設計の具体的な内容について以降で述べる．

4.1 ワーカーノード障害

4.1.1 ワーカーノード障害の検知

Pwrake では，ワーカーノードに障害が発生した場合，そのノードを脱退させてワークフローを続行させる方針とする．ワーカーノードの障害として，次のものが含まれる．

- ハードウェア障害
- ネットワーク障害
- OS の不具合
- 動作環境の設定ミス

軽量システムである Pwrake では，こうしたワーカーノードの障害の原因まで検知する機能や，障害から復帰する機能は持たず，単純にワーカーノードを脱退させることとする．

Pwrake においてワーカーノード障害と判定するのは，以下のケースである．

- 同一ノードにおけるタスク実行の連続失敗
- ワーカーノードとの通信切断
- ハートビートのタイムアウト

タスクの実行失敗については次の 4.1.2 節で述べる．ハートビートの実装については 5.2 節で述べるが，ワーカーノードから heartbeat を一定間隔で送信するという一般的な方法を用いる．

4.1.2 タスクの実行失敗とリトライ

Pwrake では，ワーカーノードで障害が起きたことを判定する手段の 1 つに，タスクの実行に失敗したことを用いる．しかし，タスクの実行失敗の原因として，ノードの障害以外に，タスク自身に不具合がある場合も考えられる．そこで，タスクを別のノードで再実行（リトライ）し，その結果によって判断する．

まず，Pwrake においてタスクの実行に失敗したことを判定する方法について述べる．Rake では，タスクのアクション実行中に例外が発生すると，そのタスクが失敗したと判定する．例外が起こるケースのうち，ワーカーノード障害が原因である可能性があるのは，次のいずれかである．

- sh から起動したプロセスの終了ステータスが非ゼロ．
- 出力ファイルが生成されていない．

1 つ目のケースにおいて問題となるのは，プログラムが終了ステータスを正しく返さないかもしれないことである．そのような場合，Rake の仕様である sh の後処理ブロックを用いて対処できる．ワークフローの Rakefile を記述する際，sh にコマンドラインに加えて，コードブロックを与えると，プロセス実行後，終了ステータスを引数としてコードブロックが呼ばれる．ここでタスクを失敗させたければ例外を上げ，失敗させたくなければ例外を上げずに成功とすることができる．

また，上記 2 つが矛盾するケースのうち，タスクが失敗し，かつ出力ファイルが生成されている場合は，ワークフローの再実行の際に問題が起きるため，4.2 節で述べるような後処理を行う．

タスクが実行に失敗した場合，別のノードでリトライを行い，その結果によって次のような判断を行う．

- 同じタスクが連続失敗　→　タスクに不具合
- 同じノードで連続失敗　→　ノード障害

タスクに不具合があると判定された場合は，そのタスクの結果に依存するタスクが実行不能となる．Pwrake ではオプションによって次の動作を選択可能とする．

(1) タスク失敗以降は新しいタスクを起動せず，現在実行中のタスクの終了を待ち合わせて終了．（デフォルト）
(2) 失敗したタスクとの依存関係がないタスクについては以降も起動し，すべての実行が完了してから終了．
(3) 実行中のすべてのタスクに kill シグナルを送信し，直ちに終了．

4.1.3 ファイル自動複製作成

3 節で述べたように，Pwrake では，Gfarm のファイルシステムノード（FSN）をワーカーノードとして使用することを想定している．Gfarm の FSN に障害が起きた場合，そのノードを除いて運用を続行できるように Gfarm ファイルシステムは設計・実装されている．しかしそのノードに格納されているファイルにはアクセスできなくなる．

ノード障害によるファイル消失やアクセス不能を回避するため，Pwrake では Gfarm ファイルシステムによる自動複製作成機能を利用する．自動複製作成機能とは，ファイルを書き込んでクローズした後，自動的に別のノードに複製を作成する機能である．Gfarm クライアントプログラムの gfncopy コマンドを用いて，ファイルやディレクトリに複製数を設定できる．FSN に障害が発生した場合でもワークフローの続行を可能にするには，ワークフローを実行するディレクトリについて複製数を 2 以上に設定し，書き込まれたファイルの複製が別の FSN に自動的に作成されるようにする．これにより，ワークフロー実行中にファイルを格納する FSN の 1 つがダウンした場合でも，複製先の FSN が正常に稼働していればファイルアクセスが継続でき，ワークフローを続行できる．

