大規模ワークフローに対応した Pwrakeシステムの設計と実装

田中 昌宏[1,3,a)]　建部 修見[2,1,3,b)]

概要：ポストペタスケール時代における大規模データ処理のためのワークフローシステムを実現するため，分散化した Pwrake システムの設計と実装を行う．これまで 1 プロセスで行っていた Pwrake マスタの機能のうち，タスクの並列実行の部分を複数のプロセスに分散させ，それぞれ別のノードで実行する設計とする．評価として，分散化した Pwrake プロセスのメモリ使用量について測定した．その結果，想定する 100 万コア及び 1 億タスクの場合でも，分散化した Pwrake プロセスのメモリサイズは現実的な範囲に収まるという見通しを得た．

1. はじめに

科学データの大規模化が進む中，そのデータを活用したデータインテンシブなサイエンスを推進するため，100 万コアを搭載したポストペタスケールシステムの活用が期待されている．しかし，現在のシステムソフトウェアではそのような大規模なシステムにおいて性能を発揮することは容易ではない．本研究では，ポストペタスケール以降でも高い性能を発揮することを目標とした大規模データ処理実行基盤システムの研究開発を行っている．

大規模なシステムでは多くの場合，ジョブを投入するために PBS のようなバッチシステムが用いられる．一方，依存関係のある多数のプロセスを投入するには，処理内容や依存関係を記述した「ワークフロー」を記述し，それに基づいて各計算ノードで計算プロセスを並列分散実行する．大規模な処理を分割し，並列実行可能なプロセスに分割することができれば，並列プログラムを書くことなくシングルコア用のプログラムを活用して並列処理が実現できる．そのようなワークフローの処理系として，Pegasus[1]，Swift[2]，GXP make[3] などがある．

我々は，Pwrake (Parallel Workflow extension for RAKE)[4], [5] というワークフローシステムを開発している．Pwrake は，Rake という 記述力が高い Ruby 版ビルドツールをベースに，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 [6] を利用し，ローカリティを高めるスケジューリング [7] により，高い I/O 性能を発揮する処理を可能にする．

Pwrake は，マスタ・ワーカ方式の分散プログラムであり，単一のマスタがワークフローの実行を統括している．この設計が，スケールアウトの障害となっている．そこで，本研究では，ポストペタスケール以降の大規模ワークフローシステム実現に向けた第一歩として，Pwrake のタスク制御を分散化する提案を行う．本稿では，Pwrake プロセスのメモリ使用量について検証を行い，ポストペタスケールシステムにおける適用可能性について評価する．

本稿の構成は以下の通りである．第 2 節で関連研究について述べ，第 3 節で我々がこれまでに開発した Pwrake の概要について述べる．第 4 節で Pwrake 分散化の設計について述べ，第 5 節で実装について述べる．第 6 節で評価について述べ，第 7 節でまとめと今後の課題について述べる．

2. 関連研究

Megascript[8][9] は, 大規模な並列処理を行うための Ruby ベースのタスク並列スクリプト言語である．各タスクを「ストリーム」と呼ばれる通信路によって接続し，タスク間のデータの受け渡しを実現する．そのため，ファイルを介したデータインテンシブなワークフロー向けではない．

Xcrypt[10] は，大規模並列計算機へ多数のバッチジョブを投入するための Perl ベースのジョブ並列スクリプト言語である．submit による非同期実行，sync による同期機

1 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
2 筑波大学 システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba
3 科学技術振興機構 CREST
JST CREST
a) tanaka@hpcs.cs.tsukuba.ac.jp
b) tatebe@cs.tsukuba.ac.jp

構を提供する．タスクスケジューリング機構はバッチスケジューラに依存する．

ファイル単位で並列性があれば Makefile のタスク記述方式が適している．GXP make [3] は，GNU make と GXP[11] 分散シェル環境に基づく並列分散ワークフローシステムである．GNU make からタスクの代わりに mksh プロセスを起動することによりタスクをフックし，スケジューラを通して GXP 分散シェルを通してプロセスを起動する．そのため，並列タスク数と同じ数の mksh プロセスを同一ノードで起動する必要がある．また，ワークフローの記述は GNU make の仕様の範囲内で可能である．

