Vol.2014-HPC-143 No.3
2014/3/3

ワークフローシステム Pwrake における I/O 性能を考慮したタスクスケジューリング

田中昌宏[†1†2]　　建部修見[†3†1†2]

ワークフローシステム Pwrake（プレイク）は，Ruby 版 make である Rake をベースに，タスクの並列分散実行機能を加えたシステムであり，データインテンシブなメニータスクワークフローのスケーラブルな実行を目指している．Pwrake では，スケーラブルなファイル I/O 性能を実現するため，ローカルストレージを活用する Gfarm ファイルシステムを利用する．Gfarm において高い I/O 性能を発揮する鍵は，計算ノードのローカルストレージに格納され，かつ，主記憶にキャッシュされたファイルへのアクセスである．本研究では，ローカルアクセスおよびキャッシュアクセスの機会を高め，かつ CPU 使用率を高めるための，タスクキューのスケジューリング手法について提案する．

1. はじめに

科学分野では観測装置が生成するデータ量は増加しており，膨大なデータを処理するために並列分散処理の必要性が増している．しかし，標準的な並列分散処理フレームワークである MPI は初心者には難しく，また，並列化されてないレガシーなプログラムを使用せざるを得ない状況もある．そのような場合の並列分散実行の手法として，処理を小さい逐次処理プログラムとして実装し，そのプログラムを並列に実行する方法が考えられる．そのような処理に対してバッチシステムを用いることもできるが，短時間で終わるようなタスクを大量に実行するような状況を想定していないため，オーバーヘッドによる実行効率の低さが知られている．このような多数（10^3-10^6）のタスクを並列実行するアプローチのことを Raicu らは Many Task Computing (MTC) と呼び，その取り組みについて述べている[1].

Pwrake[2][a]（プレイク）は，MTC かつデータインテンシブなワークフローの実行を目的として開発しているシステムである．Pwrake の特徴は，既存技術を活用するという点である．ワークフローの定義言語として，Ruby で記述されたビルドツール Rake を採用する．Rake は，ビルドツールの make と同様，入出力ファイルによる依存関係のあるタスクを記述できる．それに加えて Rake は，Ruby スクリプトを利用したマッピングルールなどの機能により，メニータスクワークフローの高い記述力がある．また，Pwrake ではスケーラブルなファイル I/O 性能を実現するため，Gfarm 分散ファイルシステム[3]の利用を想定した実装を行っている．Gfarm ファイルシステムは，計算ノードのローカルストレージをファイル格納場所に設定できる．そのため，ローカルストレージへのファイルアクセスを活用できれば，スケーラブルな並列 I/O 性能を実現できる．加えて，OS の機能であるファイルキャッシュを活用できれば，さらにファイル I/O の高速化が可能である．しかし，ワークフローにおいて I/O 性能を発揮するには，タスクの実行ノードと実行順序を決めるタスクスケジューリングが重要な課題となる．

本稿では，ファイルのローカリティとキャッシュを考慮しつつ，コア使用率の向上，計算と I/O のオーバーラップについても取り入れた，MTC 向けのタスクスケジューリングについて提案する．評価対象のワークフローは，単純な I/O のみのタスクからなるワークフロー，および，天文画像処理の Montage ワークフローである．これらのワークフローを計算機クラスタにて実行した結果に基づいて，提案したスケジューリング手法の有用性を示す．

2. 関連研究

グリッドでのワークフローシステムとして Pegasus[4]など，Web サービス統合ワークフローシステムとして Kepler[5]などがあるが，いずれも MTC 向けではない．MapReduce[6]はデータインテンシブな多数のタスクを並列に処理するプログラミングモデルであるが，そのモデルにあてはまるようなワークフローのみに適用可能である．

Swift[7] + Falkon[8] + Data diffusion[9]は，大規模なグリッドシステムで MTC を行うために開発されたフレームワークである．Swift は，Swiftscript という独自開発したワークフロー定義言語に基づくワークフローシステムであるが，特殊用途の言語でありユーザ層が限られる．一方，Pwrake では，Ruby ユーザに広く使われている Rake を用いる．Falkon は，高速なタスク実行のためのフレームワークであり，タスクディスパッチのピーク性能で 1500 tasks/sec を得ている．しかしタスクをまとめてディスパッチする bundling および piggy-backing という手法を必要とする．Data diffusion は，実行ノードにステージされたデータの管理とタスクスケジューリングを行う機構であり，ローカリティの考慮を行うことができる．Pwrake では，Gfarm ファイルシステムの機能を活用し，データ移動をリモートアクセスで置き換える．

GXP make[10]は，ワークフロー定義言語として記述力が高く，広く使われている GNU make を用いて，GXP による

†1 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
†2 独立行政法人科学技術振興機構 CREST
JST CREST
†3 筑波大学システム情報系
Faculty of Engineering, Information Systems, University of Tsukuba

[a] https://github.com/masa16/pwrake

分散並列実行を行うシステムである．動作原理は，GNU make が発行するコマンドラインを mksh によってトラップし，GXP のタスクディスパッチシステムによって分散実行を行う．そのため，タスクの実行順序が GNU make の実装に依存する，入出力ファイル名を容易に得られない，という制約がある．後者については，I/O 履歴を元にファイルアクセスを予測する方法が提案されている[11]．Pwrake では，Rake::Task クラスのインスタンスから得られる入出力ファイル名に基づき，タスクの実行ノードおよび実行順序を制御するスケジューリングの実装を行う．

3. ワークフローシステム Pwrake

Pwrake は，以前の発表[2]より後の開発により，性能，機能，Rake との互換性が向上している．ここで Pwrake の概要について簡単に述べる．

3.1 Rake の概要

Rake は，UNIX make と似た機能を持つビルドツールであり，Ruby 言語で記述され，Ruby のパッケージに標準添付されている．Rake では，1 つのタスクを Rake::Task クラス（以下 Task クラス）のインスタンスで表し，タスク名・事前タスク名・アクションなどの情報を保持している．事前タスク（prerequisite task）とは，あるタスクの実行前に終了していなければならないタスクをいう．Rake のタスクは，0 個以上の事前タスクを持つ．Rake では事前タスクの指定によってタスクの依存関係，すなわち，DAG（Directed Acyclic Graph）を構成する．