4.2 マスターノード障害

Pwrake プロセスが動作するマスターノードに障害があり，Pwrake が異常終了すれば，ワークフローが終了する．そのような場合，障害を回避して Pwrake を再実行する役割は，Pwrake を起動する上位の層，例えばバッチスケジューラが担うことを想定する．

中断したワークフローの途中から再実行することは，以下に述べるような Pwrake の仕様によって可能である．Pwrake は Make と似たビルドツールの Rake をベースとしており，出力ファイルが入力ファイルより新しい場合にそのタスクをスキップするという仕様を引き継いでいる．そのため，ワークフローの途中で中断した場合，同じ Rakefile

に対して Pwrake を再実行すると，出力ファイルが生成されたタスクまではスキップし，それ以降のタスクについて処理を開始する．すなわち，Pwrake では中間ファイルがチェックポイントの役割を担い，それを記録するファイルシステムがタスク実行状況のデータベースに相当するといえる．この点においても，耐障害性を持つ Gfarm ファイルシステムが適している．

ここで述べたワークフローの再実行には 1 つ問題がある．それは，「タスクが実行に失敗し，かつ出力ファイルを生成している」という場合，出力ファイルが不完全である可能性があるにもかかわらず，ワークフローを再実行すると，このタスクを実行せずスキップしてしまうことである．この仕様の基となった Make では，想定するタスクがプログラムのビルドであり，そのようなプログラムは通常エラーの時はファイルを出力しないように設計されている．一方，科学ワークフローでは，プログラムが失敗したときに出力ファイルを消すように設計されているとは期待できない．そこで，Pwrake に次のような仕様を追加する．4.1.2 節で述べたタスク失敗判定において，「プロセスが終了ステータスでエラーを示し，かつ出力ファイルが生成されている」場合は，出力ファイルに対して，オプションによって次のような後処理を行う．

(1) .__fail__という文字列を加えたファイル名にリネームする（デフォルト）
(2) 削除する．
(3) 何もしない（ワークフローの再実行対策なし）

5. Pwrake 耐障害機能の実装

Gfarm ファイルシステムにおけるファイル複製機能，および，Pwrake における出力ファイルが生成されているタスクをスキップする機能は以前から備わっている．本稿では，新しく実装した耐障害機能について述べる．実装した耐障害機能は主にエラー発生時の処理の強化であり，これによる正常時の処理性能への影響はないと考える．

5.1 タスクのリトライ

Pwrake には，Rake の `Task` クラスに対して Pwrake 用の情報を追加する `TaskWrapper` クラスがある．リトライ回数の管理は，この `TaskWrapper` クラスのインスタンス変数 `@n_retry` で行う．`@n_retry` の変数の初期値はリトライ回数である．タスクの処理が終了したとき，`TaskWrapper` クラスの `retry_or_subsequent` メソッドが呼ばれる．ここでタスクの終了状態を確認し，エラーのとき `@n_retry` の値によってリトライか失敗か決める．`@n_retry` $= 0$ のときはこのタスクを失敗としてリトライしない．`@n_retry` > 0 のときはこの変数値をカウントダウンさせ，Array のインスタンス変数 `@tried_hosts` に実行に失敗したホスト名を記録し，再びタスクキューに投入する．スケジューリングによって `@tried_hosts` に含まれるワーカーノードでは実行されないようにする．

5.2 ハートビート

4.1 節で述べたハートビートの実装について述べる．ワーカーからは指定されたハートビート間隔で heartbeat コマンドをマスターに送信する．マスターでは，Ruby の I/O 多重化メソッド `IO.select` のタイムアウトをハートビート間隔に設定してワーカーからのデータ受信を待つ．`IO.select` で設定するタイムアウトは，ハートビートタイムアウトの検出には不十分である．というのは，ワーカーノードごとに複数の IO からデータを受信するため，いずれか 1 つの IO オブジェクトがハートビートを返さなくても，他の IO がデータを返し続ければ `IO.select` でタイムアウトは発生しないからである．そこで，`IO.select` の待ちから戻ったとき，いずれかのワーカーノードがハートビートタイムアウトになっていないかチェックを行う．具体的には，その時点で最も長く待ち状態にある IO について，待ち時間とハートビート間隔を比較し，待ち時間の方が長いときにハートビートタイムアウトと判定する．