3. 並列分散ワークフローシステム Pwrake

3.1 Rake の概要

ここで，並列分散ワークフローシステム Pwrake のベースとしている Rake について簡単に述べる．Rake は，ビルドコマンド make の Ruby 版である．デフォルトでは Rakefile というファイルに実行プログラムや依存関係を記述する．Rake の大きな特徴は，Rakefile が Ruby スクリプトそのものということである．Rake 側で用意したタスク定義メソッドにより，あたかもタスク定義言語のような記述が可能となる．このように，特定目的の言語を定義する際に，ホスト言語（ここでは Ruby）が用いられているものは，外部 DSL (Domain Specific Language) と呼ばれている．Ruby は様々な方面で DSL として使われているが，Rake はその代表格といえる．

3.1.1 Rakefile 記述例

次のコードは，プログラムをビルドするための典型的な Rakefile の例である．

```
1  SRCS = FileList["*.c"]
2  OBJS = SRCS.ext("o")
3
4  task :default => "prog"
5
6  file "prog" => OBJS do |t|
7    sh "cc  o prog #{t.prerequisites.join(' ')} "
8  end
9
10 rule ".o" => ".c" do |t|
11   sh "cc –o #{t.name} #{t.prerequisites[0]}"
12 end
```

4, 6-8, 10-12 行目がそれぞれ Rake のタスク定義である．これらは Ruby の文法からみれば，`task`, `file`, `rule` メソッドの呼び出しである．メソッド引数中の `=>` の記号は，Ruby のキーワード引数を表し，メソッド本体へは Hash オブジェクトとして引き渡される．このキーワード引数は，Rake において依存関係を意味する．4 行目は，`:default` というターゲットとするタスクを定義する．（ターゲットが`:default` の場合は，ターゲットを省略した際の最終ターゲットを意味する．）キーワード引数 `:default => "prog"` によって，`:default` タスクが `"prog"` というタスクに依存することを示す．6 行目では，`"prog"` をビルドするタスクを定義している．`file` により，ターゲット名がファイル名とみなされ，タイムスタンプの比較により出力ファイルが存在しないか，入力ファイルより古い場合にのみタスクが実行される．`do` と `end` で囲まれた部分は，Ruby のコードブロックである．この部分がタスクの**アクション**といい，依存関係に従って実行される．ブロック内の `sh` メソッドは Rake で定義されたものであり，引数の文字列をコマンドラインとして実行する．10 行目の `rule` は，ルールの記述である．上の例では，拡張子が `".o"` のファイルが `".c"` のファイルに依存する，という意味である．

3.1.2 ワークフロー向け Rake の特徴

Rake の最大の特徴は，記述言語として汎用のプログラミング言語の Ruby を用いている点である．これによって，複雑な科学ワークフローについても，柔軟に記述することが可能である．例えば，科学ワークフローでは入出力ともに同じ拡張子である場合も多く，その時には柔軟なパターンマッチが必要となる．`rule` では，make と同様な拡張子によるルールだけでなく，正規表現によるパターンマッチを用いることができる．さらに，ターゲットファイル名から入力ファイル名を変換ルールをコードブロックで与えることも可能である．

3.2 Pwrake の概要

Rake の高い記述性はそのままに，複数のマシンを用いた並列分散実行を可能にしたものが Pwrake (Parallel Workflow extension for RAKE)[4], [5] である．Pwrake で用いる Rakefile は，Rake とできるだけ互換性を持たせる方針を取っている．すなわち，Rake 用に記述したワークフローをそのまま Pwrake で並列分散実行することができる．Pwrake により，指定されたコア数分だけプロセスを並列に実行したり，タスクの依存関係を基にして並列実行可能なタスクを自動的に並列実行したりすることが可能となる．Pwrake は，make や Rake と同様に，ファイルのタイムスタンプに基づいてタスクを実行する．そのため，複数ノードを用いた分散処理の場合はネットワークファイルシステムを前提としている．Pwrake は，NFS の利用も可能であるが，広域分散ファイルシステム Gfarm を利用することによってスケーラブルな並列処理性能の実現に成功している [4].

