博士論文

題目

性能変動の生じる実行環境に適した大規模ワークフローの高速スケジューリング手法に関する研究

指導教員

近藤 利夫 教授

平成25年度

三重大学大学院 工学研究科 システム工学専攻
計算機アーキテクチャ研究室

松本 真樹 (410D053)

目 次

図 目 次

表 目 次

1　序論

1.1　本研究の背景

近年，増大する大規模計算の高速処理の要求に応えうる並列処理への期待がますます高まっている．特に広域ネットワーク上に分散しているクラスタ群を利用する大規模な分散並列処理はコストパフォーマンスやスケーラビリティの面で利点があるため，大きな注目を集めている．

このような並列処理環境では多くの科学技術の分野で様々なプログラムを使い膨大な量のデータに対するパラメータサーベイと，その結果を統計ソフト等を通して解析，評価をするワークフロー型の並列処理が用いられる．例えば遺伝子解析の分野では，遺伝子解析ソフト BLAST[1] や FASTA[2] を使い膨大な量の既存の DNA 情報と照合しその結果を ClustalW[3] を使い解析を行い，多くの未知の病原菌を特定する．また分子動力学の分野では分子シミュレータ AMBER[4] 等を使い各原子の振る舞いをシミュレーションし，統計解析ソフト R[5] を用いて解析を行い物体のひび割れを原子レベルで評価する．

このような大規模なワークフロー型の並列処理で高いスループットを得るには，実行単位となる各タスクを依存関係に基づいてどのような順番で各計算機に割り当てるかというタスクスケジューリングが重要になる．このためタスク間の依存関係を考慮したスケジューリング手法が多く提案されている．しかし，ワークフロー型並列処理全体の実行時間であるスケジューリング長において短い結果を得る従来手法は計算コストが高く，タスク数・ホスト数が大規模な並列処理では現実的な時間で解くことは難しい．また，ワークフローの実行中に外部プロセスなどの影響により想定外の負荷が発生する恐れがある．従って計算ホストの性能が常に一定である可能性は低く，動的な性能変動を考慮したスケジューラの計算時間はさらに膨大になる．このため，大規模なワークフローを高速にスケジューリングする手法の開発が熱望されている．

1.2 タスクスケジューリング

これまでに様々なタスクスケジューリング手法が提案されてきた．大きく分類して静的スケジューリング手法と動的スケジューリング手法に分けられ，前者はワークフロー実行前に一度だけスケジューリングする手法で，後者は実行時の性能変動に対応する手法である．

静的スケジューリング手法は多くの研究がされており，その計算コストは高いがスケジューリング長を短くできる．特にレベルスケジューリングに基づく手法は多く提案されており，古典的な手法では Mapping Heuristic(MH) [6] や Generalized Dynamic Level(GDL) [7]，Best Imaginary Level(BIL) [8] がある．また比較的引用回数の多い手法として，各タスクの実行時間の平均値を用いてレベルを定義する Heterogenous Earliest-Finish-Time(HEFT) [9] や Levelized Duplication Based Scheduling(LDBS) [10] がある．またメタヒューリスティックでレベルを最適化する手法 [11, 12] もある．レベルスケジューリング以外の手法としては，Modified Critical Path(MCP) [13] や Dynamic Critical Path(DCP) [14] のようなワークフロー中のクリティカルパスに注目した手法も多く提案されている．しかし実行時の性能変動を考慮していないため，静的スケジューリング手法でスケジューリング長を短くできても実際の実行時間に反映されるとは限らず，高い実行効率を得ることが難しい．

動的スケジューリング手法には，タスク間に依存関係が無いものを対象とした，マスタワーカー型がある．これは各ワーカーホストがアイドル状態となるたびにマスタホストへタスクを要求するため，実行環境における計算性能や通信性能の変化に柔軟に対応できる．もっとも単純な手法としては，アイドル状態になったマシンにタスクを割り当てる Opportunistic Load Balancing(OLB) [15] や，タスクの実行時間が短いホストへ割り当てる User Directed Assignment(UDA) [15] がある．また，各タスクの計算量やホストの計算性能等を考慮する手法としては，Min-min Heuristic [16] や Greedy 法 [17] 等がある．これらに対し Random Steal（RS）法 [18] では，あらかじめタスクを全ホストに分配し，実行タスクのなくなったホ

ストがランダムに選択したホストにタスクの分配を要求する．この手法ではマスタホストが不要かつ通信回数が最小限で済むため，安定して高性能が得られる．しかしワークフローを扱う場合，これらの手法は大域的な依存関係を考慮できないため，静的スケジューリング手法と比較して実行効率が低下する恐れがある．

ワークフローを考慮した動的スケジューリング手法として，AHEFT [19] や DCP-G [20]，Blythe らの手法 [21] 等のような動的再スケジューリング手法が提案されている．これらの手法は以下の 1 4 の手順により性能変動が発生するような実行環境においても，静的スケジューリング手法なみのスケジューリング長を得ることができる．(1) 静的スケジューリング手法のような最適解に近いスケジューリング結果を求める．(2) そのスケジューリング結果にしたがって各タスクを実行していく．(3) ホストの計算性能や通信性能に大きな変化が発生した場合，未実行のタスクに対して性能変化を考慮した上で再度スケジューリングを行う．(4) 残りの未実行タスクはこの再スケジューリングの結果に従って実行される．この手法では，ワークフローの依存関係を考慮しつつ，性能の変化に対応してワークフローを実行することができる利点がある．しかしこのような再スケジューリングを行う手法は，デマンドドリブン型の単純な動的スケジューリングと比較してスケジューリングにかかる計算コストが大きくなり，これがワークフローの実行において大きなオーバーヘッドとなる恐れがある．

1.3 本研究の目的と論文構成

大規模なワークフローを高速にスケジューリングするために，複数のスケジューラを階層的に配置し，上位層の大域スケジューラと下位層の局所スケジューラで異なるスケジューリング方法を使い分ける階層型スケジューリング手法を提案している [22]．特に，下位層の局所スケジューラに焦点を当て，ワークフローを高速にスケジューリングする手法の開発を目的としている．また，ワークフロー実行中における計算ホストの

動的な性能変動にも対応することも目的としている．

本論文は 5 章からなる．本章に続く 2 章では，科学技術で用いられるワークフローを開発，実行するようなワークフロー実行システムについて紹介する．そして，そのようなワークフローシステムにおいて必要となるタスクスケジューリングについて明らかにする．また，階層型スケジューリング手法について紹介する．3 章では静的スケジューリング手法の高速化について提案する．3 章の前半では，下位層のスケジューラで扱うタスクの依存関係に着目し，適応的にスケジューリング処理を切り替える適応型スケジューリング手法 (AS) について提案する．これは，HEFT と比較してスケジューリング時間を 1/50 ∼ 1/100 程度に削減できた．しかし類似性の高いサブワークフローが複数含まれている場合，AS のアルゴリズムでは処理の切り替えを効率的に行うことができず，スケジューリングの計算時間の大幅な削減が行えない可能性がある．そこで AS を改良し，スケジューリング処理の適応的な切り替えを再帰的に行う再帰的適応型スケジューリング手法 (RAS) を提案する [23]．RAS を抽象シミュレーションで評価を行った結果，10,000 タスク規模でスケジューリングの計算時間を 1/100 程度に削減できた．3 章の後半では RAS について示す．4 章では，3 章で述べる静的スケジューリングを動的性能変動に対応させた動的再帰的適応型スケジューリング手法 (DRAS) について提案する．これは動的再スケジューリング手法の一種だが，高い計算コストを削減させるための幾つかのテクニックを適応させている．抽象シミュレーションで評価を行った結果，1,000 タスク規模でスケジューリングの計算時間を 1/6 程度に削減できた．最後に 5 章で本研究のまとめを行い，今後の展望について述べる．

2　背景

2.1　ワークフローシステム MegaScript

本研究で開発したスケジューリング手法はワークフローシステムに組み込んで用いることを想定している．そこで本節では，既存のワークフローシステムで組み込み先の一つである MegaScript について紹介する．

2.1.1　言語の概要

MegaScript は 2 階層並列モデルの上位層を記述するための言語である．逐次または並列の独立したプログラムを計算タスクとして扱い，これらのタスクを並列実行させる．各タスクは並行並列に動作し，ストリームと呼ばれる通信路を介することでタスク間のデータ受け渡しを行う．

処理の主要な部分は外部プログラムとして用意するため，MegaScript プログラム内には主に並列実行に関する制御情報を記述する．実行制御に要する計算量は全体に対してわずかであるため，実行効率より記述性や拡張性を優先し Ruby [24] をベースとするオブジェクト指向スクリプト言語としている．

2.1.2　タスクとストリーム

タスクは MegaScript とは独立したプログラムであるため，ユーザは任意の言語でタスクを作成することができる．このため，既存プログラムを部品として流用したり，処理内容に応じて記述言語を変えるといったことが自由にできる．また，MegaScript はタスク内部の処理に関与せず，タスク間の情報のやりとりには標準入出力を利用し，行単位で一つのアトミックなメッセージとみなす．

ストリームは，あるタスクの標準出力の内容を他のタスクの標準入力に流し込むための通信路であり，MegaScript におけるタスク間通信を実現する．ストリームの入出力端にはそれぞれ複数のタスクを接続するこ

とができ，一対多，多対多などの通信を簡潔に記述することができる．入力端に複数のタスクを接続した場合，メッセージは非決定的にマージされる．また，出力端に複数のタスクを接続した場合は，メッセージはそれぞれのタスクにマルチキャストされる．

MegaScript 上では，タスクやストリームの生成･操作は，それぞれ Task, Stream クラスを用いて行う．同じ種類のタスク/ストリームを複数生成する場合は，それぞれ TaskArray, StreamArray クラスを用いて，タスク配列/ストリーム配列として生成でき，接続などの操作を一括して行うことができる．

2.1.3 プログラミングモデル

MegaScript では，並行並列に実行するタスク間をストリームで接続し合うワークフローの形状を記述する．例として，プログラム gen_data が生成したデータに対し，プログラム sim で引数 1〜1000 のパラメータサーベイを行った後，それぞれの結果をプログラム plot により可視化し，最後にプログラム composite で一つの画像に合成するという処理を考える．この処理は図 2.1 に示すようなワークフローとして表現できる．

また，そのプログラムは図 2.2 のように記述できる．MegaScript プログラムでは，まずワークフローの構成に必要なオブジェクトを生成し，connect メソッドによって，それらの間の結合を定義する．その後，create メソッドによってプロセスなどの実体を生成する許可を与える．最後に schedule メソッドを呼ぶことで，生成許可済みのオブジェクトに対するスケジューリングが行われ，タスクの実行が開始される．