5.3 ノード脱退

次のいずれかの場合，接続先のワーカーノードに障害が起きたと判断し，ノード脱退処理を行う．

- 同一ノードでのタスクの実行に連続失敗（4.1 節）
- ハートビートのタイムアウト（5.2 節）
- SSH 接続に失敗
- SSH 接続の IO がエラー

ノード脱退処理として，ワーカーノードのリストから該当ノードを削除，該当ノードを担当する Fiber スレッドの終了処理，LocalityAwareQueue の内部のノードキューの終了処理などがある．

6. 評価実験

Pwrake の耐障害性機能によるワークフローの性能への影響について，次の 2 点について評価を行った．1 点目は，Gfarm の自動複製作成によるワークフロー実行時間への影響である．2 点目は，ワーカーノードに障害が発生した時のワークフローの継続性と性能への影響である．

6.1 評価環境

評価は筑波大学 HPCS 研究室のクラスタで行った．評価に使った計算機環境を**表 1** に示す．Gfarm の FSN を兼ねるワーカーノード 8 台の他，Gfarm の MDS と Pwrake を起動するマスターノードを兼ねるノードを 1 台使用した．評価に用いた Ruby のバージョンは 2.3.0，Pwrake のバージョンは 2.1.0，Gfarm のバージョンは 2.6.11 である．

表 1　評価環境

クラスタ	HPCS 研究室クラスタ
OS	CentOS 6.8
CPU	Intel Xeon E5620 (2.40GHz)
	4 cores × 2 cpus
主記憶容量	24 GB
FSN ストレージ	HDD
ネットワーク	Gigabit Ethernet
ワーカーノード数	8

表 2　Montage ワークフロー

入力画像ファイル	2MASS
入力ファイル数	309
入力ファイルサイズ合計	639 MB
出力ファイル数	3,675
出力ファイルサイズ合計	6,980 MB
全タスク数	2,252

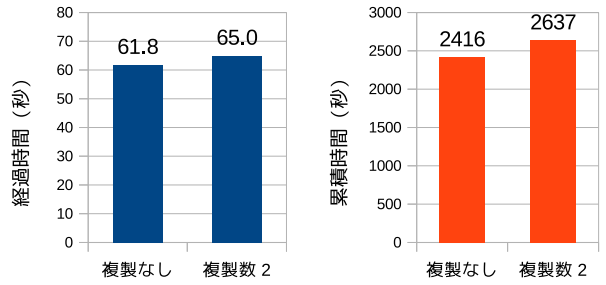

図 2　複製作成によるワークフロー実行時間

6.2　評価ワークフロー

本稿の評価は，天文画像合成処理の Montage[9] ワークフローを用いて行った．評価で実行したワークフローについて**表 2** に示す．ワークフローの Rakefile は，GitHub のサイト[*2] から取得できる．

6.3　Gfarm ファイル複製のオーバーヘッド

まず，Gfarm の自動複製作成によってワークフロー実行時間にどのような影響を与えるか実験する．実験では，ワークフローを実行するディレクトリに対して，複製を作成しない場合と，自動複製作成数を 2 に設定（`gfncopy -s 2`，つまりファイル出力後他のノードに複製を 1 つ作成）した場合について，Pwrake による Montage ワークフローの実行時間を測定した．測定は 4 回行い，結果の平均値を**図 2** に示す．図における経過時間は，ワークフロー起動から終了までの経過時間であり，累積時間は，Pwrake が起動した全プロセスの経過時間の合計である．実験の結果，複製作成なしと比較して自動複製作成数を 2 とした場合に，ワークフローの経過時間が約 5% 増加し，累積時間が約 9% 増加した．今回は使用したネットワークが 1 Gbps Ethernet であり，より高速なネットワークの場合は違いはさらに小さくなる．

[*2] https://github.com/masa16/pwrake-demo

6.4　ワーカーノード障害

ワーカーノード障害の実験として，ワークフロー実行中，8 台のワーカーノードのうちのある 1 台について，次の 3 つのうちのいずれかの方法で擬似的に障害を発生させる実験を行った．