3.3 Pwrake の実装

Pwrake では，メソッドの振る舞いを動的に変更できるという Ruby の特徴を活かして，Rake コードのうち振る舞いを変えたい部分だけを実装している．タスク情報を保持する Rake::Task クラスなどについては Pwrake においてもそのまま利用している．

Rake から拡張して実装した部分の 1 つに，実行可能なタスクの探索がある．Rake では，最終ターゲットのタスクを

起点に，依存関係を辿って深さ優先探索を行う．前段タスクが存在せず直ちに実行可能であるタスクがみつかった場合に，そのアクションをその都度実行する．一方，Pwrake では，深さ優先探索を行い実行可能なタスクがみつかった場合にそのタスクをキューに入れ，続けて探索を行う．こうしてツリー全体を探索したのち，キューに入れられたタスクを並列実行する．キューが空になると再びツリーを探索してタスクを取得する．

Gfarm を用いる場合は，ローカリティを活用したタスクスケジューリングが可能である．それに関して 2 種類の手法が実装されいる．1 つ目は，ツリーを探索してタスク取得するごとに，入力ファイルの格納ノードについての情報を，Gfarm の gfwhere コマンドを用いて取得し，その情報に基づいてできるだけ格納ノードと同じノードで実行するという手法である．2 つ目の手法は，多制約グラフ分割を利用したデータ移動を最小化スケジューリング [7] である．この手法では，ワークフロー実行前に Rakefile からグラフを作成し，グラフ分割を行った結果からタスク配置を決めることができる．

タスクの並列分散実行についても，Pwrake で新たに実装した．これまでに開発した Pwrake は，図 1 のように，単純なマスタ・ワーカー型である．マスタの Ruby プロセス内では，タスクの並列実行のため，ワーカーのコア数と同じ数の Ruby Thread をワーカースレッドとして起動する．各スレッドでは，あらかじめリモートのワーカーノードへ SSH で接続しておく．そしてキューから実行可能タスクを取り出し，そのアクションを実行する．アクション中に sh メソッドがある場合，引数として与えられた外部コマンドを，SSH を通してリモートのワーカーノードで実行する．Ruby 1.9 の Thread はネイティブスレッドを利用しているものの，GVL (Giant VM Lock) により同時に 1 スレッドのみが実行される．そのためアクション部分の Ruby スクリプトは複数コアでも高速化はされないが，sh メソッドで起動される外部プロセスの実行が並列化される．

4. Pwrake 分散化の設計

4.1 本研究の目標

ポストペタスケールデータインテンシブワークフローの実行するため，Pwrake をベースとし，大規模なワークフローを実行できるような実装を行う．目標とする規模として，

- **100 万コア** からなるシステムの利用
- **1 億タスク** からなるワークフローの実行

とする．

この目標を達成するためには，現状のマスタ・ワーカ型の構成では，次のような問題がある．まず，100 万コアを利用するためには，マスタプロセス内で 100 万個のスレッドを起動する必要がある．さらに，マスタプロセスからワーカーノードに 100 万本の接続を行う必要がある．また，1 億タスクのワークフローを実行するためには，マスタプロセス内で 1 億個の Rake::Task オブジェクトを作成し，依存関係を辿ってタスクの探索を行う必要がある．このように，マスタプロセスが 1 つの場合，負荷が大き過ぎるため前述の目標を達成できない．

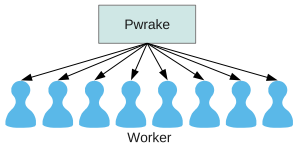

図 1 単一マスタ Pwrake

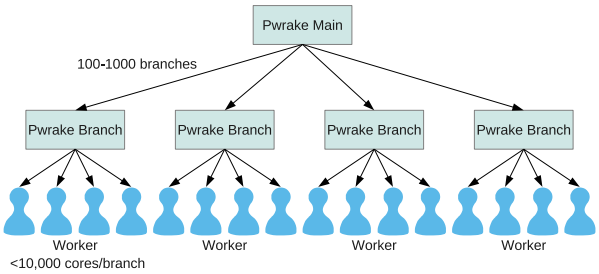

図 2 分散 Pwrake

4.2 分散 Pwrake の設計

前述の目標を達成するため，新しい Pwrake では，次の 3 階層からなる設計を行う．

- **メイン**
- **ブランチ**
- **ワーカー**

これらは図 2 のようなツリー状の構成とする．この名前は，本社（メイン）が全体を統括し，複数の支社（ブランチ）の下で，従業員（ワーカー）が実際の仕事を行う，というイメージで名付けた．