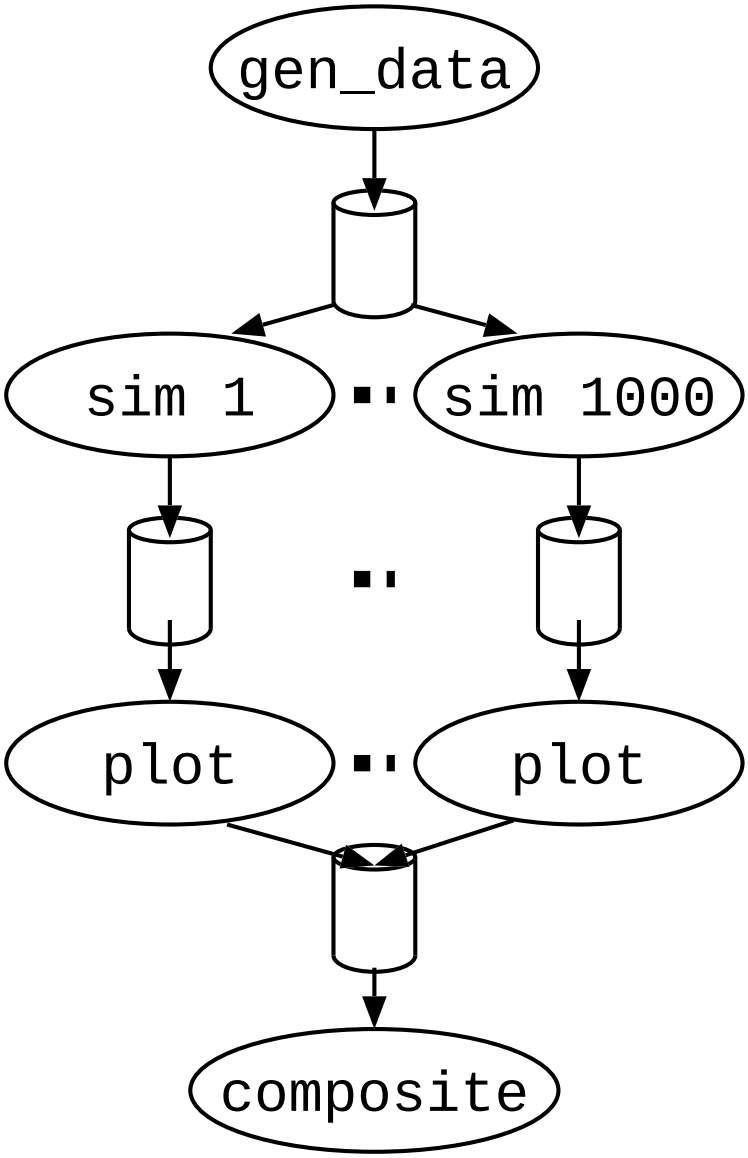

図 2.1: ワークフローの例

```
1. N = 1000
2. t1 = Task.new(”gen_data”)
3. t2 = Task.new_array(N, ”sim”, 1..N)
4. t3 = Task.new_array(N, ”plot”)
5. t4 = Task.new(”composite”)
6. s1 = Stream.new
7. s2 = Stream.new_array(N)
8. s3 = Stream.new
9. s1.connect(t1, IN)
10. s1.connect(t2, OUT)
11. s2.connect(t2, IN)
12. s2.connect(t3, OUT)
13. s3.connect(t3, IN)
14. s3.connect(t4, OUT)
15. t1.create
16. t2.create
17. t3.create
18. t4.create
19. s1.create
20. s2.create
21. s3.create
22. Scheduler.new.schedule
```

図 2.2: ワークフローのコード例

2.2 タスクスケジューリング

2.2.1 スケジューリング条件

一般のワークフローはDirected Acyclic Graph(DAG)で表現ができるが，MegaScriptのようにタスクの動的生成が可能なシステムもある．しかし，本論文では議論を簡単にするために，ワークフローは予め与えられており動的に変化しないものとする．従って本論文で扱うスケジューリング手法は，DAGのオフラインスケジューリングの一種になる．一般のDAGの形を取るオフラインタスクスケジューリングとして，様々な従来手法が提案されている．これらの手法はスケジューリング長を短くできるが，大規模なタスク数やホスト数を想定した並列処理においては，計算コストが非常に高くなり現実的な時間で解くのは難しい．したがって，スケジューリング長の短さを維持しつつ計算コストを大幅に低減することが必須である．

ワークフロー型の大規模並列処理を容易に実行でき，また効率よく自動で実行するようなシステムの需要は高い．しかし本論文で扱うタスクスケジューリングでは，以下の特徴を考慮する必要がある．

(1) 一般のワークフローではMegaScriptのようにネイティブプログラムを扱うことも多く，タスクの分割やタスク実行中のマイグレーションは行えない．

(2) 実行環境として複数のホストを想定しており，計算性能は非均質である．

(3) 計算資源を独占して利用できるとは限らないので，ワークフローの実行中に他のプロセスによってホストの計算性能が変わる可能性がある．

(4) MegaScriptでは各タスクの計算量や通信量がユーザーにより記述されるため，100% 正確とは限らない．

(5) 大規模な並列処理を目的としており，10 万 ~100 万規模のタスクやホストを扱えることが求められる．

(6) 独立したプログラムを粗粒度のタスクとして組み合わせ，ユーザが明示的にワークフローを記述する．このためデータの分散・収集や処理の流れの分岐といった，比較的単純なパターンの組み合わせになる．

(7) 科学技術の分野では，解析やシミュレーションを行うために，図 3.11 の *sim* や *plot* のように入力データや実行時引数を変えて同一プログラムを実行するパラメータスウィープを用いることが多く，これら実行は互いに依存関係がない．MegaScript のワークフローではこのようなパラメータスウィープを含んでいることが多い．

(8) 科学技術の分野では，解析やシミュレーションを行うために，図 3.11 の太枠内のように類似性の高いサブワークフローを多数含んだワークフローを用いることが多い．

ワークフロー型の大規模並列処理では DAG のタスクスケジューリングが必要になり，(2)，(5) の特徴により計算コストは非常に大きくなる．しかし，(6)，(7)，(8) の特徴により複雑な DAG の記述が難しいため，直並列グラフに近い単純なワークフローになることが多いと考えられる．また，(3) や (4) の特徴から厳密なスケジューリングは求められていない．しかし (3) を考慮した DAG のタスクスケジューラは厳密なスケジューリングを行うことに着目しているものが多く，それらの計算コストは高い．そのため，(2)，(5) の特徴を考慮した場合，スケジューリングの計算時間が非現実的になり，実システムで用いることは難しい．従って，スケジューリングの計算コストが小さく，動的な性能の変化に対してスケジューリング結果を補正するようなスケジューリング手法が必要になる．

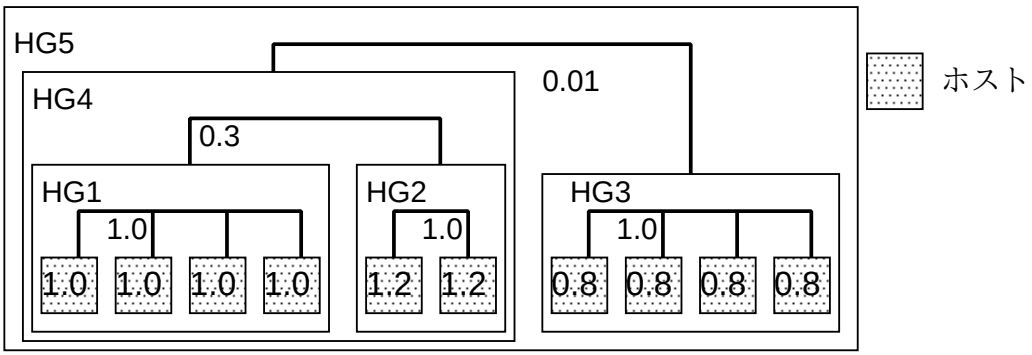

図 2.3: ホストの階層化

2.2.2 実行環境

大規模な広域分散環境の利用例として，SETI@home [25] 等がある．これは家庭などに存在する PC を集めて利用するもので，各ホストの処理性能や通信性能は非均質である．こうした方法では大量の計算機を確保できるが，ほとんどのホスト間で高速な通信を期待できず，また所有者の都合により頻繁かつ予告なくホストが利用できなくなる問題がある．このため，個々のタスクが完全に独立した大規模問題には有効ではあるが，タスク間通信が頻繁に起きる並列処理には適していない．

想定するワークフロー型の大規模並列処理としては，図 2.3 のように PC クラスタのようにある程度の規模・品質の計算資源が多数，広域に分散した環境を考える．この場合，各クラスタ内の通信性能は高速かつ均質である．したがって，階層型スケジューリング手法はこのような環境を対象としている．そして，本研究ではその中核となる各クラスタ内で実行する局所スケジューラに焦点を当てる．

2.3 階層型スケジューリング手法

階層型スケジューリング手法では，最内側のクラスタを対象に実行する局所スケジューリング手法と，それ以外の上位層に対して行う大域スケジューリング手法の二種類を用いる．これにより，全体のワークフローを高速にスケジューリングすることができる．大域スケジューリング手法では直下のホストグループに対するタスクの分配のみを行う．一方，局所スケジューリング手法では扱うタスク数やホスト数が大幅に削減されているため，従来手法のような短いスケジューリング長を得ることができる．

2.3.1 基本方針

1 章で述べたように様々な静的スケジューリング手法が提案されているが，MegaScript のようなワークフローシステムが必要とするヘテロ環境に対する DAG スケジューリングは，高速に短いスケジューリング長を得る手法が発見されていない．またワークフローシステムではタスク数やホスト数が多いため，計算量やメモリ消費量の大きいアルゴリズムは利用が難しい．また，実行環境の性能が動的に変動する場合を考えると，精度の高い静的スケジューリングを行っても，そのまま実行効率の向上に繋がるとは限らない．そこで，以下のようなハイブリッド型のスケジューリングを行う．

1. はじめにホストを階層モデル化し，各階層ごとにスケジューラを配置することで，段階的なスケジューリングを行う．また，最下層の均質環境とそれ以外とで，異なるスケジューリング手法を用いる．図 2.4 に，図 2.3 のホストグループに対するスケジューラの配置を示す．

2. 静的スケジューリングを行い，その後，動的スケジューリングによる補正を行う．

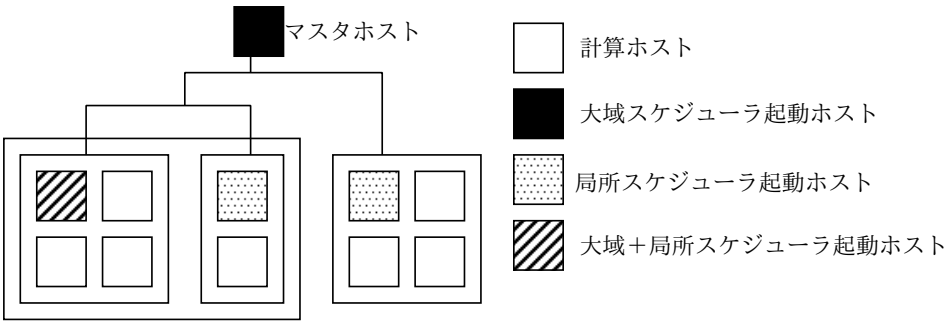

図 2.4: スケジューラの起動ホスト

最下層のホストグループを対象とする局所スケジューラは，台数が少なく均質環境であるので，従来のように精密なスケジューリング手法が利用できる．これに対し，上位層の大域スケジューラは，直下のホストグループに対するタスクの分配のみ担当し，個々のタスクやホストに関する情報は考慮しないため，高速なスケジューリングが可能である．結果として静的スケジューリングの精度は低下するが，もともとスケジューリング情報が不完全であるため，不必要に高精度なスケジューリング手法に計算時間を費やすよりも，その時間を実行開始後に動的な補正に費やす方が，短いスケジューリング長を期待できる．また，2.2.1 節で述べたように，ワークフローシステムの動的スケジューリングでは，タスク間の依存関係や通信ボトルネックを考慮する必要がある．そこで本研究では，まず大規模なワークフローにおいて高速にスケジューリングすることが可能な静的スケジューリング手法を開発した．そして，動的性能変動によるスケジューリング結果の補正を行う改良手法を開発した．