(1) Pwrake のワーカープロセスを kill する．
(2) FSN のデーモンプロセス gfsd を kill する．
(3) ノード内のユーザ所有プロセスを全て kill する．

実行したコマンドは，それぞれ次の通りである．

(1) `kill -KILL` [Pwrake worker process ID]
(2) `pkill -KILL gfsd`
(3) `kill -KILL -1`

実験に使用した Gfarm ファイルシステムはユーザ権限で構築したプライベートシステムであるため，(3) で kill されるプロセスには gfsd も含まれる．実験では，ワークフローのディレクトリに対する複製数を 2 に指定（`gfncopy -s 2`）し，Montage ワークフローを実行した．その実行中，ワーカーノードのうちの 1 台において，手動で前述の 3 種類の kill を行った．実験の結果，いずれのケースでもワークフローは最後まで正常に動作し，kill を行わない場合と同じ最終画像が得られた．

この実験において，Pwrake 起動からの経過時間に対するプロセス数の推移のグラフを**図 3** に示す．いずれの場合も，Pwrake 起動から約 20 秒後に，前述の擬似障害を発生させている．(1) の Pwrake ワーカーを kill したケースと，(3) の全て kill したケースでは，途中から最大プロセス数が 64 から 56 へ，1 ノード 8 コア分だけ減少し，その後は残りのワーカーノードで実行されていることがわかる．（経過時間が 30 秒台後半から 50 秒付近まで，プロセス数が 1 となるのは，Montage ワークフローの仕様により，mBgModel の 1 プロセスが逐次で実行されるためである．）

一方，(2) の gfsd のみを kill したケースでは，プロセス数が 64 のまま減っていない．これは Pwrake のワーカープロセスが動作しているためである．しかし，gfsd が動作しないノードではローカルストレージへの読み書きができなくなる．このことが実行時間にどう影響するかを見るため，この実験におけるワークフロー実行時間を**図 4** に示す．(2) のケースでは，最大プロセス数が 64 から減っていないにも関わらず，ワークフローの実行時間，特に累積時間が増加している．この理由として，gfsd を kill したノードではファイルアクセスが常にリモートとなり，ファイル I/O にかかる時間が増えたことが考えられる．

gfsd が kill された (2) と (3) のケースでは，kill 以前に障害ノードに格納されたファイルにはアクセスできなくなっている．しかし，ワークフローの実行結果が正常に得られ