Task クラスのサブクラスとして，ファイル生成タスク（Rake::FileTask，以下 FileTask）がある．FileTask では，タスク名が出力ファイル名とみなされ，事前タスクとして入力ファイルを指定できる．FileTask の場合は，出力ファイルが存在しないか，出力ファイルが入力ファイルの方が新しいときに実行される．この仕組みにより，ワークフローが中断した場合，再びワークフローを実行すれば未完了のタスクから実行することができる．

3.2 Pwrake の概要

Pwrake は Rake に対する機能拡張であり，Rake のタスク定義の仕様を踏襲する．Task クラスの実装など Rake の多くの実装を Pwrake でも利用する．Pwrake で拡張した機能は，おもにタスクの並列実行機能と，遠隔実行機能である．Pwrake の概要を図 1 に示す．Pwrake は起動すると，実行ノードを記したファイルを読み，コア数と同じ数のワーカースレッドを作成する．次に Rakefile を読み，事前タスク名とアクション部などの情報を持った Task クラスのインスタンスを生成する．そしてターゲットタスクから事前タスクを順に辿り，事前タスクを持たないタスクが見つかれば，TaskQueue クラスとして実装したタスクキューに投入する（enq，エンキュー）．それぞれのワーカースレッドでは起動すると SSH で担当のワーカーノードに接続する．その後，タスクキューからタスクを取り出し（deq，デキュー），スレッド内でタスクのアクション部を実行する．その中で sh メソッドが呼ばれると，引数の文字列をコマンドラインとして SSH を通じてワーカーノードで実行する．アクション部の実行が終わると，次に実行可能なタスクを探してタスクキューに投入した後，再びタスクキューからタスクを取り出して実行する，というサイクルを繰り返す．

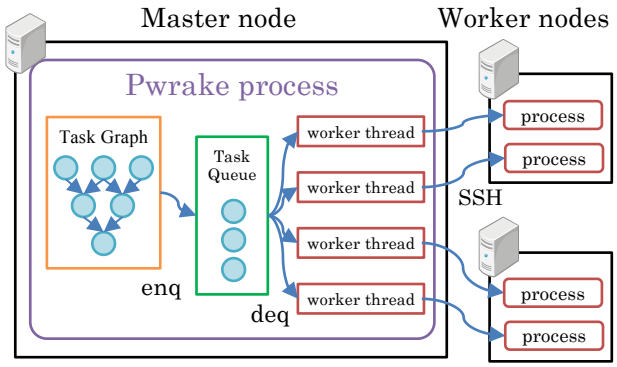

図 1　Pwrake システムの概要

4. Pwrake タスクスケジューリングの設計

タスクスケジューリングに関する研究では，多くの場合，makespan（ワークフローの開始から終了までのスケジュール時間）の最小化を目的としている．代表的なものに HEFT[12]がある．HEFT のようなスケジューリングアルゴリズムは，事前に個々のタスクの実行時間が与えられる．一方，MTC の場合，タスク数は 10^3-10^6 であり，個々のタスクの実行時間の事前予測は難しい．そこで MTC 向けに設計している Pwrake では，個々のタスクの実行時間を事前に予測しない．Pwrake におけるタスクスケジューリングは，ワーカーがタスクキューからタスクを取り出す時にどのタスクを選択するかを決める動的なスケジューリングとする．

データインテンシブなワークフローを対象とする Pwrake では，ファイル I/O の性能が重要な課題である．

表 1　Gfarm ファイルの I/O 性能

			転送速度 (MiB/s)
Read	Local	disk	70
		cache	592
	Remote	disk	39
		cache	71
Write	Local	disk	59

表 1 に，Gfarm ファイルシステムに格納したファイルの読み込み・書き込み速度を，InTrigger の東北クラスタ（環境の詳細は 5.1 節で述べる）で測定した結果を示す．この環

境では，ローカルストレージに格納され，かつ，キャッシュされたファイルを読む場合が最も速い．そこでローカルかつキャッシュへのアクセスを最大化することを目的としたタスクスケジューリングを設計する．

4.1 タスク実行ノードの決定

ローカリティを考慮したスケジューリングとは，入力ファイルへのアクセス全体のうち，ローカルストレージへのアクセスの割合を高めるように，タスクの実行ノードを決めることである．ローカリティを考慮したタスクキューは，図 2 のような入れ子構造とする．タスクキューは，実行ノードに対応するノードキューを持つ．(各ノードキューでは，4.2 節で述べるように，タスクの実行順序を担う．)

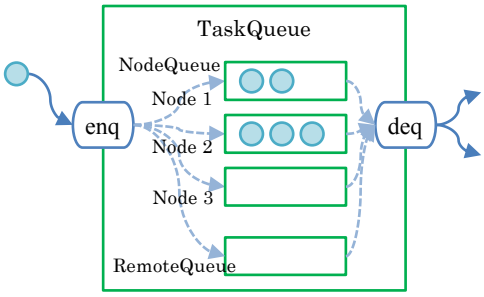

図 2　タスクキューの構造

タスクキューにおけるローカリティスケジューリングの大まかな流れは次の通りである．

(1) タスクを enq するとき，「候補ノード」を決定する．
(2) タスクを候補ノードのノードキューに入れる．
(3) ワーカースレッドが deq するとき，該当するノードキューからタスクを取り出す．

それぞれのステップについて以下で説明する．

(1) 候補ノードの決定

Gfarm ファイルシステムでは，ファイル単位で格納ノードが決まる．タスクが複数の入力ファイルを持つ場合，または入力ファイルが複数のノードに複製されている場合に，1 タスクにつき入力ファイルの格納ノードが複数となる可能性がある．そのような場合，サイズが小さいファイルを持つノードよりも，大きいファイル持つノードにタスクを割り当てる方が良い．ここでは，タスクを実行する「候補ノード」を選ぶ方法について述べる．

図 3 に候補ノード決定の例を示す．この例では，タスク t は入力ファイルを 3 つ取り，ファイル C は Gfarm の機能により 3 つのノードに複製されているとする．ここで，もし，入力ファイルの存在するノードすべてを候補ノードとすると，小さいサイズのファイル C を持つノード 3 で実行される可能性があり，その場合リモートアクセスの割合が多くなる．一方，最大のファイルサイズを持つノード 1 のみを候補ノードとすると，2 番目のファイルサイズを持つノード 2 が空いていても割り当てられない．そこで，候補ノードの決定方法としては，ノードごとにファイルサイズを合計し，その最大値の半分以上を持つノードを候補ノードとする．図の例では，ノード 1 と 2 が候補ノードとなる．