2.3.2 実行環境のモデル化

大域スケジューリング手法では以下のようにして 2.2.2 で説明したような実行環境になるようにモデル化する．

(1) 処理性能が同一，かつ，相互の通信速度が同一なホスト群を，それぞれ一つのホストグループとする．あるホストに対し，条件を満たす

他のホストがない場合は，その 1 台のみで一つのホストグループとする．

(2) 相互の通信速度が閾値以上のホストグループをそれぞれ一つのホストグループとする．

(3) 全体が一つのホストグループになるまで，閾値を徐々に小さくしながら（2）を繰り返す．

(1) により，個々のクラスタは最内側のホストグループとなる．また，(2) により，ホストグループは内側ほど通信が高速になるように階層化される．

図 2.3 は 10 台のホストからなる 3 つのクラスタを階層化した例である．ホストの中の値は処理性能を，ネットワーク付近の値は通信性能を，それぞれ表す．

2.3.3 ワークフローのモデル化

大域スケジューリングでは，大量のタスクやホストを直接扱うことで生じる計算コスト増大を防ぐため，タスクのグループをホストのグループに割り当てるという抽象スケジューリングを行う．このため，2.3.1 節でホストを階層モデル化したように，タスクも以下のアルゴリズムにより，グループ階層の形でモデル化する．

(1) 入力側と出力側の両方について，タスク/タスクグループ/タスク配列が各々一つずつしか接続されていないストリームを探す．

  (a) 該当するストリームが存在した場合，その中で通信コストが最大のストリームとそれに接続されているタスク/タスクグループ/タスク配列を，まとめて新たなタスクグループとし，(1) へ戻る．

  (b) 該当するストリームが存在しない場合，(2) へ進む．

(2) 以下の条件を満たすストリームを探す．

(a) 入力側に接続されているすべてのタスク/タスクグループ/タスク配列が，同一のストリームの出力側に繋がれている．

(b) 出力側に接続されているすべてのタスク/タスクグループ/タスク配列についても，同様に同一のストリームの入力側に繋がれている．

(3) (2) を満たすストリームが存在した場合，その中で通信コストが最大のストリームとそれに接続されているタスク/タスクグループ/タスク配列を，まとめて新たなタスクグループとし，(1) へ戻る．

(4) 該当するストリームが存在しない場合，処理を終了する．

このように階層グループ化することで，ワークフロー中で通信量の多い部分ほど下位のグループになる．大域スケジューリングの際に，階層の上位からグループを分割していくことで，通信量の少ない上位のグループが，通信の遅いホストグループにまたがって割り当てられるようになり，全体として通信遅延による速度低下が抑えられる．また，一対一接続のタスクは別々のホストグループに配置しても並列性が得られないため，優先的に下位のタスクグループになるようにしている．

図 2.5 に，この階層グループ化の適用例を示す．図中でタスク内の数字は計算コストを，ストリーム横の数字は通信コストを，それぞれ表す．このネットワークに対し，まず手順 (1) により，ストリーム s4，続いてストリーム配列 s3 が選ばれ，これらがグループ化される (**i**, ii)．続いて手順 (2) に移ると，残るストリームの中で条件 (a), (b) を満たすのは s2 だけなので，手順 (3) によりこれらがグループ化される (iii)．その後，手順 (1) に戻るが，条件を満たすものがないので手順 (2) に進むと，(iii) のグループ化により s1, s5 も条件 (a), (b) を満たすようになっている．このうち通信コストの大きい s1 の方がグループ化される (iv)．その後，手順 (1) に戻ると，残る s5 が条件を満たしているので，グループ化される (v)．

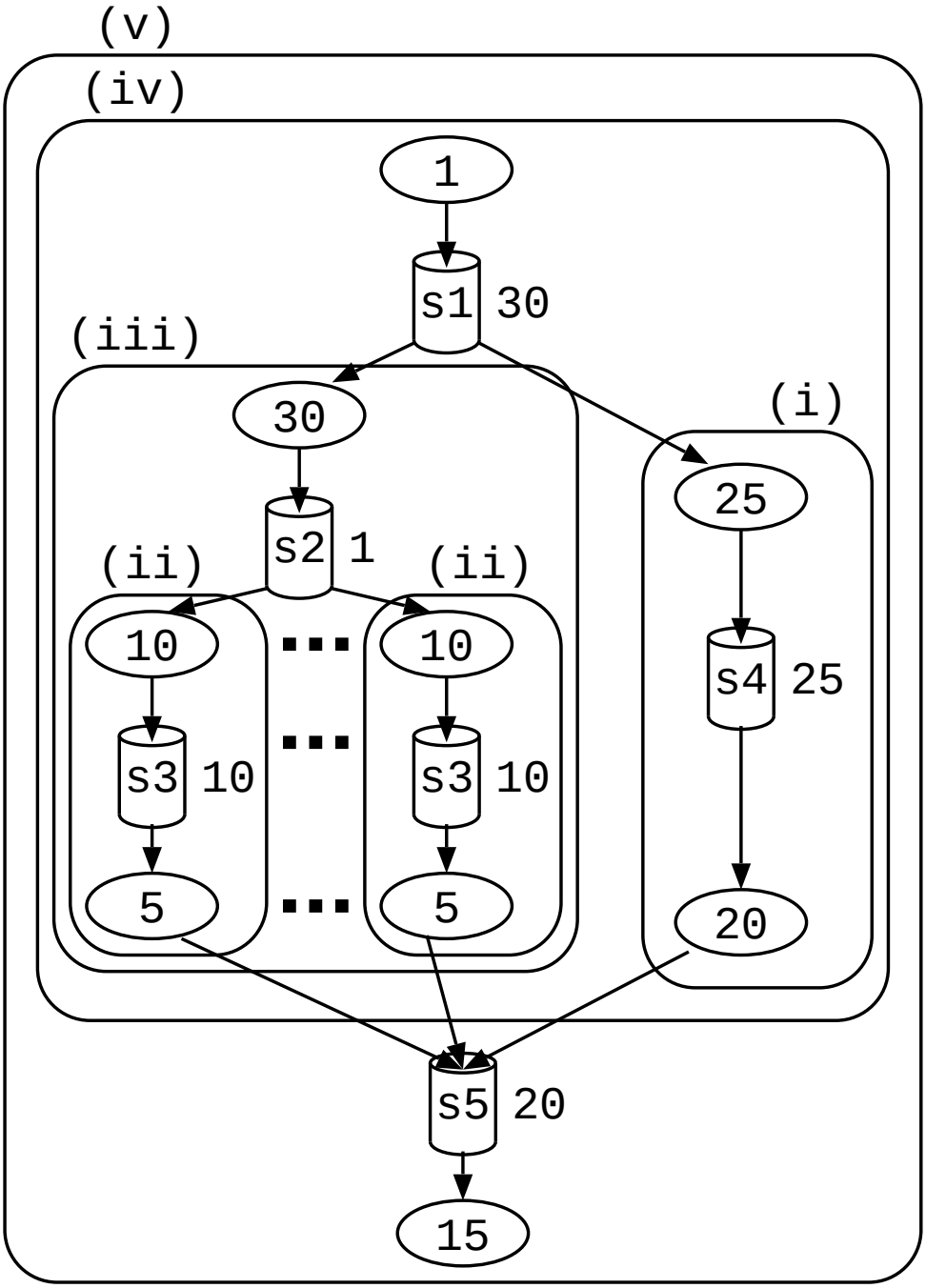

図 2.5: ワークフローの階層モデル化

2.3.4 大域スケジューリング手法

大域スケジューリングでは，タスクとホストの階層モデルを用いて，タスクグループをホストグループに割り当てていく [26].

大域スケジューラは，2.3.1 節で述べたように階層化され，各スケジューラは自身の直下にあるホストグループに対し，自身が割り当てられたタスクグループを分配する．各ホストグループには下位のスケジューラが一つずつ動作しており，同様にしてさらにタスクグループを分配していく．これを繰り返すことで，タスクは段階的に細かいホスト群へと分配されていき，最終的に最下位の局所スケジューラによって，個々のホストに配置される．

本手法では，大域スケジューラはホストの計算・通信性能を個別に扱わず，自身のタスク分配作業の対象となる，直下のホストグループの計算性能 (グループ内ホストの計算性能の合計) のみ考慮する．また，タスクについても階層モデルを用いることで，分配に必要な上位層のタスクグループとその計算コストだけを用いる．このため，大規模な並列処理を対象としても，非常に高速なスケジューリングが可能である．さらに，複数のスケジューラによる段階的スケジューリングにより，スケジューリングを行うホストの性能やメモリがボトルネックになることも避けられる．

ホストグループの計算性能およびタスクグループの計算コストは，それぞれ含まれるホストおよびタスクの計算性能・計算コストの合計である．したがって，計算負荷の均等化という観点からは，ホストグループの計算性能に計算コストが比例するように，タスクグループを分配すればよい．しかし，タスク間には依存や通信量の偏りが存在するので，計算コストのみ考慮したのでは，アイドル時間や通信ボトルネックが生じてしまう．

2.3.3 項で述べたアルゴリズムにより，一つのタスクグループは一つのストリームとその入出力端に接続されたタスク／タスク配列／タスクグループで構成されている．そこで，このストリームの入力側・出力側の

タスク群それぞれについて上記の比例配分を行うことにより，各ホストグループで前者のタスク群の実行が終了し後者の実行に移るタイミングを揃えることができる．実際には計算量を完全に比例配分できなかったり，不確定要因による誤差が発生したりするが，この方法を用いることによって，非常に小さいスケジューリングコストでタスクの依存によるアイドル時間の期待値を最小化できる．また，タスクの階層モデル化の段階で通信量の多いストリームが下層になるようにしているため，通信ボトルネックは自然とホスト内や下位のホストグループ内に閉じ込められる．

一つの大域スケジューラは，ホストグループの集合 $H = \{H_1, \ldots, H_n\}$ に対し，タスクグループ T をスケジューリングする．ここで，$P(H_i), N(H_i)$ をそれぞれ，ホストグループ H_i の計算性能 (そのホストグループに含まれるホストの計算性能の合計)，H_i に含まれるホスト数とする．H_i に含まれるホストの平均計算性能は，$P_{avg}(H_i) = P(H_i)/N(H_i)$ である．H に含まれるすべてのホストの計算性能の合計は，$P(H) = P(H_1)+\ldots+P(H_n)$ で表記する．また，タスクグループ T は，ストリーム S，その入力側に接続されたタスク／タスク配列／タスクグループ $I = \{I_1, \ldots, I_l\}$，および，出力側に接続されたタスク／タスク配列／タスクグループ $O = \{O_1, \ldots, O_m\}$ で構成されるとする．入力側および出力側タスクの計算コストの合計値を，$C(I) = C(I_1) + \ldots + C(I_l)$，$C(O) = C(O_1) + \ldots + C(O_m)$ とする．

これらを用いたスケジューリングアルゴリズムを以下に述べる．

1. タスク未分配のホストグループのうち，ホストの平均計算性能 $P_{ave}(H_i)$ が最大のものを選び，これを次の分配対象とする．

2. 入力／出力側タスクのそれぞれについて，H_i に割り振るべき計算コストを以下のように求める．

$$\begin{aligned} C_{in}(H_i) &= C(I) \times P(H_i)/P(H) \\ C_{out}(H_i) &= C(O) \times P(H_i)/P(H) \end{aligned}$$

3. 入力側タスクを H_i に分配する．

(a) I から計算コストが最大である要素を取り出す (I_j とする).