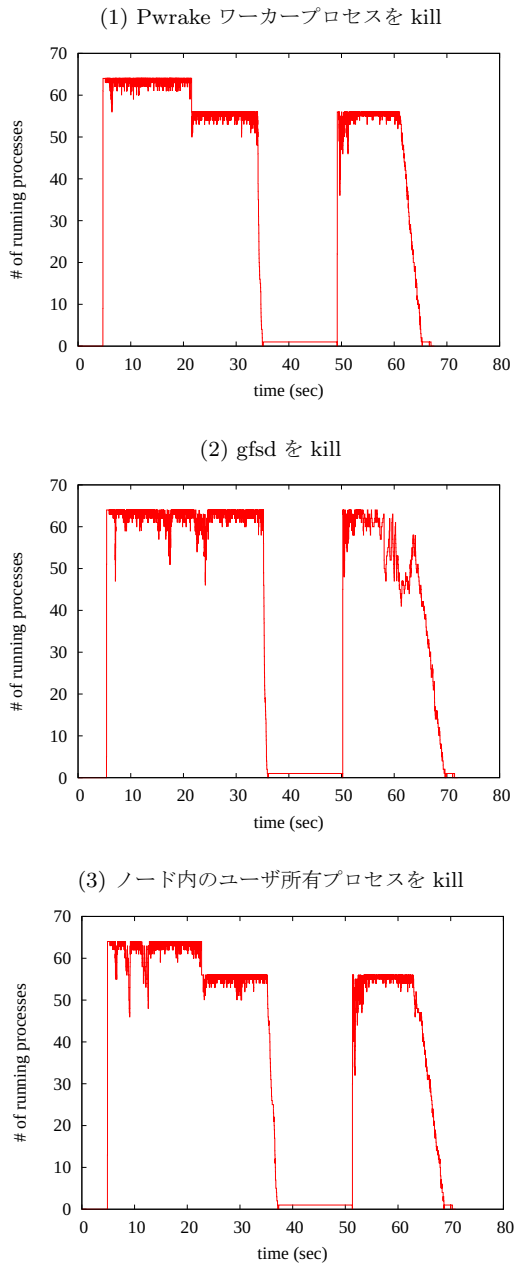

図 3 ワークフロー実行中の障害発生による稼働プロセス数の推移

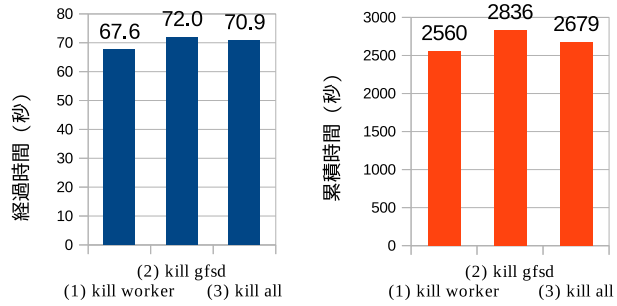

図 4 ワーカーノード障害発生時のワークフロー実行時間

たことから，自動生成された複製ファイルへのアクセスに成功していることがわかる．

7. 関連研究

Pegasus[2] では，ハードウェアレベルの障害に対しては Condor DAGMan[10] の支援によりリカバーを行う．ワークフローレベルの障害については，タスクのリトライやチェックポイントについては Pegasus が行う．Pegasus では障害からの復帰のために頻繁にチェックポイントを行うとそのオーバーヘッドが問題となり，その軽減のため，タスクを動的にクラスタリングする手法が提案されている [11]．Pwrake では，タスクの出力ファイルが分散ファイルシステムへ書き込まれた記録がチェックポイントの役割を果たすため，明示的なチェックポイントは不要である．

Swift[3], [4] では，資源管理やジョブ実行の機能を CoG Karajan[12] あるいは Falkon[13] が担う．障害対策として，rerun（再実行），restart log（チェックポイント），job replication（重複実行）の機能を持つ [4]．

Pegasus や Swift などのワークフローシステムは，資源管理の機能も持ち，ハードウェアレベルを含む様々なレベルの障害に対して対策が行われている．一方 Pwrake は，ワークフローにしたがって外部プロセスを起動するコマンドであり，資源管理機能については PBS など外部のフレームワークに依存する設計である．そのため，Pwrake 自身にはハードウェア障害について監視や復旧する機能などを持たないが，手元のマシンでも多数のノードからなるクラスタでも同じように実行できるという手軽さを備える．また，Pwrake は Gfarm ファイルシステムの耐故障性，利便性を活用する方針である．ファイルの自動複製作成によりファイル消失に対する堅牢性があるだけでなく，タスクを別ノードで再実行する場合のファイル移動が不要になるなどのメリットもある．

8. おわりに

本稿では，ワークフローシステム Pwrake における耐障害機能について，Gfarm ファイルシステムの耐障害機能を活用した設計について述べた．ワーカーノードの障害に対しては，Gfarm のファイル自動複製作成，および，Pwrake によるタスクのリトライにより，ワークフローの継続を可能にする設計とした．ノード障害とプログラムの不具合とを区別するため，同じノードで別のタスクが続けて失敗した場合はノード障害と判定し，別のノードで実行した同じタスクが続けて失敗した場合はタスクの不具合として，ワークフローを中断する設計とした．マスターノードの障害などにより Pwrake が中断した場合，ビルドツールの Make と同様に，出力ファイルの更新状況に基づいてワークフローを途中から再開できる．この仕組みを機能させるため，タスク失敗時に出力ファイルをリネーム・削除するオプションを導入した．Pwrake に対して，ハートビート，

通信切断のエラー処理などにより，障害対策について強化する実装を行った．評価実験では，Gfarm ファイル自動複製によるワークフロー実行時間への影響が 10% 以下であることを確認した．さらに，ワーカーノードに擬似的に発生させた障害に対して，ワークフローが継続して正常な結果が得られることを確認できた．