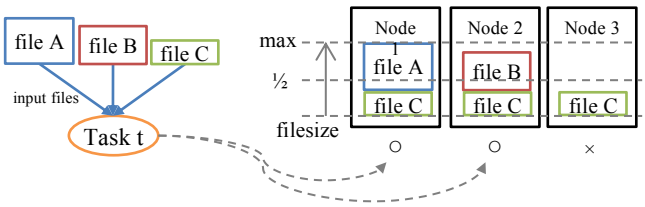

図 3　候補ノードの決定の例

(2) ノードキューへのエンキュー

タスクキューにタスクを enq するとき，(1)で選んだ候補ノードに対応するノードキューにタスクを投入する．また，候補ノードが複数の場合は，対応するすべてのノードキューに投入する．

タスクキューに入れようとするタスクが入力ファイルを持たない場合，または，入力ファイルが計算ノード以外のノードに格納されている場合は，候補ノードがないため，ノードキューに入れることができない．そこで，タスクキューはノードキューの他にリモートキューを持ち，候補ノードがないときはリモートキューに投入する．

(3) ノードキューからのデキュー

ワーカースレッドは，アイドルになると，ノード名を引数として TaskQueue の deq メソッドを呼び，次に実行するタスクを取り出す．該当するノードキューにタスクが存在すればそのタスクを返す．この時同じタスクが複数のノードキューに入っていればそれらをすべて削除する．

一方，該当するノードキューにタスクが存在しない場合もある．これは，入力ファイルが一部のノードに偏ったり，タスクの実行時間がばらついたりした結果として生じる．この場合の手順は，まず，リモートキューにタスクが存在すれば，そこから deq して返す．さらにリモートキューも空のときは，(a)アイドル状況のまま待つ，または，(b)別ノードに割り当てられたタスクを「スチール」する．ノード間の転送速度が十分速い場合は，スチールによって動的な負荷分散が期待できる．どちらが良いかは計算機環境に依存するため，オプションで選択できるようにする．

4.2 タスク実行順序の決定

前節で述べたノードキューは，タスクの実行順序を決める役割を担う．実行順序は，以下で述べるファイルのキャッシュと末尾タスク問題を考慮して決める．

4.2.1 キャッシュの考慮

キャッシュを考慮したスケジューリングとは，タスクの入力ファイルがキャッシュに乗る確率を最大化することである．しかし，個々のファイルがキャッシュされているかどうか正確にはわからない．一般的な傾向として，キャッ

シュは古いデータから消去されるから，新しいファイルほどキャッシュされている確率は高いといえる．したがって，キャッシュされている確率が高い（＝新しい）入力ファイルを持つタスクを優先するというスケジューリングを行う．これを実現するには，入力ファイルのアクセス時刻による優先度つきキューを用いればよい．しかし，優先度つきキューは探索コストを必要とする．一方，入力ファイルを生成したタスクが終了したときにタスクがキューに入れられることを考えれば，後からキューに入れられたタスクほど入力ファイルの生成時刻が遅い．以上のことから，キャッシュを考慮したスケジューリングとして，単純に後から来たタスクを先に実行する LIFO(Last-In-First-Out)を適用する．

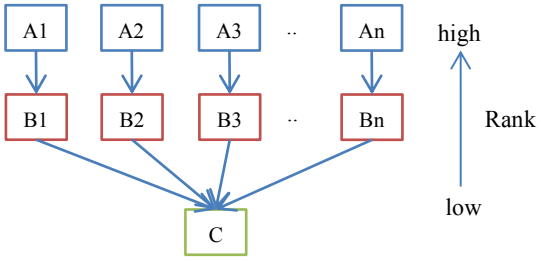

図 4　スケジューリング対象のワークフロー

例として，図 4 に示すワークフローのスケジューリングを考える．この図で，グラフの頂点はタスクを表し，グラフの辺はファイルを通した入出力を表す．このワークフローは，n 個のファイルを入力とし，それぞれをタスク A が処理し，その出力ファイルを入力とするタスク B で処理するものとする．簡単のためタスクの相対的な処理時間は全て同じであると仮定する．このワークフローを 2 コア 1 ノードで実行する場合について，LIFO でスケジューリングした結果を図 5 の(1)に示す．

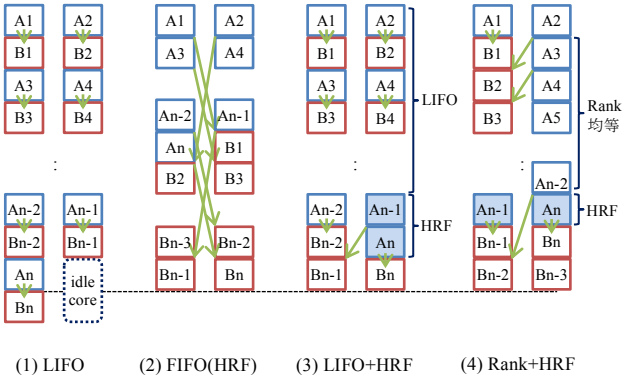

図 5　タスクスケジューリング手法

図 5 は，上から下へ時間の流れを表し，最終タスク C の前までのタスクの実行状況を示している．LIFO の場合，タスク Ai の直後に必ずタスク Bi が処理されるから，キャッシュに乗る確率を最大にしているといえる．一方，FIFO (First-In-First-Out)でスケジューリング場合（図 5 の(2)）は，Ai と Bi の間隔は平均的にタスク処理時間×(n/2)となり，キャッシュに残る確率が低くなる．キャッシュに乗らなければ読み込み性能が低くなり，個々のタスクの実行時間が図で示したよりも増加する．（矢印が隣の列に向かっていることは，同一ノード内であるから問題ない．）

4.2.1 末尾タスク問題

末尾タスク問題（Trailing Task Problem[13]）とは，MTC ワークフローにおける並列タスクの待ち合わせ，あるいは終了直前などの状況において，実行可能なタスクがコア数より少なくなったとき，アイドルコアが発生し，コア使用率が低くなるという問題である．図 5 (1) の LIFO では，入力ファイル数 n が奇数のとき，最後のタスク An と Bn を実行する間，別の 1 コアがアイドルになる．一方 FIFO では，先にすべてのタスク A を実行した後でタスク B を実行する．つまり，アイドルコアの発生する時間は，最悪で LIFO は An+Bn，FIFO は Bn のタスク実行時間である．このように，末尾タスク問題において LIFO は不利である．