メインは，ユーザがコマンドラインから起動するプロセスである．ワークフローを統括し，スケジューリングによってどのタスクをどのノードで実行するかを決める機能を持つ．

ブランチは，単一マスタ Pwrake の機能のうち，スレッドを用いて実現していたタスクの並列実行の部分を担当する．ブランチは，メイン Pwrake から SSH によりリモートプロセスとして起動される．ブランチの数として 100 個程度を想定している．100 個のブランチが 100 万コアの制御を分担することにより，1 つのブランチプロセス内の並列スレッド数を 10,000 以下とすることができる．また，ブランチからワーカーへの接続数も同様に減らすことができる．

ワーカーは，処理プロセスが実行されるノードであり，ブランチから接続される．ノードあたりのコア数は 10-50 とし，1 ブランチあたり 1,000 ノード程度のワーカーに接

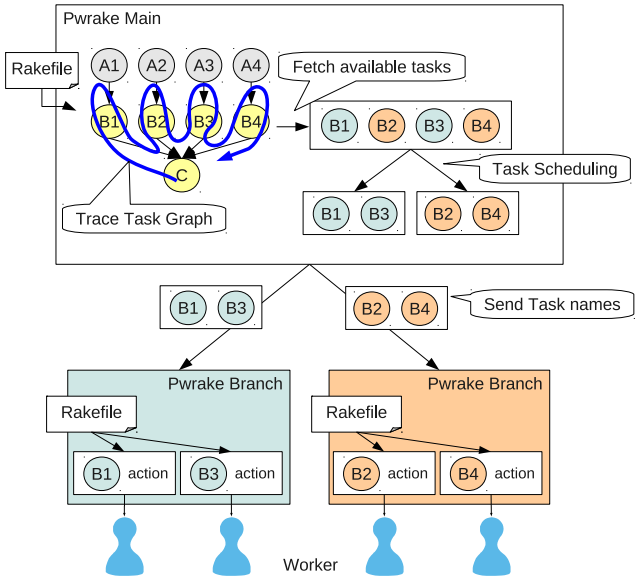

図 3 分散 Pwrake におけるタスク制御

続することを想定する．

4.3 タスク分散の課題と設計

ワークフローを分散制御する上で課題となるのが，タスクの情報をリモートに転送する方法である．これは，ワークフロー処理系が採用しているワークフロー定義の方法に大きく依存する．

Pegasus[1] において採用されているワークフローを記述する言語の 1 つに，DAG を XML で表現するための DAX と呼ばれる仕様がある．DAX では，全てのタスクについて，プログラム名，引数，入出力ファイルなどが記述される．これはタスクがシリアライズされた状態といえる．このような方式であれば，個々のタスク定義を送信するだけで，リモートノードでのタスクの実行が可能である．一方，この方式の欠点は，ワークフローが大規模になりタスク数が多くなると，タスクデータの量が極めて多くなるという点である．

Pwrake では，前述のようにワークフロー定義言語として Rakefile を採用している．rule による記述により，多数のタスクを代表することができる．タスクの情報をリモートに転送する方法として，Rakefile をパースした際のオブジェクトをシリアライズして転送する方法が考えられる．しかし，タスク情報に含まれるタスクのアクションは Ruby のコードブロックであり，Ruby の実行時コンテキストが含まれるためシリアライズすることができない．

そこで，今回の設計では，Rakefile 自身を送信する方法を採用する．その動作概要を図 3 に示す．まず，メインとブランチのプロセス全てで，同一の Rakefile を読み込む．メインプロセスでは，Rakefile に記述されたワークフロー全体の情報に基づき，実行可能なタスクをリストし，どのワーカーで実行するかのスケジューリングを行う．そしてそのタスクリストを担当するブランチへ送信する．ブランチ側では，受け取ったタスク名から Rakefile に書かれた rule などの定義に基づいてタスクを生成して実行する．