(b) $C(I_j) < C_{in}(H_i) - L$ の場合，I_j を H_i に割り当て，$C_{in}(H_i) \leftarrow C_{in}(H_i) - C(I_j)$ として，(3)(a) に戻る．ここで，L は閾値として用いる微小な正数である.

(c) $C_{in}(H_i) - L \leq C(I_j) \leq C_{in}(H_i) + L$ の場合，I_j を H_i に割り当て，(4) に進む.

(d) $C_{in}(H_i) + L < C(I_j)$ の場合,

i. I_j がタスクならば，I_j はこれ以上分割できないので，I_j を H_i に割り当て，(4) に進む.

ii. I_j がタスク配列ならば，その要素であるタスクの計算コスト合計値が $C_{in}(H_i)$ にもっとも近い位置で I_j を分割して割り当て，(4) に進む.

iii. I_j がタスクグループならば，I_j に対して再帰的に (2) から繰り返す．これを，(3)(d)(i) によりそれ以上分割できなくなるか，H_i に分配された総計算コストが $C_{in}(H_i) \pm L$ の範囲に収まるまで再帰的に繰り返す．その後，(4) に進む.

4. 出力側タスクについても，同様にして H_i に分配し，(1) に戻る.

平均性能が大きいホストグループや計算コストの大きいタスクグループから順に処理を行うのは，計算量の大きいタスクを性能の高いホストを含むホストグループに配置し，各ホスト上の実行時間を均等にしやすくするためである.

図 2.3 に示した階層モデル化したホスト群と図 2.5 の階層グループ化したワークフローに対し，上記アルゴリズムを適用する場合を例に説明する．最上位スケジューラには，直下のホストグループとして $HG3$, $HG4$ が与えられる．各ホストグループの性能として，以下の値が計算できる.

$$
\begin{aligned}
P(HG3) &= 0.8 \times 4 = 3.2 \\
P(HG4) &= 1.0 \times 4 + 1.2 \times 2 = 6.4
\end{aligned}
$$

$$
\begin{aligned}
P_{avg}(HG3) &= 0.8 \\
P_{avg}(HG4) &= 6.4/6 \approx 1.07
\end{aligned}
$$

$P_{avg}(HG4) > P_{avg}(HG3)$ なので，(1) より，まず HG4 を分配対象とする．最外側のタスクグループ (v) に注目すると，(2) で入力側タスクはタスクグループ (iv)，出力側タスクはコスト 15 のタスク一つである．仮にタスクグループ (ii) の配列の要素数を 4 とすると，

$$
\begin{aligned}
C_{in}(HG4) &= (1 + 30 + (10 + 5) \times 4 + 25 + 20) \\
&\quad \times 6.4/9.6 \\
C_{out}(HG4) &= 15 \times 6.4/9.6
\end{aligned}
$$

となり，タスクグループ (iv) のうち 2/3 の計算コストの部分を割り当てる必要がある．そこでタスクグループ (iv) に対し同じ手順を適用すると，入力側タスクはコスト 1 のタスク一つであり，出力側タスクはタスクグループ (**i**), (iii) である．前者についてはこれ以上分割できないので，HG4 に割り当てられる．後者については (3)(a) により，まずタスクグループ (iii) が選択される．(iii) のコストはちょうど (**i**) と (iii) のコストの和の 2/3 なので，(iii) が HG4 に割り当てられ，(**i**) が HG3 に割り当てられる．最後に，最初に注目したタスクグループ (v) の出力側タスクの割り当てに戻ると，これはコスト 15 のタスク一つしかないため，HG4 に割り当てられる．以上の流れにより，図 2.6(a) のように分割される．もしタスクグループ (ii) の配列の要素数が 4 より大きければ，タスクグループ (iii) がさらに分割され，図 2.6(b) のように (ii) の一部が HG3 に割り当てられる．

2.3.5 局所スケジューリング手法

大域スケジューリング手法によって扱うタスク数が減少するため，局所スケジューラでは従来の静的スケジューリング手法を利用しやすくなる．しかし，高精度な静的スケジューリング手法は計算コストが非常に高い．また，動的性能変動を考慮した動的再スケジューリング手法を利

用する場合，その計算コストはさらに高くなりこれが局所スケジューラがボトルネックになる．従って，動的性能変動が生じる環境で大規模なワークフローを効率よく実行するためには，高速な動的再スケジューリング手法が必須となる．

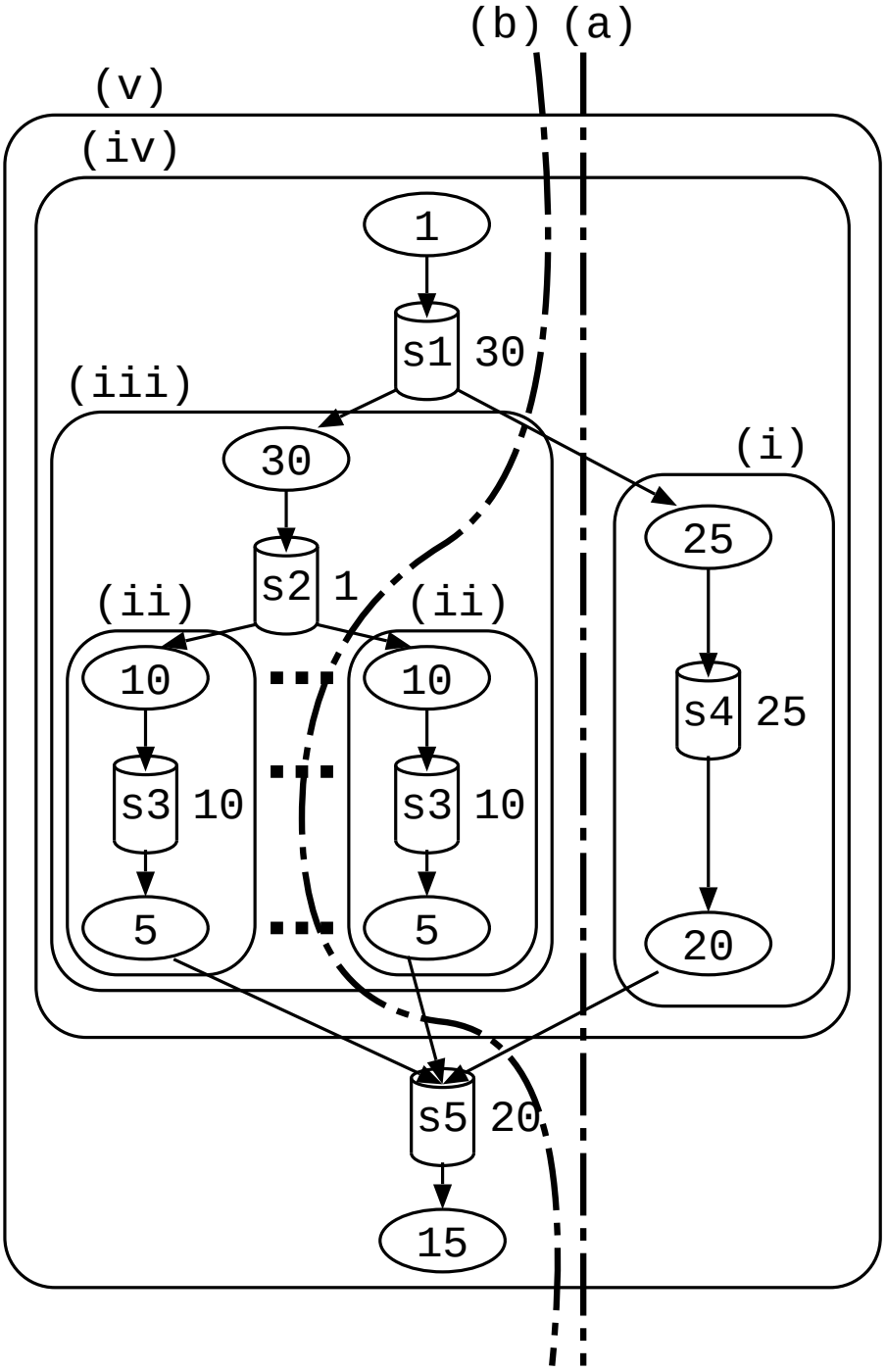

図 2.6: 大域スケジューリングの例

3　静的スケジューリング手法の高速化

3.1　はじめに

大規模なワークフロー型の並列処理で高いスループットを得るには，実行単位となる各タスクをどの計算機でどの順番で行うかというタスクスケジューリングが重要になる．そのため様々なタスクスケジューリング手法が提案されている．HEFT[9] や LDBS[10] のような静的スケジューリング手法は，ワークフロー型並列処理全体の実行時間であるスケジューリング長において短い結果を得ることができる．しかし，大規模なワークフローを想定した場合，計算量が膨大になり現実的な時間で解くことが難しい．

そこで，複数のスケジューラを階層的に配置し，上位層の大域スケジューラと下位層の局所スケジューラで異なるスケジューリング方法を使い分ける階層型スケジューリング手法を提案している (2.3 節)．これによりスケジューリング長を維持しつつ計算コストを大幅に削減することができる．それでも，下位層に従来のスケジューリング手法を用いた場合，10,000 タスク/1,000 ホストの条件において下位層のスケジューリング処理が全体のスケジューリング処理の 98% 以上を占めボトルネックとなる．そこで，下位層のスケジューラで扱うタスクの依存関係に着目した適応型スケジューリング手法 (AS) を提案する．これはワークフロー内のタスクの依存関係に着目し適応的にスケジューリング手法を切り替える方法で，ワークフロー内にパラメータスウィープのような互いに依存関係を持たないタスクの集合を多数含んでいた場合，スケジューリングの計算時間を大幅に削減することができる．適応型スケジューリング手法を大規模なワークフロー型の並列処理を目的とするタスク並列スクリプト言語 MegaScript に本手法を実装し抽象シミュレーションで評価を行った結果，スケジューリング時間を 1/50 ∼ 1/100 程度に削減できた．