末尾タスク問題を抑えるために効果的な方法は，ターゲットタスクから遠いタスク A を早い段階で実行することである．HEFT などの事前にスケジューリングアルゴリズムには，計算・通信コストに基づいてターゲットタスクからの遠さ（ランク）を決定し，ランクが高い順にタスクを CPU に割り付けるものがある．このようにランク順にタスクを実行するポリシーのことを，ここでは Highest Rank First (HRF)と呼ぶ．HRF は末尾タスク問題を抑えるための有効な手法であるといえる．

ここで問題となるのは，HRF（FIFO と同順序）と LIFO とでは，順序が逆であり，両立しないことである．並列タスク数 n が多いときは Ai と Bi の間隔が大きくなるため LIFO が有効であり，少ないときは末尾タスク問題が顕著になるため FIFO が有効である．これら 2 つをオプションで切り替える方法も考えらえるが，そうするとどちらが効率的かをユーザが決める必要がある．オプションで切り替えずに，どのようなケースにも対応できるような手法について，4.2.3 節で述べる．

4.2.2 ランクの定義

提案手法について述べる前に，まず本手法におけるランクの定義を行う．本研究では MTC スケジューリングを目的とし，タスクの計算コストは事前にわからないという前提であるため，計算コストに基づくランクは得られない．そこで本手法におけるランクを次のように定義する．まずワークフローの最終ターゲットであるタスクをランク 0 とする．そして，ランク i の事前タスクをランク i+1 とする．ランクの異なる複数のタスクの事前タスクとなっている場合は，その中で最大のランクに 1 を加えてそのタスクのランクとする．ランクは整数である．ワークフローの実行前に全てのタスクについてランクを決めておく．

4.2.3 提案手法

前述のようなキャッシュとコア使用率の問題を同時に解決するための，タスク実行順序を決定する手法として，次の 2 つの手法を提案する．

(1) **LIFO + HRF**

1 つ目の手法は，LIFO と HRF を組み合わせた手法である．キューに入っているタスクのうち最高ランクが r であるとき，そのランク r のタスク数を Nr とし，キューが担当するノードのコア数を Nc とすると，

1. Nr＞Nc のとき，LIFO の順序でタスクを選択
2. Nr≦Nc のとき，HRF の順序でタスクを選択

図 4 に示した DAG について，これらの手法でスケジューリングした結果を図 5 の(3)に示す．タスク Ai と Bi (i=1..n-2) のスケジューリングの時は，キューにタスク A がコア数の 2 個より多く残るから，LIFO の順序となり，後からキューに入った B が先に実行される．このとき Ai の直後に Bi が実行されるから，キャッシュに乗る確率が最大である．一方，キューに An-1 と An のみ残る場合，HRF の順序により，Bn-1 よりもランクが高い An が先に実行される．これによって Bn-1 と Bn が同時に実行できるようになる．

(2) **Rank+HRF**

2 つ目の手法では，計算と I/O のオーバーラップを考慮する．図 4 のようなパターンのワークフローにおいて，タスク A, B のうち一方が計算インテンシブなタスクであり，もう一方が I/O インテンシブなタスクであるという特殊なケースでは，タスク A とタスク B がオーバーラップすることにより，実行時間を短縮できる可能性がある．A と B がオーバーラップするスケジューリングの例を図 5 の(4)に示す．キャッシュ利用の指標となる Ai と Bi のタスク間隔が，LIFO ではゼロであるのに対し，オーバーラップスケジューリングでは 1 と少し不利であるが，FIFO での n/2 より有利である．

オーバーラップを実現するためには，各ランクで同時に走るタスク数が等しくなるように，タスクを選ぶ必要がある．ここで注意すべきは，各ランク同じ頻度でタスクを起動すると，コア占有率がタスクの実行時間に比例するため，実行中のタスク数が均等にならないという点である．そこで，オーバーラップを考慮したスケジューリングのアルゴリズムとして，

1. エンキューの時，タスクをランクごとに分類．
2. デキューの時，
   - **タスクの平均実行時間の逆数**を重みとして，乱数でランクを選択．
   - 選択されたランクから LIFO でタスクを取得．

という方法で実現する．ここでも末尾タスク問題が発生するため，最高ランクのタスク数がコア数以下のときは HRF に切り替える．ワークフロー実行前にはタスクの実行時間が与えられていないため，最初は重みを各ランク均等にしておく．終了したタスクがある場合は，タスク実行時間の測定値から各ランクの平均値を計算して用いる．

5. 性能評価

5.1 評価環境

表 2 測定環境

クラスタ	InTrigger 東北大学拠点
CPU	Intel Xeon E5410 2.33GHz
主記憶容量	32GiB
コア数/ノード	8
計算ノード数（最大）	12
ネットワーク	1Gb Ethernet
OS	Debian 5.0.4
Gfarm	ver. 2.5.8.4
Ruby	ver. 2.1.0
Pwrake	ver. 0.9.9

評価環境について表 2 に示す．用いた計算機クラスタは，InTrigger プラットフォーム[14]の東北大学拠点である．用いたノードのうち，Gfarm メタデータサーバ兼 Pwrake マスタノードとして 1 ノード，計算ノード兼 Gfarm ファイルシステムノードとして最大 12 ノードを使用した．各ノードのローカル HDD の 1 つに Gfarm ファイルを格納するように設定した．Gfarm の書き込みをすべてローカルノードにするため，gfarm2.conf のオプションにて schedule_idle_load_thresh 100 と設定した．

5.2 I/O のみを行うワークフロー

タスクスケジューリングによるローカリティとキャッシュの効果を明らかにするため，ファイルのコピーのみを行うワークフローについて評価を行う．ワークフローのタスクとして C 言語で作成した copyfile というプログラム[b]を用いた．copyfile は，入力ファイルをすべて主記憶に読み込んだ後，そのデータを出力ファイルに書き込むプログラムである．（一方，科学計算では，主記憶に読み込んだ後に計算を行う．）ワークフローの DAG は図 4 と同じであり，入力ファイル 1 つにつき 2 回の copyfile 処理を行う．入力ファイルとして，あらかじめ 3 GiB のファイルを 100 個作成し，10 ノードに分散して格納した．入力ファイル 1 つにつき 2 回コピーするから，読み込みと書き込みの合計サイズはそれぞれ 600 GiB である．