5. 実装

5.1 遠隔通信の実装

Pwrake では，利便性のため，ユーザがメインプロセスを起動すると，設定に従って自動的にブランチ，ワーカープロセスを各ノードで起動する方針を取る．これを手軽に実現する方法として SSH を採用した．通信についても，SSH 接続の標準入出力を利用する．Ruby ver. 1.9 で新たに導入された spawn を用いて SSH プロセスを起動し，その I/O を通じて通信を行う，というように実装した．通信は行単位で行い，1 つの経路で複数のコアに関する通信を多重化するため，行の先頭にワーカー ID やステータスを付加して区別するという設計を行った．

5.2 軽量スレッド Fiber の利用

Pwrake では，複数のタスクのアクションを同時に実行する必要がある．前述のように，これまでの Pwrake では Ruby の Thread を用いて実装した．Thread により，スレッド間の切り替えが自動的に行われ，また，I/O 待ちのスレッドは自動的に sleep することから，プログラミングが容易である．一方，Ruby ver. 1.9 には，Fiber[12] と呼ばれる軽量スレッドが導入された．Thread と異なり，Fiber では，複数の処理間での制御の流れをスイッチすること（コンテキストスイッチ）を，プログラム中で明示的に行う必要がある．ただし，この違いは Rakefile の仕様に対してほとんど影響しない．今回，大規模なワークフローに対応するため，より軽量な Fiber を利用した実装を行った．

Fiber を用いたタスク並列実行の流れは次のようになる．まず各コアごとに Fiber スレッドを起動しておき，Fiber 内でタスクのアクションを順次実行する．アクション内部で sh メソッドが呼ばれると，SSH 接続へ外部コマンドを送信し，Fiber の親コンテキストにスイッチする．Fiber の親コンテキストでは，ワーカーへの SSH 接続に対して，IO.select を用いた I/O 待ちを行っている．入力を受け取ると，担当 Fiber にその入力を渡して，コンテキストをスイッチする．

6. 評価

6.1 評価環境

今回の評価実験では，実際に 100 万コアのシステムを用意できないため，研究室規模のクラスタを用いたマイクロベンチマークを行い，得られたデータから大規模システムへの適用可能性について評価をおこなう．用いた評価環境は，InTrigger プラットフォーム [13] の筑波大学ノードで

表 1　測定環境

Intrigger	筑波拠点
CPU	Xeon E5410 (2.33GHz)
主記憶容量 (GiB)	32
コア数/ノード	8
最大使用ノード数	16
ネットワーク	10G Ethernet
OS	Debian 5.0.7
Kernel	2.6.26-2-amd64
Ruby	ver. 1.9.3p125

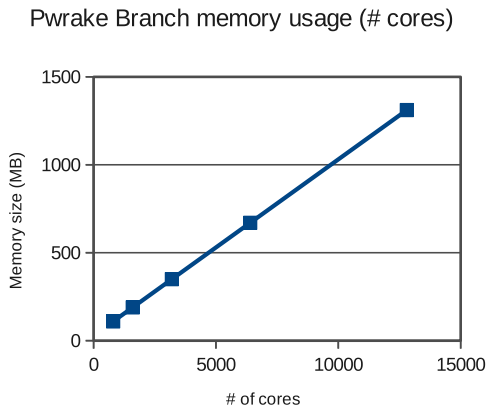

図 4　コア数に関するブランチプロセスのメモリサイズ．コア数と同じ数の Fiber が起動．1 ノードに 100 本の SSH 接続を行い，1 接続あたり 8 コアを制御．

ある．用いた計算ノードの仕様を**表 1** に示す．今回の評価では，Pwrake システムが使用するメモリについて測定する．測定では，ワークフロー終了時にブランチプロセスが占めるメモリサイズを ps コマンドにより取得した．