しかし BLAST を用いた遺伝子解析のワークフローのように類似性の高いサブワークフローが複数含まれている場合，適応型スケジューリン

グ手法のアルゴリズムでは処理の切り替えを効率的に行うことができず，スケジューリングの計算時間の大幅な削減が行えない可能性がある．そこで適応型スケジューリング手法を改良し，スケジューリング処理の適応的な切り替えを再帰的に行う再帰的適応型スケジューリング手法 (RAS) を提案する．RAS はサブワークフローを一個の疑似タスクと見なして適応型スケジューリング手法でスケジューリングを行う．そして疑似タスクをスケジューリングする段階でサブワークフローに展開し，サブワークフロー内のタスクを適応型スケジューリング手法を用いてスケジューリングする．このように再帰的に適応型スケジューリング手法を実行することで，一度の適応型スケジューリング手法の実行で扱うタスク数を減らし，またサブワークフローを疑似タスクに見なし効果的に処理の切り替えを発生させることで，計算時間の削減を行う．この RAS を実装し抽象シミュレーションで評価を行った結果 AS と比較して，10,000 タスク規模でスケジューリングの計算時間を 1/100 程度に削減できた．

3.2 従来の静的スケジューリング手法

3.2.1 DAG スケジューリング手法

DAG の静的スケジューリング手法は多くの研究がされており，その計算コストは高いが短いスケジューリング長を得ることができる．Heterogenous Earliest-Finish-Time(HEFT) [9] や Levelized Duplication Based Scheduling(LDBS) [10] がある．HEFT では，予め求めた優先度に従ってタスクの配置を行う．割り当てるホストの決定には，対象となるタスクの実行終了時間を用いる．HEFT の計算オーダーは，ホスト数とタスク数をそれぞれ m と t とおいた場合，$O(t^2m)$ である．DAG を扱うオフラインスケジューラとしては高速であるが，本手法で扱うような大規模な並列処理で利用するには，計算コストが非常に高くなるため，そのまま使用することは難しい．LDBS では，タスク間のデータ通信時間を減らすために，同一タスクの複数ホストへの割り当てを試みる．LDBS はデータ通信時間の比重が高くなるほど，スケジューリング長が短くなる傾向がある．ホスト数を m，タスク数を t，全てのタスク間通信の回数を e とした場合，計算オーダーは $O(t^3em^2)$ であり計算コストが非常に高い．そのため本手法で扱うような大規模な並列処理で使用することは難しい．また，同一のタスクを複数回実行する可能性があるので，全てのタスクに対して重複実行が認められている必要がある．

大規模ワークフローのタスクは基本的に静的スケジューリングが可能であるが，スケジューリングにかかる計算コストが高くなってしまい，現実時間でスケジューリングを行うことが困難である．従って，計算時間を削減することが必須となる．

3.2.2 独立スケジューリング手法

各タスク間に依存関係が無いものを対象とした，独立タスクスケジューリングの研究も盛んである．本節ではいくつかの既存の独立タスクスケジューリング手法を紹介する．

スケジューリング対象のタスクを，実行完了時間が最小となるホストに順次配置していく手法に，Min-Min ヒューリスティック法 [16] がある．これはタスクの実行時間が計算量や実行環境に比例しない $unrelated$ な問題に対応した手法である．複数の独立タスクスケジューリング手法と比較した結果，スケジューリングにかかる計算コストが低く，比較的短いスケジューリング結果を導き出している [27].

より計算コストが低いスケジューリング手法としては OLB や UDA があるが [15]，これらのスケジューリング長は Min-Min ヒューリスティックと比較すると長くなる [27]．遺伝的アルゴリズムを使ったスケジューリング手法 [28] では，Min-Min ヒューリスティックと比較すると短いスケジューリング結果を得られるが，計算コストが 10 倍以上と非常に高い結果になっている [27].

ワークフロー型の大規模並列処理のタスクスケジューリングは，各タスク間の依存関係を考慮する必要があるが，ワークフローに含まれる独立型タスク群に対して，上記の手法を部分的に適用させることができれば，より効率的なスケジューリングが行える．

3.3 適応型スケジューリング手法

3.3.1 手法の概略

2.2.1 節で示したようにワークフロー中には多くの独立型タスク群を含むと考えられる．したがって，各クラスタに分配されるタスクの多くも独立型タスク群の一部である可能性は高い．そこで各独立型タスク群を疑似タスクに縮退し，従来の DAG のタスクスケジューリング手法を利用する．本論文の局所スケジューラでは，DAG のタスクスケジューリング手法として比較的スケジューリング結果が短く，スケジューリング時間の短い HEFT を用いた．

2.2.1 節で述べたように，独立型タスク群は互いに依存関係は無い．したがって，このタスク群をスケジューリングする際，DAG タスクスケジューリング手法ではなく，独立タスクスケジューリング手法を利用することができる．局所スケジューリングは各クラスタ毎に実行されるので，タスクの実行時間が計算量や実行環境に比例する $uniform$ な問題とみなせる．従って今回は 3.2.2 節で紹介した MinMin ヒューリスティック法を簡略化し利用した．

3.3.2 独立型タスク群

本節では独立型タスク群の定義を示す．

- 独立型タスク群を構成するタスクは互いに依存関係を持たない．
- 独立型タスク群を構成する各タスクに対して，直接依存するタスクの集合は全て同じである．
- 独立型タスク群を構成する各タスクに対して，直接依存されるタスクの集合は全て同じである．

図 3.7 は独立型タスク群の例である．図中の (a)，(b) はそれぞれ独立型タスク群である．しかし (c) は，(1)〜(3) と (4)〜(6) において，直接依存

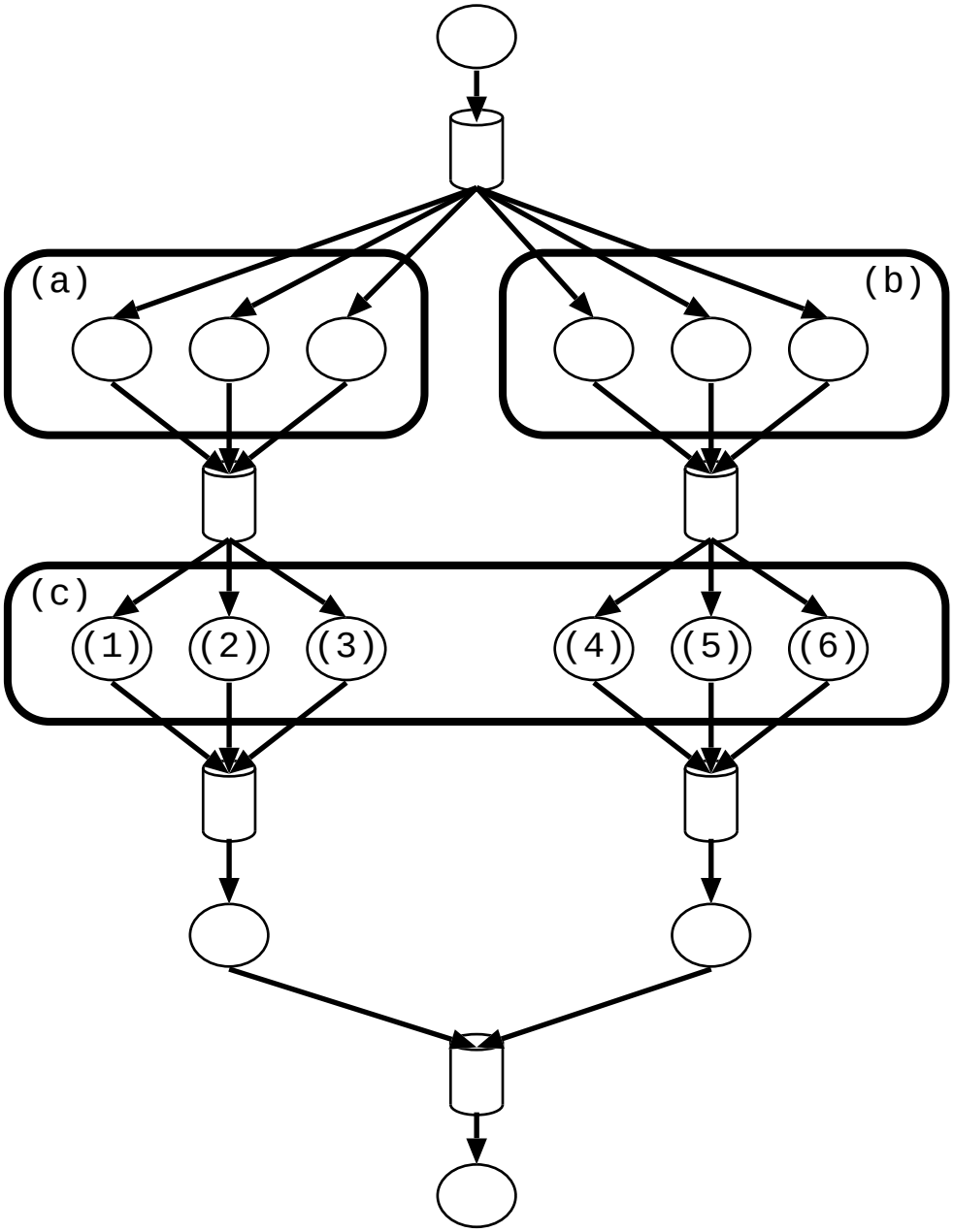

図 3.7: 独立型タスク群の例

するタスクや直接依存されるタスクの集合が異なるため，独立型タスク群ではない.

3.3.3 定義

本節では適応型スケジューリング手法で用いる値や関数を定義する．$w_{i,p}$ はタスク T_i をホスト H_p で実行した時の実行時間，$c_{i,j}$ はタスク T_i からタスク T_j へのデータ通信時間である．$\overline{w_i}$ はタスク T_i を各ホストで実行させた時の平均実行時間，$\overline{c_{i,j}}$ はタスク T_i から T_j への平均データ通信時間を表している．$available(H_p)$ はホスト H_p で新たにタスクを実行できるようになる時刻である．$pred(T_i)$ はタスク T_i が直接依存しているタスクの集合，$succ(T_i)$ はタスク T_i に直接依存するタスクの集合を表す．$host(T_i)$ はタスク T_i が実行されるホストを示す．

3.3.4 HEFT

本手法の局所スケジューラでは，DAG のタスクスケジューリング手法として HEFT を利用している．本節では HEFT のアルゴリズムについて詳細を紹介する．

HEFT では $rank_u(T_i)$， $EST(T_i, H_p)$，$EFT(T_i, H_p)$ の三つの関数を使いスケジューリングを行う．$rank_u(T_i)$ はタスク T_i の実行開始から全タスクの実行終了までの予想時間であり，次式で表される．

$$\begin{aligned} rank_u(T_i) &= \overline{w_i} + \\ &\quad \max_{T_j \in succ(T_i)} (\overline{c_{i,j}} + rank_u(T_j)) \end{aligned} \tag{1}$$