使用した計算ノードは InTrigger 東北拠点のうち 10 ノードである．I/O のみの処理のため，1 ノードあたり 1 コアのみを用いた．したがって 10 プロセスの並列処理である．計算ノードの主記憶は 32GiB であるから，3 GiB のファイルを 1 回読んだ直後はキャッシュに乗っているが，100 個の入力ファイルをすべて読んだ後では最初のファイルはキャッシュから追い出される．

[b] https://gist.github.com/masa16/5956881

評価するスケジューリング手法は，実行ノード，実行順序それぞれ 2 種類である．実行ノードについては，ローカリティの考慮あり・なしの 2 種類について比較する．ローカリティを考慮しないスケジューリングとは，ノード毎にノードキューを区別しない単純なスケジューリングである．実行順序については，FIFO と LIFO の 2 種類について比較する．今回は 1 ノード 1 コアの測定であるから HRF の有無は性能に影響しないため，HRF の評価は行わない．

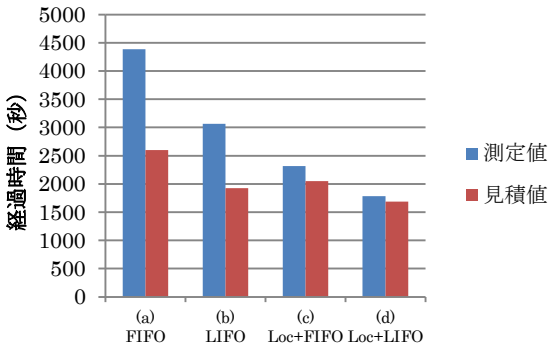

図 6　copyfile ワークフローの経過時間

ワークフロー実行時間を 3 回測定し，その平均値を図 6 の青いバー（各項目の左側）で示す．ラベル先頭の Loc の有無によって，スケジューリングにおけるローカリティ考慮の有無を表す．Loc+LIFO のケース(d)の性能は，(c) Loc+FIFO から 1.30 倍，(b) Loc なし LIFO から 1.72 倍，(a) Loc なし FIFO から 2.46 倍の高速化を達成し，ローカリティかつキャッシュを考慮したスケジューリングが性能向上に有効であることを示している．

測定結果と比較するため，ワークフロー実行時間の見積もりを行う．copyfile の実行時間は $T = I/R + O/W$ として計算できる．ここで，I, O はそれぞれ入，出力ファイルサイズ，R, W はそれぞれリード，ライト速度である．R と W の値は表 1 に示した Gfarm の I/O 性能を用いる．リード速度として，LIFO 順序で起動したタスク B の場合のみキャッシュの速度，その他はディスクの速度を用いる．ローカルとリモートのアクセスの割合は，表 3 に示す測定時の結果を用いる．

こうして見積もったワークフローの実行時間を図 6 の赤いバーで示す．見積値に対する測定値の比は，ローカリティありの場合は約 1.1 と 1 に近く，見積値が実測値をだいたい再現できている．一方，ローカリティなしの場合は約 1.6 であり，ローカリティなしの場合よりも大きく増加している．この原因として，データが同時にネットワークのハブを通るため輻輳が起きていることが考えられる．

表 3　測定時のファイルアクセスの割合（%）

	Task A		Task B	
	Local	Remote	Local	Remote
(a) FIFO	10.7	89.3	9.0	91.0
(b) LIFO	10.7	89.3	**99.3**	0.7
(c) Loc+FIFO	**100.0**	0.0	**92.0**	8.0
(d) Loc+LIFO	**96.3**	3.7	**99.7**	0.3

ここで表 3 の内容について補足する．赤い太字はローカルアクセスが 100%に近いことを示す．LIFO を適用すると，ローカリティを考慮していないにもかかわらずタスク B の読み込みがほぼローカルアクセスとなった．その理由は以下の通りである．図 4 のワークフローにおいて，タスク Ai が終了した時の手順は，(1)Ai の次のタスク Bi をタスクキューに入れる，(2)Ai の実行を終えたワーカーが次のタスクをタスクキューから取得，となる．このとき，LIFO であれば，取得するタスクは直前に入れた Bi である．このように結果的に Ai と Bi を実行するワーカーノードが同じとなり，タスク Bi に関しては偶然ローカルアクセスとなった．

5.3 Montage ワークフロー

データインテンシブな科学計算ワークフローとして，天文画像の合成（モザイキング）を行う Montage[15] のワークフローに対して評価を行う．Montage は，座標をずらして撮影された複数の画像を重ね合わせて，1 枚の広い領域の画像を作成するためのソフトウェアである．写真のパノラマ合成と同様に，異なる天球座標への投影する際の画像の歪み補正，および，背景の明るさの変換を行う．Montage は処理ごとにプログラムが分かれており，それらを実行するワークフローを Rakefile として記述した[c]．Montage ワークフローの DAG の例を図 1 に示す．実際の処理では，入力ファイル数に応じてタスク数はさらに多くなる．Rakefile の記述で，mDiff と mFitplane のプログラムは 1 つのタスクの中にまとめた．

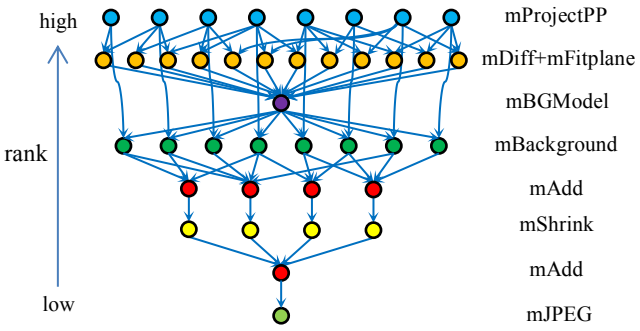

図 7　Montage ワークフローの DAG

[c] https://github.com/masa16/pwrake-demo/tree/master/montage

入力画像ファイルとして，SDSS DR7[16]の画像を用いた．測定したワークフローの概要を表 4 に示す．

表 4　Montage ワークフローの概要

入力ファイル	SDSS DR7
入力ファイル数	421
入力ファイルサイズ（1 個）	2.52 MB
入力ファイルサイズ（合計）	1061 MB
中間画像ファイル数	4720
中間画像ファイルサイズ（合計）	63.5 GB
タスク数	2707