6.2　コア数とメモリ使用量の関係

本研究の目標は，1 ブランチあたり 10,000 コアを扱うことである．ここでは，まず，ワーカーのコア数を増やした場合にブランチプロセスが必要とするメモリサイズについて評価する．1 つのブランチから接続するワーカーとして，1,000 ノード程度を想定しているが，今回の評価はシステムのメモリ使用量についての評価であるから，16 ノードを用いて，1 ノードあたり SSH 接続を 100 本行うことによって代用した．1 接続あたり 8 コア分のタスクを扱うから，接続数の 8 倍の Fiber スレッドがブランチ内で起動される．ここでは与えたタスク数は 1 個であり，タスク処理に使われるメモリは十分小さい．ノード数にして 1 から 16 まで，コア数にして 800 から 12,800 までについて，ブランチプロセスのメモリ使用量を測定した．測定結果を 図 4 に示す．この結果は，使用コア数に関して良い線形性を示している．各ブランチプロセスが必要なメモリサイズは，1 コアあたりほぼ 100 kB であることがわかった．目標では 1 ブランチあたり 10,000 コアのワーカーを扱うことを

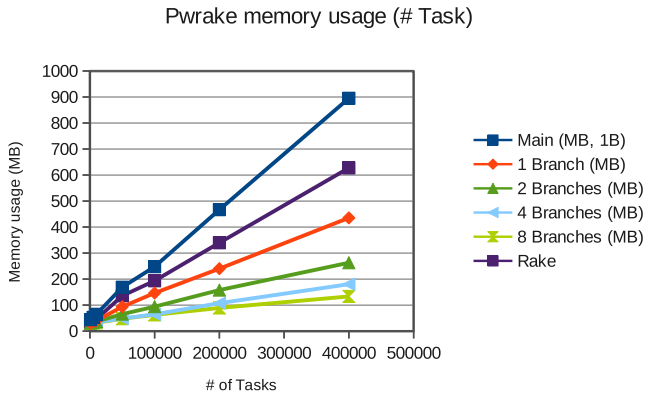

図 5　タスク数に関する メイン、ブランチ各プロセスのメモリサイズ．

想定しているから，ブランチプロセスには約 1 GB のメモリが必要となる．これは，現状の計算機のスペックでも十分に許容可能なメモリサイズであり，実際，本測定で動作を実証できた．実際には，これに加えて，次に述べるタスク実行に使用するメモリが加わることになる．

6.3　タスク数とメモリ使用量の関係

本研究が目標とするタスク数は，1 億程度である．分散化 Pwrake では，約 100 のブランチにタスクを分散するから，ブランチ 1 つにつき 100 万タスク程度を処理することを想定する．本評価では，大量のタスクを与えた場合に Pwrake システムの各プロセスが使用するメモリサイズについて調査する．評価ワークフローとして使用した Rakefile は次の通りである．

```
1  src = FileList["src/*.c"]
2  obj = src.ext("o")
3
4  rule ".o" => ".c" do |t|
5  end
6
7  task :default => obj
```

このワークフローでは，タスク生成を行うがプロセスの起動は行わない．ワークフローのタスク数は，src ディレクトリのファイル数によって決まる．本評価では，最大 40 万タスクまで測定した．使用するワーカーは 1 コアとした．したってリモート接続や Fiber に使われるメモリ量は十分小さい．

タスク数を変えた場合のメインプロセス，ブランチプロセスのメモリ使用量を測定した結果を図 5 に示す．ブランチの数は 1 から 8 まで測定した．この図は，どのプロセスもタスク数の増加とともにメモリ使用量が線形に増加することを示している．1 ブランチの場合，1 タスクあたりのメモリサイズは約 1kB である．これを 100 万タスクに外挿すると，ブランチプロセスのメモリサイズは 1 GB 程度となり，現状の計算機のスペックでも実現可能なサイズである．

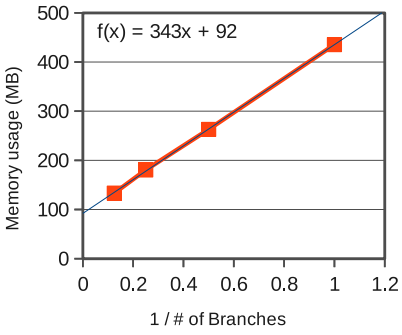

図 6 タスク数が 4×10^5 の場合の，ブランチ数の逆数に対するブランチプロセスのメモリサイズの関係．

40 万タスクの場合について，ブランチ数の逆数に対するメモリ使用量の関係をプロットしたものが図 6 である．1 ブランチの場合，40 万タスクを与えると，メモリサイズは 435 MB になる．この図は，ブランチの数を増やすことにより反比例してメモリ使用量が減少することを示しており，Pwrake プロセスを分散化した効果があることが確認でききる．