これは各タスクのスケジューリングされる順番を求めるために使われる．$EST(T_i, H_p)$ はタスク T_i がホスト H_p で実行可能となる時刻で，タスク T_i の依存関係やホスト H_p のスケジューリング状況を考慮して求められる．$EFT(T_i, H_p)$ は $EST(T_i, H_p)$ に $w_{i,p}$ を加えた時刻で，タスク T_i をホスト H_p で実行した時の実行終了時刻になる．$EST(T_i, H_p)$ と $EFT(T_i, H_p)$ は次式で表される．

$$\begin{aligned} EST(T_i, H_p) &= \max(available(H_p), \\ &\quad \max_{T_j \in pred(T_i)} (EFT(T_j, host(T_j)) \end{aligned}$$

$$+c_{j,i}))\tag{2}$$

$$EFT(T_i, H_p) = w_{i,p} + EST(T_i, H_p)\tag{3}$$

これらは各タスクをどのホストで実行するか決定するために使われる．

HEFT のアルゴリズムは以下のようになる．

1) リストに各タスクを格納

2) 各タスク T_i の $rank_u(T_i)$ を算出

3) リストのタスクを $rank_u$ の降順でソート

4) リストが空の場合処理を終了

5) リストの先頭のタスク T_i を取得

6) リストからタスク T_i を除去

7) $EFT(T_i, H_p)$ が最小となるホスト H_p を選択

8) タスク T_i をホスト H_p に割り当て

9) 処理 (4) へ戻る

3.3.5 MinMin ヒューリスティック

3.3.1 節で示した理由により，$uniform$ な問題を想定した MinMin ヒューリスティックを示す．

MinMin ヒューリスティックは $CT(T_i, H_p)$ 関数を利用する．これはタスク T_i をホスト H_p で実行した時の実行終了時刻を示す関数であり，以下の式となる．

$$CT(T_i, H_p) = available(H_p) + w_{i,p}\tag{4}$$

この関数は各タスクをどのホストで実行するか決定するために使われる．

まずオリジナルのMinMinヒューリスティックについて説明する．未割り当ての各タスクT_iについてCTが最小値を取るホストを探し，その最小値を$MCT(T_i)$とする．各タスクのMCTの中でさらに最小値を探す．MCTが最小値を取るタスクを該当のホストへ割り当てる．以上の操作を全てのタスクがスケジューリングされるまで繰り返す．従ってオリジナルのMinMinヒューリスティックの計算オーダーは，タスク数とホスト数をtとmと置いた時$O(t^2m)$で表される．

$uniform$な問題に限定した場合，MCTが最小となるタスクT_iは未割当のタスク群の中で計算のコストが最小となるタスクと一致する．従って，予め計算のコストでソートすることで，MinMinヒューリスティックの簡略化を行える．以下に本手法で用いた簡略化したMinMinヒューリスティックを示す．

a) リストに各タスクを格納

b) リストのタスクを計算量の昇順にソート

c) リストが空の場合処理を終了

d) リストの先頭のタスクT_iを取得

e) リストからタスクT_iを除去

f) $CT(T_i, H_p)$が最小となるホストH_pを選択

g) タスクT_iをホストH_pへ割り当て

h) 処理(c)へ戻る

従って，本手法で用いる簡略化したMinMinヒューリスティックの計算オーダーは$O(tm)$となり，独立型タスク群に対してこの手法でスケジューリングした場合，スケジューリング処理の高速化が期待できる．

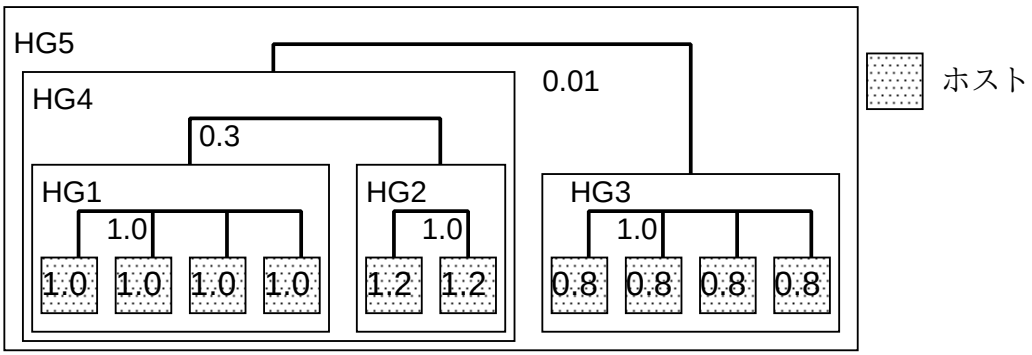

図 3.8: AS における実行環境の例

3.3.6 アルゴリズム

局所スケジューリングで適応型スケジューリング手法を利用した場合の処理について，以下に示す．

i) 各々の独立型タスク群を縮退し疑似タスクに置換する．

ii) 3.3.4 節で紹介した HEFT でスケジューリングを行う．HEFT の処理 (5) でタスク T_i が疑似タスクであった場合，処理 (6) を実行させた後で HEFT を一時停止させる．

iii) 疑似タスク T_i を展開し，展開したタスクに対して 3.3.5 節で示した MinMin ヒューリスティックでスケジューリングを行う．展開したタスクのスケジューリングが全て完了したら，HEFT の処理 (4) から処理を再開する．

疑似タスクの計算量/通信量は，縮退前の独立型タスク群に含まれる各タスクの計算量/通信量の平均値を用いる．疑似タスクの EFT は，縮退前の独立型タスク群に含まれる各タスクの EFT の最大値とする．

ここで図 3.9 のワークフローを図 3.8 の HG2 で実行した時の局所スケジューラの動作を，適応型スケジューリング手法の例として説明する．

2.3 節で説明したように，図 3.9 のワークフローを図 3.8 の環境で大域スケジューリングした時，図 3.9 の点線（A）と点線（B）に挟まれたタ

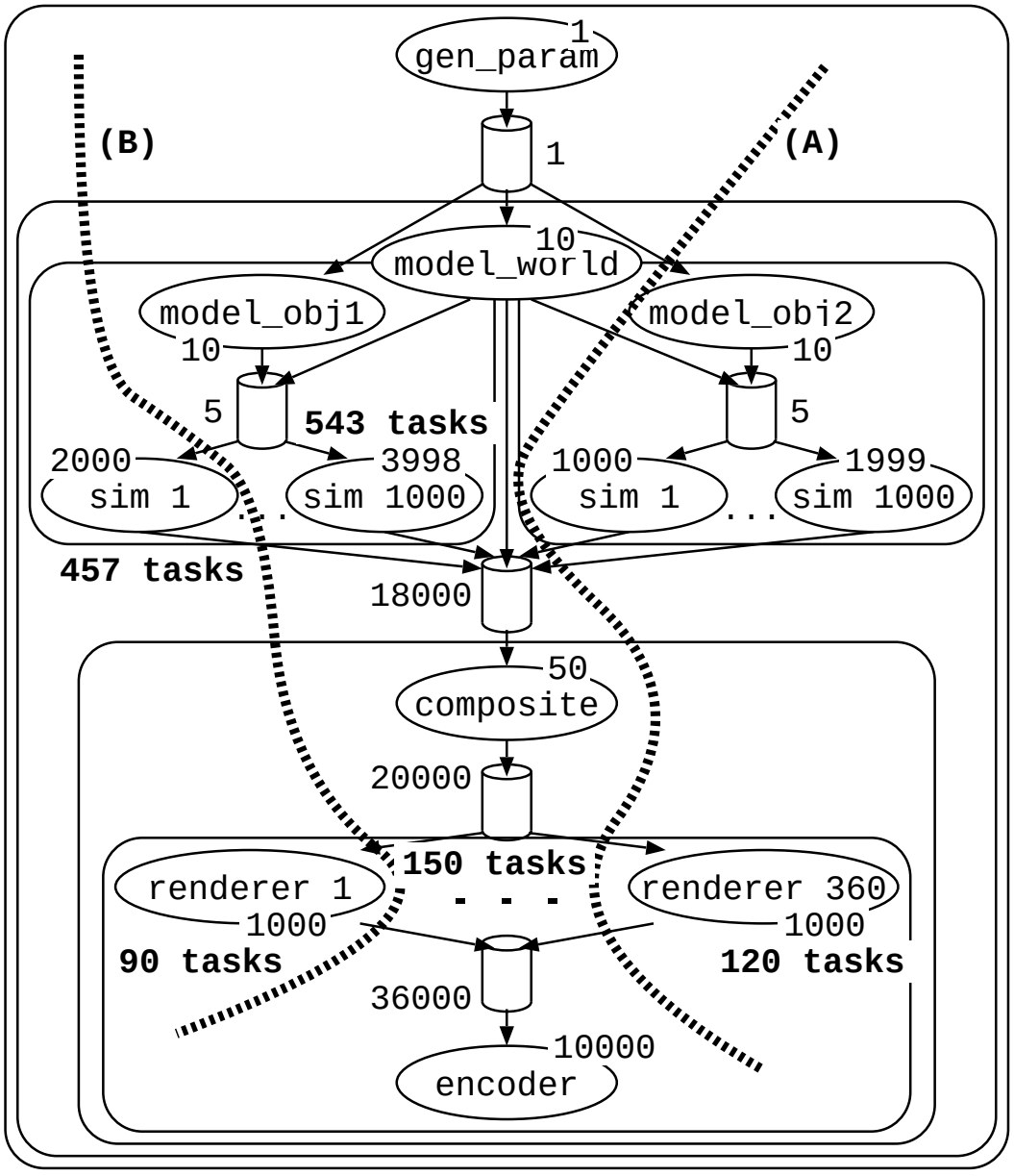

図 3.9: AS における大域スケジューリングの例

スク群が HG2 に割り当てられる．

図 3.10 の左は，独立型タスク群 sim と $render$ を縮退し疑似タスクへ置換した後のワークフローである．各タスクの値は各タスクの $rank_u$ で，

網掛けのタスクは疑似タスクを示している．

HEFT を実行させ 2 台のホストへスケジューリングを行う．図 3.10 の右図の（a）は，gen_param，model_obj1，model_world のタスク（図 3.10 の左図の点線（a））までのスケジューリングが完了した状態を示している．次にスケジューリングを行うタスク *sim* は疑似タスクであるため，HEFT を一時停止させて MinMin ヒューリスティックの処理を開始させる．展開した 543 個のタスク sim を，MinMin ヒューリスティックでスケジューリングする．図 3.10 の右図の点線（b）までのスケジューリング結果は，図 3.10 の左図の（b）である．全ての sim のスケジューリングが完了したら HEFT を再開させ，次のタスクである composit をスケジューリングする．