今後の課題として，障害検知とスケジューリングの連携が挙げられる．今回の評価実験では，ワーカーノードにおいて gfsd のみを落としてローカルストレージにアクセスできなくなった場合に，そのワーカーノードを使い続けるよりも脱退させた方がワークフローが早く終了するという結果となった．一方，今回の実験ではノードをつなぐネットワークは 1 Gbps であったが，より高速なネットワークを用いた場合，リモートアクセスに対して性能が維持できれば，そのワーカーノードを使い続けた方が性能が良いかもしれない．ローカルストレージが使用可能かどうかによってノード脱退をさせるべきかどうかは自明ではない．

謝辞 本研究は，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」，「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」および「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」の支援により行った．

参考文献

[1] Raicu, I., Foster, I. T. and Zhao, Y.: Many-task computing for grids and supercomputers, *Workshop on Many-Task Computing on Grids and Supercomputers, 2008 (MTAGS 2008)*, IEEE, pp. 1–11 (online), DOI: 10.1109/MTAGS.2008.4777912 (2008).

[2] Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., Katz, D. S., Others, Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C. and Katz., D. S.: Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems, *Scientific Programming Journal*, Vol. 13, No. 3, pp. 219–237 (2005).

[3] Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T. and Wilde, M.: Swift: Fast, Reliable, Loosely Coupled Parallel Computation, *2007 IEEE Congress on Services (Services 2007)*, pp. 199–206 (online), DOI: 10.1109/SERVICES.2007.63 (2007).

[4] Wilde, M., Hategan, M., Wozniak, J. M., Clifford, B., Katz, D. S. and Foster, I.: Swift: A language for distributed parallel scripting, *Parallel Computing*, Vol. 37, No. 9, pp. 633–652 (online), DOI: 10.1016/j.parco.2011.05.005 (2011).

[5] Taura, K., Matsuzaki, T., Miwa, M., Kamoshida, Y., Yokoyama, D., Dun, N., Shibata, T., Jun, C. S. and Tsujii, J.: Design and implementation of GXP make - A workflow system based on make, *Future Generation Computer Systems*, Vol. 29, No. 2, pp. 662–672 (online), DOI: 10.1016/j.future.2011.05.026 (2013).

[6] Tanaka, M. and Tatebe, O.: Pwrake: A parallel and distributed flexible workflow management tool for wide-area data intensive computing, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, New York, New York, USA, ACM Press, pp. 356–359 (online), DOI: 10.1145/1851476.1851529 (2010).

[7] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (online), DOI: 10.1007/s00354-009-0089-5 (2010).

[8] Tanaka, M. and Tatebe, O.: Workflow Scheduling to Minimize Data Movement Using Multi-constraint Graph Partitioning, *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, IEEE, pp. 65–72 (online), DOI: 10.1109/CCGrid.2012.134 (2012).

[9] Jacob, J. C., Katz, D. S., Berriman, G. B., Good, J. C., Laity, A. C., Deelman, E., Kesselman, C., Singh, G., Su, M.-H., Prince, T. A. and Williams, R.: Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking, *International Journal of Computational Science and Engineering*, Vol. 4, No. 2, pp. 73–87 (online), DOI: 10.1504/IJCSE.2009.026999 (2009).

[10] Couvares, P., Kosar, T., Roy, A., Weber, J. and Wenger, K.: Workflow Management in Condor, *Workflows for e-Science*, Springer London, London, pp. 357–375 (online), DOI: 10.1007/978-1-84628-757-2_22 (2007).

[11] Chen, W. and Deelman, E.: Fault Tolerant Clustering in Scientific Workflows, *2012 IEEE Eighth World Congress on Services*, IEEE, pp. 9–16 (online), DOI: 10.1109/SERVICES.2012.5 (2012).

[12] von Laszewski, G., Hategan, M. and Kodeboyina, D.: Java CoG Kit Workflow, *Workflows for e-Science*, Springer London, London, pp. 340–356 (online), DOI: 10.1007/978-1-84628-757-2_21 (2007).

[13] Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I. and Wilde, M.: Falkon: a Fast and Light-weight tasK executiON framework, *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, New York, NY, USA, ACM, pp. 1–12 (online), DOI: 10.1145/1362622.1362680 (2007).