5.3.1 スケジューリング手法による性能比較

まずタスクスケジューリングが Montage ワークフローの性能にどう影響するかについて調べる．この測定では InTrigger 東北拠点の 12 ノード 96 コアを用いた．測定は 5 回行い，最も性能が良いデータを採用した．スケジューリング手法として，実行ノード選択についてはローカリティ考慮なし・ありの 2 種類，実行順序については FIFO, LIFO, LIFO+HRF, Rank+HRF の 4 種類について測定した．ここに示すワークフロー実行時間には，逐次処理のタスク（mBGModel と最後の mAdd, mJPEG）を除いている．測定結果を表 5 に示す．

表 5　Montage ワークフローの測定結果

	Locality	FIFO	LIFO	LIFO+HRF	Rank+HRF
ワークフローの経過時間（秒）	No	195.5	180.9	175.4	173.6
	Yes	136.2	141.5	133.8	131.8
タスクの累積実行時間（秒）	No	17686	15225	15667	15656
	Yes	11607	11523	11456	11230
コア使用率	No	94%	88%	93%	94%
	Yes	89%	85%	89%	89%
入力ファイルのローカルアクセス率	No	9%	23%	19%	16%
	Yes	48%	47%	48%	47%

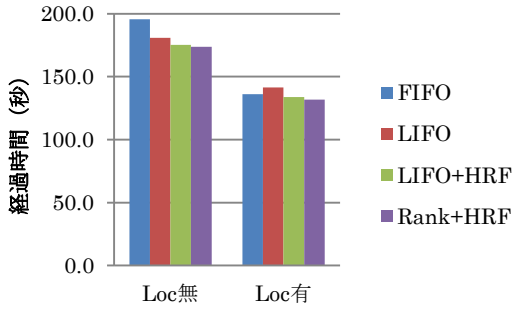

図 8　ワークフロー実行の経過時間

この測定結果に基づくワークフロー実行の経過時間のグラフを図 8 に示す．「Loc 無」で示したローカリティを考慮しない場合に比べて，「Loc 有」のローカリティを考慮したスケジューリングでは，約 28-44%の速度向上が得られた．この速度向上は，図 9 に示した入力ファイルのローカルアクセス率が，ローカリティを考慮しない場合は約 9%-23%であるのに対し，考慮した場合では約 47-48%と増加したためであり，ローカリティを考慮したスケジューリングによる性能向上を確認できた．

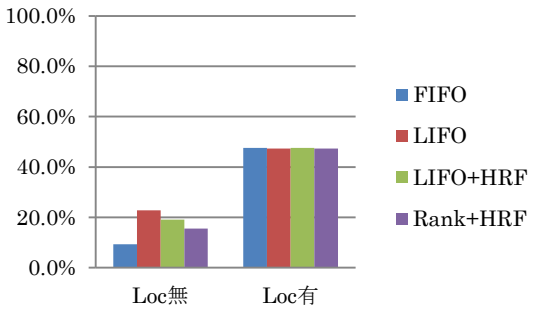

図 9　ローカルアクセス率

一方，図 8 のローカリティを考慮したスケジューリングの間では，Rank+HRF が最も良い性能が得られた．Rank+HRF への性能向上は，FIFO, LIFO, LIFO+HRF からはそれぞれ約 3%, 7%, 2%である．この測定で FIFO の性能が LIFO より良い理由は，1 ノードあたりで処理するファイルサイズが小さいため，FIFO であってもすべてのファイルがキャッシュに収まることが考えられる．例えば，mProjectPP が出力するファイルサイズの合計は約 20GB であるが，これが 12 ノードに分散されるため，1 ノードあたり約 1.7GB であり，32GiB の主記憶より十分小さい．

LIFO の性能が劣る理由は，末尾タスク問題に起因する，コア使用率の低下である．ここでコア使用率は $t_{\mathrm{accum}}/(t_{\mathrm{elap}}n_{\mathrm{cores}})$ と定義する．t_{accum}は個々のタスクの累積実行時間，t_{elap}はワークフローの経過時間，n_{cores}はコア数（並列数）である．コア使用率のプロットを図 10 に示す．ローカリティを考慮したスケジューリングの中では，コア使用率が LIFO のみ約 85%と低く，他は約 89%である．

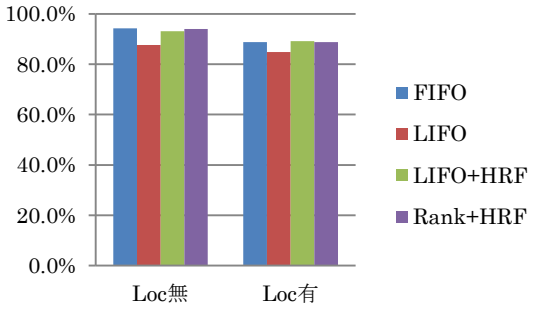

図 10　コア使用率

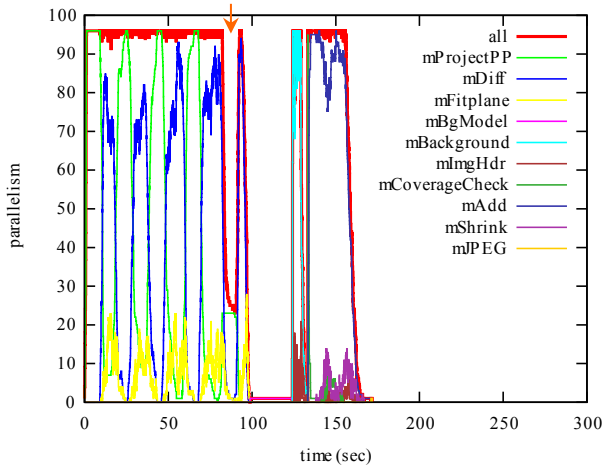

図 11　実行中のプロセス数の推移 (LIFO)

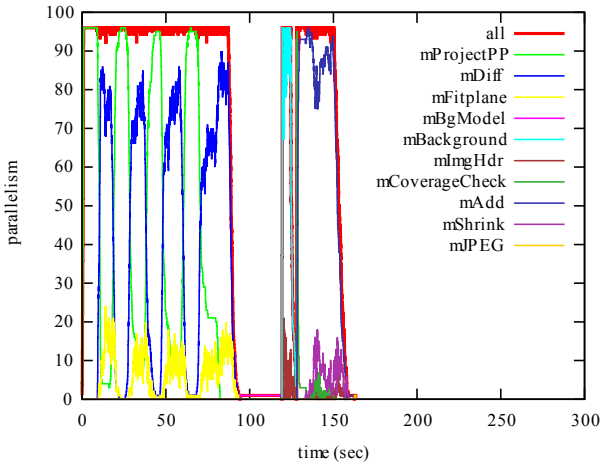

図 13　実行中のプロセス数の推移 (LIFO+HRF)