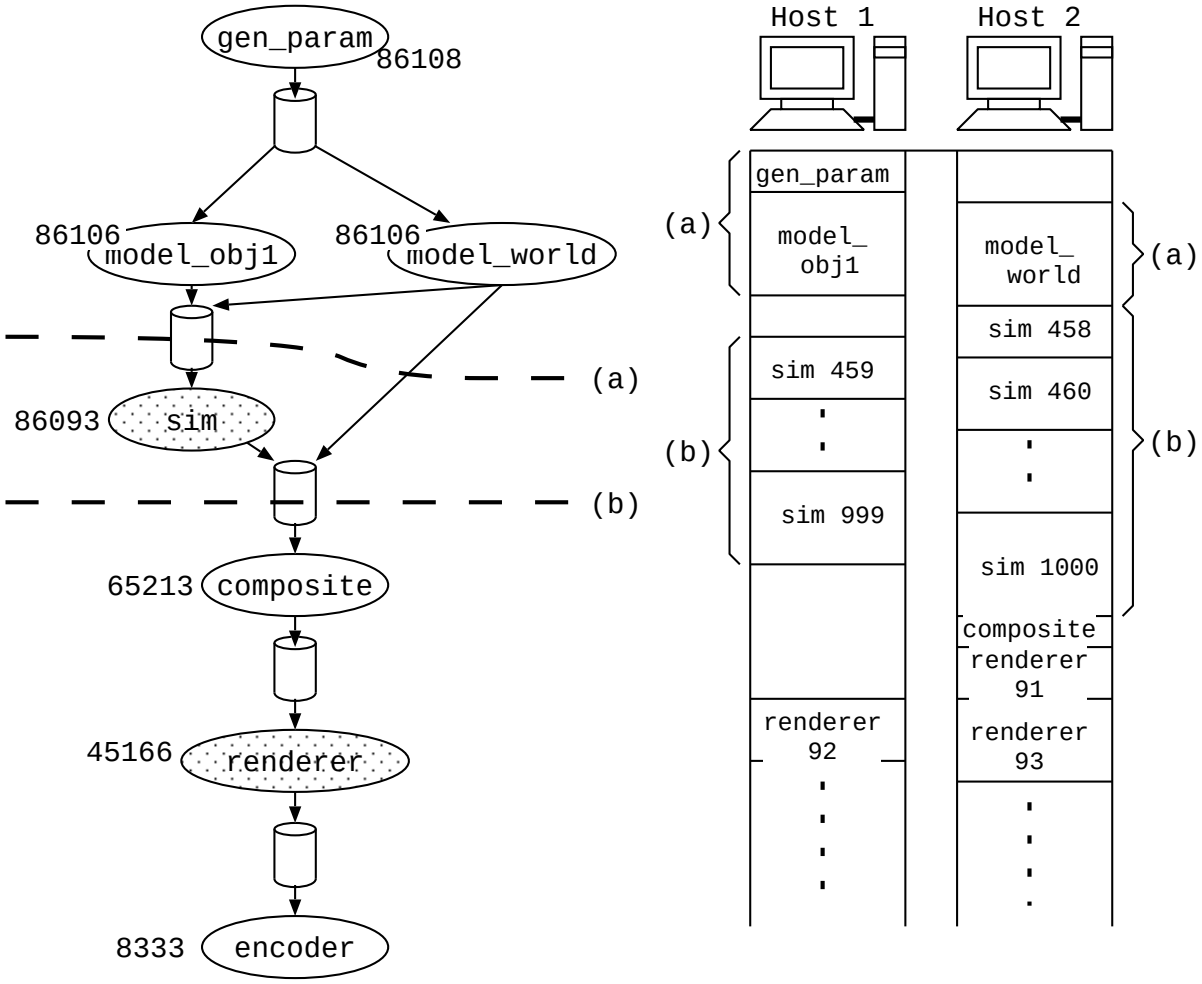

図 3.10: 適応型スケジューリング手法の例

3.3.7　シミュレーション評価の方法

本評価では実際にタスクの実行を行わず，以下の条件によりイベントドリブン型の抽象シミュレーションを行う．各タスクの計算時間はそのタスクの計算量に従い，通信時間はそのタスクの通信量に従う．

- 各タスクの通信は，そのタスクの実行終了時に行う．
- 各ホストでは一度に一つのタスクしか実行せず，スケジューリングされた順序は追い越さないものとする．つまり，次に実行するべきタスクが依存関係により実行できない場合，そのホストは当該タスクが実行可能になるまでアイドル状態となる．
- 実行時間は最初のタスクの実行開始から全てのタスクが終了するまでとし，スケジューリング自体に要する時間は考えない．
- タスク間通信が同一ホスト上で行われる場合の通信時間は 0 とする．

本評価では，CPU に Intel®Xeon®Processor X3330 (2.66GHz，L2 cache 6MB)，メモリに 2GB を搭載した環境でスケジューリングを行った．

適応型スケジューリング手法の効果を確認するために，スケジューリング時間とスケジューリング性能の二つの評価を行った．ここでスケジューリング時間とは，スケジューリング処理の完了までにかかる実行時間を示す．スケジューリング性能とは，HEFT のスケジューリング長を 1 として正規化した場合の，適応型スケジューリング手法のスケジューリング長の増加率を示す．

スケジューリング時間の評価には，上記のスケジューリング環境で各スケジューラを逐次実行し，各々の消費した CPU 時間を求めた．また各スケジューラを起動させるためのオーバーヘッドは無視する．

3.3.8　シミュレーション評価における条件

本評価では以下の手順により，ランダムなワークフローを生成した．

1. 初期構造として，2 個のタスクを 1 本のストリームで接続したワークフローを生成する．

2. 以下のいずれかの操作を等確率で適用する．

   連結 ランダムにタスクまたは独立型タスク群 T_i を選択する．ただし，ワークフローの先頭のタスクは除外する．新しくタスク T_j とストリーム S_k を生成し，T_i を，T_i と T_j を S_k で接続した構造で置き換える．

   コントロール並列 ランダムにタスクまたは独立型タスク群 T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しくタスク T_j を生成し，T_i と同じ入力側・出力側ストリームを持つように接続する．

   データ並列 ランダムにタスク T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しく独立型タスク群 T_j を生成し T_i と置き換える．この時，独立型タスク群の大きさは各評価の条件に従う．

3. タスクの総数が設定値以下であれば，(2) から繰り返す．

各タスクの計算量や通信量について，表 3.1 の範囲でランダムに選んだ値を利用した．広域ネットワーク上に分散したクラスタ群を利用するにあたり，クラスタ間の通信はインターネットを利用していることが多く高性能を期待できない．従ってタスクの計算量と比較して通信量が非常に大きい場合は想定しにくい．一方，ワークフロー中のほとんどのタスクは独立型タスク群に含まれており，独立型タスク群で実行するようなパラメータサーベイではある程度の量の解析結果を出力することが一般的である．従ってタスクの計算量と比較して通信量が非常に小さい場合も想定しにくい．したがって今回は通信量が計算量の 0.5 ～ 2.0 倍になるように設定した．

実行環境については表 3.2 の条件に従ってランダムに生成した．BLAST [1] や AMBER [4] の実行時間が通常数百秒以内であり，その出力結果が数百

表 3.1: AS の評価におけるワークフローの生成条件

タスクの計算量	100 - 200
タスクの通信量	100 - 200

表 3.2: AS の評価における非均質環境の生成条件

ホストの計算性能	1.0 - 3.2
ホスト間通信性能	100
一つのクラスタのホスト台数	16, 32, 64

MB 程度であることが多い．一般的に各クラスタ内については，ギガビット・イーサネット等の高速回線で接続しており，数百 MB のデータ転送時間は数秒程度である．そこで上で設定した計算・通信量から，計算・通信の時間比が BLAST や AMBER の場合に近くなるように設定した．一方，広域ネットワークにおける通信性能はクラスタ内の通信性能より低いことが多く，数百 MB のデータ転送に数十秒から数分程度かかることが多い．また場合によってはそれ以上の時間を要することもある．よって今回の各クラスタ間の通信性能においては，クラスタ内の通信性能の 1/10 ∼ 1/1000 に設定した．またクラスタ数については，マルチクラスタ環境である InTrigger [29] のような環境を想定して設定した．

今後より広い範囲におけるデータセットで評価を行い，様々な実アプリケーションや実行環境における効果を確認していく必要がある．

3.3.9　シミュレーション評価

適応型スケジューリング手法単独での効果を明らかにするために，均質環境で HEFT と適応型スケジューリング手法の比較評価を行った．表 3.3 は，10,000 タスクからなるワークフローを 16，32 ，64 ホストのクラスタ環境へスケジューリングした結果である．またワークフロー中には，1000 タスクから成る独立型タスク群を 10 個含んでいる．表 3.3 の HEFT・適応型は，それぞれの手法におけるスケジューリング時間を示したものである．

BLAST や FASTA のように，従来から科学技術の分野では大量のデータに対する解析処理の需要はあったが，計算資源の制約から十分な処理を行うことは難しかった．しかし近年の計算機の性能向上や低価格化に伴い，より大量のデータ解析を行うことが可能になり，パラメータサーベイのパラメータの個数も増加していくことが予想される．一方ワークフロー全体で実行する独立型タスク群の集合の数は，2.2.1 節で述べたようにユーザーがワークフローを記述するため，パラメータサーベイのパラメータの個数と比較すると小さいと考えられる．したがって今回は独立型タスク群の集合の数に比べそのパラメータの個数を十分に大きい値にした．また数百から数千の DNA 情報の解析が可能な megablast が注目を集めており，シーケンスデータベースも数十に及ぶことから，ワークフロー全体のタスク数を 10,000 程度と想定した．

スケジューリング時間に関しては，HEFT 単独でスケジューリングする場合に比べて，適応型スケジューリング手法は非常に高速である．また適応型スケジューリング手法ではクラスタの台数が増えても数秒でスケジューリングが完了するため，現実的台数で構成されるクラスタで利用する場合，有効であると言える．スケジューリング性能においては，適応型スケジューリング手法は HEFT を利用した場合に比べ，スケジューリング長が 2〜5%程度増加した．適応型スケジューリング手法は，スケジューリング時間を大幅に削減するために，従来の HEFT に比べスケジューリング処理の簡略化を行っている．しかしスケジューリング時間の大幅な

表 3.3: AS におけるクラスタ台数による評価

クラスタの台数	16	32	64
HEFT(秒)	543.98	1028.5	1667.0
適応型 (秒)	0.5840	0.8738	1.4590
スケジューリング性能	1.0125	1.0286	1.0498

削減に比べスケジューリング長の増加率はわずかである．したがって，計算量があまり大きくないタスクを大量に実行するなどスケジューリング時間の影響が大きくなるケースでは，高い効果が得られる．

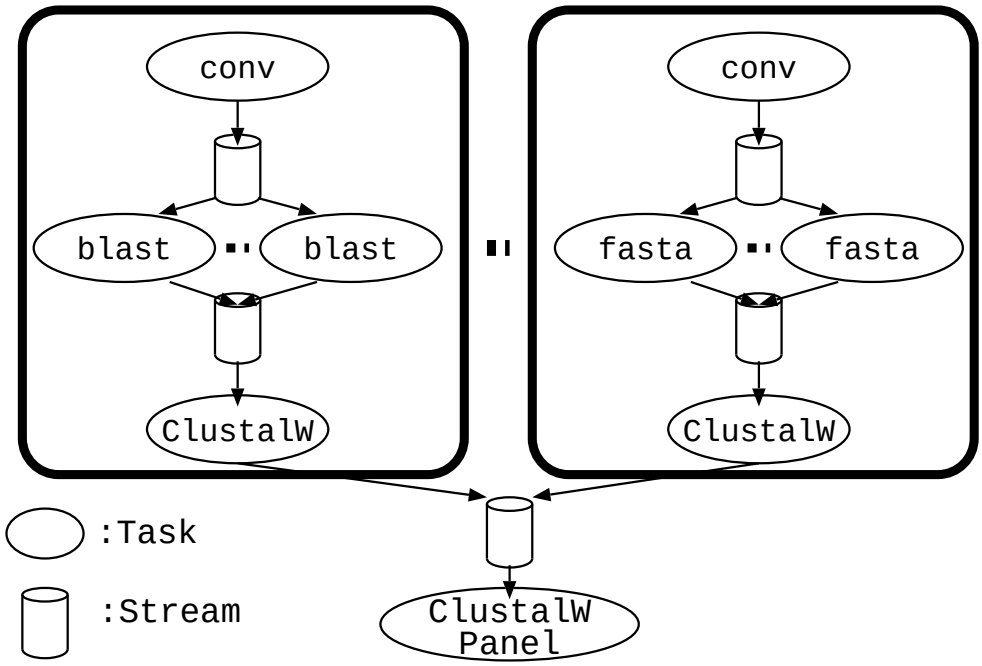

図 3.11: サブワークフローを含むワークフローの例

3.4 再帰適応型スケジューリング手法

3.4.1 手法の概要

3.2.1 節で紹介した従来の DAG のタスクスケジューリング手法はタスク間の依存関係を考慮しつつスケジューリングを行うため，短いスケジューリング長を得ることができる．しかし，想定するような大規模なワークフローで従来の DAG のタスクスケジューリング手法を用いた場合，計算量が膨大となり現実時間で解くことが難しい．そこで我々はワークフロー内の独立型タスク群に着目した適応型スケジューリング手法を提案し，計算時間を大幅に削減できていることを示している．しかし，適応型スケジューリング手法には独立型タスク群を含んでいる箇所しか効果が得られない欠点がある．従って，図 3.11 のようなサブワークフローが多数含まれているワークフローでは，*conv* や *ClustalW* のようなタスクを一度にスケジューリングしてしまうため，計算量の大幅な削減は見込めない．