ここで明らかになった問題点として，今回の実装では，図 5 に示されるように，メインプロセスのメモリ使用量が大きいことである．メインプロセスでは，実行すべきタスクを見つけてスケジューリングを行うため，一度はワークフローのグラフ全体を探索する必要があり，その際にタスクオブジェクトが作成される．測定結果から，約 2.1 kB/タスクであるから，1 億タスクの場合にはメインプロセスのメモリとして約 200 GB が必要となる．このサイズのメモリは，将来的には実現可能であると思われる．しかし，グラフ探索にかかる処理コストにも注意する必要がある．ワークフローのグラフ全体が必要となるケースとして，グラフ分割を用いるような高度なスケジューリングががある．そのような手法を用いずにタスクを分配してもよいのであれば，メインプロセスのメモリ使用量を減らせる可能性がある．

7. まとめと今後の課題

ポストペタスケールデータインテンシブワークフローシステムを実現するため，Pwrake システムを分散化する設計と実装を行った．これまで 1 プロセスで行っていた Pwrake マスタの機能を分散化させ，タスクの並列実行の部分を 100 程度のブランチに分散化し，それらを 1 つのメインプロセスが統括する，という設計とした．この設計により，100 万コアの計算機による 1 億タスクのワークフローを実行することが目標である．評価として，分散化した Pwrake プロセスのメモリ使用量について測定した．その結果，100 プロセスに分散化した Pwrake ブランチにおいて，100 万タスク，10,000 コアを扱うために必要なメモリサイズは，それぞれ 1 GB 程度であり，現状の計算機でも実現可能であるという見通しを得た．今後の課題として，メインプロセスのメモリ使用量が大きいため，これを抑えることが大きな課題として残った．これはスケジューリング方法とも関連する課題である．

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」の支援により行った．

参考文献

[1] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, Vol. 13, No. 3, pp. 219–237, 2005.

[2] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. *Services, IEEE Congress on*, Vol. 0, pp. 199–206, 2007.

[3] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun'ichi Tsujii. Design and Implementation of GXP Make – A Workflow System Based on Make. *eScience, IEEE International Conference on*, Vol. 0, pp. 214–221, 2010.

[4] Masahiro Tanaka and Osamu 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, pp. 356–359, New York, NY, USA, 2010. ACM.

[5] Pwrake. http://github.com/masa16/pwrake.

[6] Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. Gfarm Grid File System. *New Generation Computing*, Vol. 28, No. 3, pp. 257–275, 2004.

[7] Masahiro Tanaka and Osamu Tatebe. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. *Cluster Computing and the Grid, IEEE International Symposium on*, Vol. 0, pp. 65–72, 2012.

[8] 松本真樹, 片野聡, 佐々木敬泰, 大野和彦, 近藤利夫, 中島浩. ヘテロ型大規模並列環境の階層型タスクスケジューリングの提案と評価. 情報処理学会論文誌. プログラミング, Vol. 2, No. 1, pp. 1–17, 2009.

[9] 三田明宏, 仲貴幸, 松本真樹, 大野和彦, 佐々木敬泰, 近藤利夫. Megascript における大規模ワークフローの縮約機構の設計. 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], Vol. 2011, No. 57, pp. 1–8, 2011.

[10] 平石拓, 安部達也, 三宅洋平, 岩下武史, 中島浩. 柔軟かつ直観的な記述が可能なジョブ並列スクリプト言語 xcrypt. 先進的計算基盤システムシンポジウム SACSIS2010 論文集, Vol. 2010, No. 5, pp. 183–191, 2010.

[11] Kenjiro Taura. GXP: An Interactive Shell for the Grid Environment. In *In Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA 2004)*, pp. 59–67. IEEE Computer Society, 2004.

[12] 芝哲史, 笹田耕一. Ruby1.9 での高速な Fiber の実装. 第

51 回プログラミング・シンポジウム予稿集, pp. 21–28, 2010.

[13] 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.