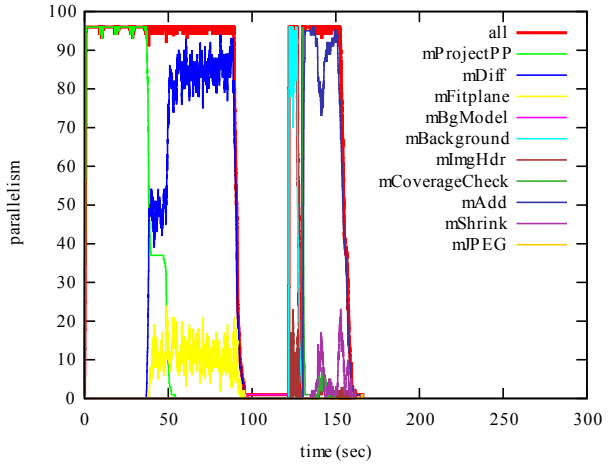

図 12　実行中のプロセス数の推移 (FIFO)

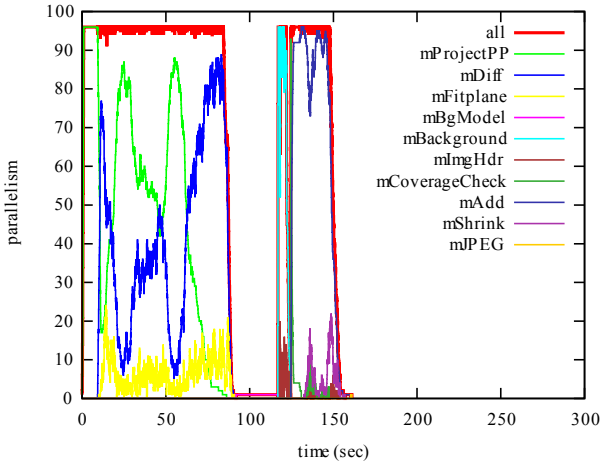

図 14　実行中のプロセス数の推移 (Rank+HRF)

実行順序がコア使用率に影響する様子を詳しく見るため，ワークフロー実行中のプロセス数（並列度）の推移を図 11 から図 14 までに示す．これらは全てローカリティを考慮したケースである．横軸がワークフロー開始からの時間経過，縦軸が実行中のプロセス数であり，赤い太線は全体のプロセス数，他の線はプログラム毎のプロセス数を示す．

ここで，最初のタスクである mProjectPP とその次のタスク mDiff+mFitplane に着目する．（mDiff と mFitplane は，1 つのタスクとして記述したが，プロセスとしては別々にカウントされている．以降は mDiff で代表して説明する．）図 11 の LIFO の場合，mProjectPP と mDiff が交代で実行されていることがわかる．ところが，矢印で示した 90 秒付近で並列度が下がっている．これは，キューに残った mProjectPP のタスク数がコア数よりも少ないためである．図 12 の FIFO のケースでは，先にすべての mProjectPP を実行し，その後で mDiff を実行するため，LIFO のような並列度が下がる現象は発生しない．しかし FIFO には，問題サイズが大きい場合はキャッシュから追い出されるという問題がある．両方の問題を解決するために考案した LIFO+HRF では（図 13），並列度が下がる部分が解消されている．これは，mProjectPP の残りの数が少なくなったとき，mProjectPP を優先して実行するためである．

Rank+HRF（図 14）では，mProjectPP と mDiff の比率が波打ってはいるものの LIFO のときのような交代現象は起きていない．これは，4.2.3 節の(2)で述べた，計算と I/O のオーバーラップを狙ったものである．mProjectPP は異なる天球座標へ投影するタスクであり計算量が比較的多い．一方 mDiff はピクセル値の差を計算するタスクであり計算量が比較的少ない．そこで mProjectPP と mDiff をオーバーラップさせることで，タスクの実行時間を短縮する効果が期待できる．その効果を見るため，タスクの累積実行時間のプロットを図 15 に示す．

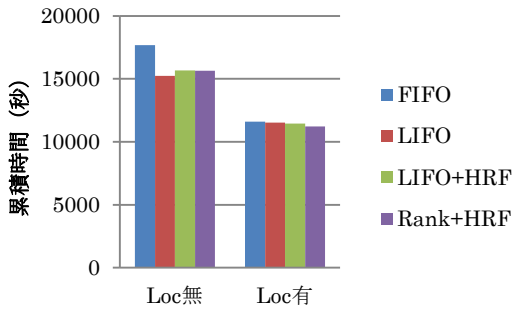

図 15　タスクの累積実行時間

図が示すように，Loc+LIFO+HRF に比べて Loc+Rank+HRF は累積実行時間を約 2%の削減できた．このようにスケジューリングの工夫により計算と I/O がオーバーラップすれば，性能向上につながることが確認できた．

5.3.2 スケーラビリティ

Montage ワークフローのストロングスケーリングによるスケーラビリティを調査するため, 前節のワークフローを, 問題サイズを変えずに, ノード数を 1 から 12 まで変えて測定した．1 ノード 8 コアであるから使用コア数は 8-96 である. 測定回数は 1-3 ノードでは 1 回, 4-11 ノードでは 3 回, 12 ノードでは 5 回とし，そのうち最も性能が良い結果を採用した．こうして得た測定結果を図 16 と図 17 に示す．ここで 5.3.1 節と同様，逐次タスクは除外している．これらの図で, 破線は縦軸と横軸が反比例することを示しており，この線に沿えばスケールするという目安である．

4 ノード 32 コア以上の測定結果は，反比例の破線に沿っており，スケールしていることがわかる．特にローカリティ考慮を考慮したスケジューリングは，しない場合よりもスケーラビリティが向上している．この結果は，大規模環境でワークフローを実行する場合のローカリティの重要性を示している．また，最も性能が良いスケジューリングは Loc+Rank+HRF という結果となった．Montage のワークフローに対しては，計算と I/O がオーバーラップするスケジューリングが有効であるといえる．

図 16 で，ノード数 1-3，コア数 8-24 では，経過時間が破線よりも全体的に増えており，中でもキャッシュ活用に不利な FIFO スケジューリングの経過時間が他のスケジューリングより長い．この原因として，少ないノード数では 1 ノードあたりで処理するデータ量が多くなり，多くのデータがアクセス前にキャッシュから追い出されていることが考えられる．この結果は，大規模データ処理におけるキャッシュを考慮したスケジューリングの必要性を示している．

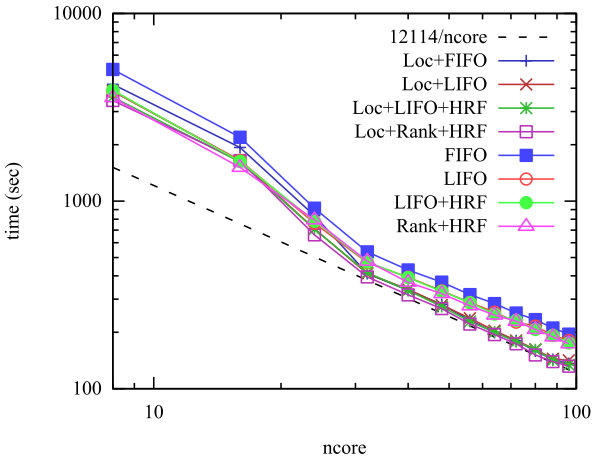

図 16　コア数対ワークフロー経過時間
（両対数グラフで表示）

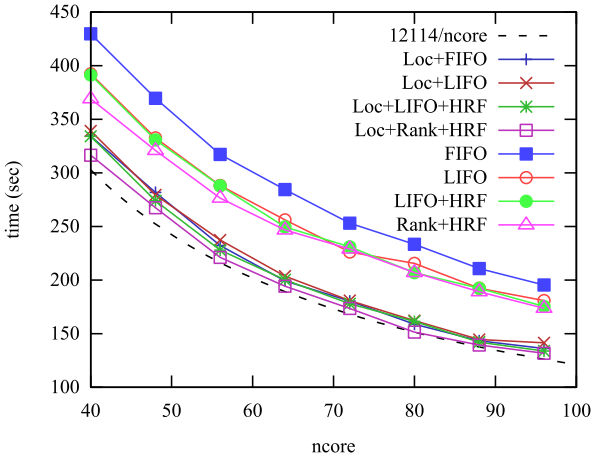

図 17　コア数対ワークフロー経過時間
（40 コア以上の部分を拡大，線形軸で表示）

6. まとめと今後の課題

Many Task Computing (MTC) およびデータインテンシブなワークフローのスケーラブルな実行を目的として開発したワークフローシステム Pwrake において, ローカリティおよびキャッシュを考慮したタスクスケジューリング手法について提案した．ローカリティに関しては，入力ファイルサイズを考慮に入れて複数のノードを候補とする手法を提案した．ファイルのキャッシュに関しては，実行順序が LIFO の場合の末尾タスク問題を Highest Rank First (HRF)で解決する手法，および，計算と I/O のオーバーラップを目的として各ランクのタスクを均等に実行する手法を提案した．I/O のみのワークフローによる性能評価では，ローカリティとキャッシュを考慮したスケジューリングにより，I/O 性能が向上し，ワークフロー実行時間が短縮されることを確認した．Montage ワークフローを用いた性能評価で

は, 末尾タスク問題に起因するコア使用率減少が HRF により改善すること，および，計算と I/O のオーバーラップを目的とするスケジューリングによって性能が向上することを確認した．

今後の課題としては，大規模な環境での評価，グラフ分割を用いた事前スケジューリング[17]の取り入れなどが挙げられる．

謝辞 本研究は，JST CREST「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] I. Raicu, I. T. Foster, and Y. Zhao, “Many-task computing for grids and supercomputers,” in *Workshop on Many-Task Computing on Grids and Supercomputers, 2008 (MTAGS 2008)*, 2008, pp. 1–11.

[2] M. Tanaka and O. Tatebe, “Pwrake: A parallel and distributed flexible workflow management tool for wide-area data intensive computing,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC ’10)*, 2010, pp. 356–359.

[3] O. Tatebe, K. Hiraga, and N. Soda, “Gfarm Grid File System,” *New Gener. Comput.*, vol. 28, no. 3, pp. 257–275, Aug. 2010.

[4] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Sci. Program.*, vol. 13, no. 3, pp. 219–237, 2005.

[5] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the Kepler system,” *Concurr. Comput. Pract. Exp.*, vol. 18, no. 10, pp. 1039–1065, Aug. 2006.

[6] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Comput.*, vol. 37, no. 9, pp. 633–652, Sep. 2011.

[8] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, “Falkon: a Fast and Light-weight tasK executiON framework,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 43:1–43:12.

[9] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay, “Accelerating large-scale data exploration through data diffusion,” in *Proceedings of the 2008 international workshop on Data-aware distributed computing - DADC ’08*, 2008, pp. 9–18.

[10] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, and J. Tsujii, “Design and implementation of GXP make — A workflow system based on make,” *Futur. Gener. Comput. Syst.*, vol. 29, no. 2, pp. 662–672, Feb. 2013.

[11] 堀内 美希, 田浦 健次朗, “広域分散ファイルシステムのための適応的な先読み手法,” *情報処理学会研究報告 2012-HPC-135*, no. 26, 2012.

[12] H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[13] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, “Scheduling many-task workloads on supercomputers: Dealing with trailing tasks,” in *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*, 2010, pp. 1–10.

[14] 斎藤秀雄, 鴨志田良和, 澤井省吾, 弘中健, 高橋慧, 関谷岳史, 頓楠, 柴田剛志, 横山大作, 田浦健次朗, “InTrigger: 柔軟な構成変化を考慮した多拠点に渡る分散計算機環境,” *情報処理学会研究報告 2007-HPC-111*, pp. 237–242, 2007.

[15] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams, “Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking,” *Int. J. Comput. Sci. Eng.*, vol. 4, no. 2, pp. 73–87, Jul. 2009.

[16] K. N. Abazajian, J. K. Adelman-McCarthy, M. A. Agüeros, S. S. Allam, C. A. Prieto, D. An, K. S. J. Anderson, S. F. Anderson, J. Annis, N. A. Bahcall, et al., “THE SEVENTH DATA RELEASE OF THE SLOAN DIGITAL SKY SURVEY,” *Astrophys. J. Suppl. Ser.*, vol. 182, no. 2, pp. 543–558, Jun. 2009.

[17] M. Tanaka and O. Tatebe, “Workflow Scheduling to Minimize Data Movement Using Multi-constraint Graph Partitioning,” in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 65–72.