そこでこの問題を解決する再帰適応型スケジューリング手法を提案する．この提案手法では初めに，ワークフロー中のサブワークフローを疑似タスクに置き換える．そして置き換えたワークフローに対して適応型スケジューリング手法でスケジューリングを行う．その際に，疑似タスクをホストへスケジューリングする時点でサブワークフローへ展開し，サブワークフロー内の各タスクを適応型スケジューリング手法でスケジューリングする．このように再帰適応型スケジューリング手法では，適応型スケジューリング手法を再帰的に実行することでスケジューリングを行う．また，図 3.11 に示すような単純なサブワークフローだけでなく，より複雑な任意のサブワークフローに対しても再帰適応型スケジューリング手法を用いることが可能である．

3.4.2 サブワークフローの検出

MegaScript ではサブワークフローを効率よく記述するために TaskNet クラス [30] が用意されている．従ってスケジューラは MegaScript 処理系から TaskNet インスタンスの情報を取得することでワークフロー内のサブワークフローを検出することができる．また実ワークフローでは図 3.11 の遺伝子解析ワークフローのように，入力データを変えてサブワークフローを実行することが多い．従って，このサブワークフローを一個の疑似タスクと見なすとパラメータスウィープとみなせる．そこで MegaScript ではこのような形を容易に記述できるようにするために，サブワークフローを一個の疑似タスクとし，それらを配列として記述できるようにしている [30].

3.4.3 アルゴリズム

再帰適応型スケジューリング手法のアルゴリズムを図 3.12–図 3.14 に示す．図 3.12 は再帰適応型スケジューリング手法の最上位関数である．再帰適応型スケジューリング手法ではワークフロー内のサブワークフローや独立型タスク群を全て疑似タスクへと置換する．MegaScript では 3.4.2 節

```
1. function Schedule(workflow)
2.     workflow1 = SubWorkflowToPseudoTask(workflow)
3.     workflow2 = TaskArrayToPseudoTask(workflow1)
4.     DagSchedule(workflow2)
5. end
```

図 3.12: RAS のアルゴリズム

で説明したように TaskNet クラスが用意されており，ユーザーがワークフロー内のサブワークフローを明示的に記述することができる．また MegaScript 処理系の内部でも TaskNet を用いたワークフローの管理がされている．*SubWorkflowToPseudoTask()* 関数ではこれを利用することで，ワークフロー内のサブワークフローを疑似タスクへと置き換える．また，ワークフロー内の独立型タスク群を *TaskArrayToPseudoTask()* 関数を用いることで疑似タスクへと置き換える．MegaScript では TaskNet も 3.4.2 節で説明したように配列として扱えるため，*TaskArrayToPseudo-Task()* 関数ではサブワークフローの配列も一個の疑似タスクへと置き換える．最後に置き換えた後のワークフローを引数に *DagSchedule()* 関数の呼び出しを行うことで，全タスクのスケジューリングを行う．

DagSchedule(workflow) 関数は依存関係を含んだ DAG のスケジューリング関数であり，ワークフロー *workflow* のスケジューリングを行う．図 3.13 は *DagSchedule()* 関数のアルゴリズムである．*DagSchedule()* 関数は最初に *workflow* に含まれる各タスクのランクを求めている．この値は各タスクの平均計算時間をベースに，ワークフローの先頭に近いタスクほど大きくなる．算出方法は HEFT [9] に従っている．次いで，このランクの値が大きいタスクから順番にスケジューリングを行う (5–16 行目)．6 行目で未スケジューリングのタスクの中でランクの高いタスク T_i を取得する．この時，タスク T_i の種類によって処理が異なる．タスク T_i が疑似タスクでなく通常のタスクの場合（13 行目），*Mapping(task)* 関数によって *task* の実行完了時刻が一番早いホストへと割り当てられる．タスク T_i がサブ

```
01. function DagSchedule(workflow)
02.   foreach(task T_i in workflow)
03.     Calculate rank of T_i and Insert T_i to list
04.   Sort list in descending order of the rank.
05.   while(list.length > 0) do
06.     task T_i = list.first
07.     if T_i is a pseudo task of the ITS I_j then
08.       tasks = each task in the ITS I_j
09.       IndepSchedule(tasks)
10.     else if T_i is a pseudo task
          corresponding to the sub-workflow N_k then
11.       DagSchedule(N_k)
12.     else
13.       Mapping(T_i)
14.     end
15.     list = list - T_i
16.   end
17. end
```

図 3.13: *DagSchedule()* アルゴリズム

ワークフローを置換した疑似タスクの場合，サブワークフローを引数に *DagSchedule()* 関数を再帰的に呼び出すことで，各タスクに対して DAG のタスクスケジューリングを行う（10–11 行目）．タスク T_i が独立型タスク群を置換した疑似タスクの場合，適応型スケジューリング手法と同様に，この疑似タスクの各タスクに対して独立タスクスケジューリングを用いてスケジューリングを行う（7–9 行目）．*IndepSchedule(tasks)* 関数は独立タスクスケジューリングを *tasks* に対して行う関数である．

図 3.14 は *IndepSchedule(tasks)* 関数のアルゴリズムである．*IndepSchedule(tasks)* 関数では，*tasks* の各要素がサブワークフローを置換した疑似タスクであるか否かによって処理が異なる．最初に各要素がサブワークフローを置換した疑似タスクであるか否かを判定し，各要素が疑似タスクの場合 *isSubWorkflow* に true を，通常のタスクである場合 false を設定

している（2–6 行目）．その後，適応型スケジューリング手法で用いている簡略化した MinMin ヒューリスティック法と同様に，計算量の小さいタスクから順にスケジューリングを行っていく（7–18 行目）．タスク T_i が通常のタスクの場合，*Mapping(task)* 関数を用いてホストへ割り当てを行う（15 行目）．そして，タスク T_i がサブワークフローを置換した疑似タスクの場合，サブワークフローを引数に *DagSchedule()* 関数を再帰的に呼び出すことで，各タスクに対して DAG のタスクスケジューリングを行う（11–13 行目）．

このように RAS は *DagSchedule()* 関数や *IndepSchedule()* 関数を再帰的に呼び出す．*DagSchedule()* 関数の計算オーダーは扱うタスク数とホスト数をそれぞれ t と m とすると $O(t^2m)$ で，その計算量は扱うタスク数の二乗に比例する．従って，サブワークフローや独立型タスク群を疑似タスクで表すことで，*DagSchedule()* 関数の計算量が大幅に削減される．また，サブワークフローの配列は個々のサブワークフローを疑似タスクとして扱い *IndepSchedule()* 関数を用いてスケジューリングを行う．*IndepSchedule()* 関数の計算オーダーは配列のサイズを n とした場合 $O(nm)$ になり，サブワークフローの配列全体を *DagSchedule()* 関数を用いた場合と比較して計算量の大幅な削減を行うことができる．以上の理由から，再帰適応型スケジューリング手法の計算量は適応型スケジューリング手法と比較して非常に小さくなる．

ここで図 3.11 のワークフローに対して再帰適応型スケジューリング手法を用いてスケジューリングした場合の例を示す．図 3.15 はスケジューリングされるタスクの順番を示している．また，図 3.16 は図 3.11 のワークフローにおけるサブワークフロー (実線枠内) や独立型タスク群 (点線枠内) を表している．再帰適応型スケジューリング手法では最初にサブワークフローや独立型タスク群を疑似タスクへと置換する (図 3.17(a))．そして，このワークフローを引数として *DagSchedule()* 関数を実行することで各タスクのスケジューリングを行う．*DagSchedule()* 関数では最初に疑似タスク (1) をスケジューリングし (図 3.15，1–10 行目)，続いて *ClustalW*

```
01. function IndepSchedule(tasks)
02.   if tasks is a ITS of sub-workflows then
03.     isSubWorkflow = true
04.   else
05.     isSubWorkflow = false
06.   end
07.   Insert each task of tasks to list
08.   Sort each list
            in ascending order of the computational cost
09.   while(list.length > 0) do
10.     task T_i = list.first
11.     if isSubWorkflow = true then
12.       workflow = the sub-workflow
                              of the pseudo task T_i
13.       DagSchedule(workflow)
14.     else
15.       Mapping(T_i)
16.     end
17.     list = list - T_i
18.   end
19. end
```

図 3.14: *IndepSchedule()* アルゴリズム

Panel タスクをスケジューリングする (図 3.15，11 行目)．疑似タスク (1) は独立型タスク群を置換した疑似タスクなので，図 3.17(b) のように展開する．そして，これらタスク群を *IndepSchedule()* 関数を用いてスケジューリングする．*IndepSchedule()* 関数でスケジューリングするタスクの順番は，簡略化した MinMin ヒューリスティックアルゴリズムに従って決定される．今回の例では，最初に疑似タスク (2) がスケジューリングされるものとする．この場合，疑似タスク (2) はサブワークフローを置換した疑似タスクなので，対応するサブワークフローを図 3.17(c) のように展開し実線枠 (2) 内のタスク群を *DagSchedule()* 関数を用いてスケジューリングす

01. Pseudo task of the dotted frame (1)
02. Pseudo task of the thick-frame (2)
03. *conv*
04. Pseudo task of the dotted frame (3)
05. *blast*
06. *blast*
07. ⋮
08. *blast*
09. *ClustalW*
10. ⋮ (Pseudo tasks of other thick-frames)
11. *ClustalW Panel*

図 3.15: スケジューリングする順番

る．(図 3.15，2–9 行目)．図 3.17(c) の実線枠 (2) の *conv* タスクは普通のタスクなので，*Mapping()* 関数を用いてホストへの配置を行う (図 3.15，3 行目)．続いて疑似タスク (3) をスケジューリングすることになる．疑似タスク (3) は独立型タスク群を置換した疑似タスクなので，対応する独立型タスク群を図 3.17(d) のように展開し，点線枠 (3) 内のタスク群を *IndepSchedule()* 関数を用いてスケジューリングする (図 3.15，4–8 行目)．このように残りのタスクに対しても *DagSchedule()* 関数と *IndepSchedule()* 関数を再帰的に呼び出すことでスケジューリングを行う (図 3.15，9–11 行目)．

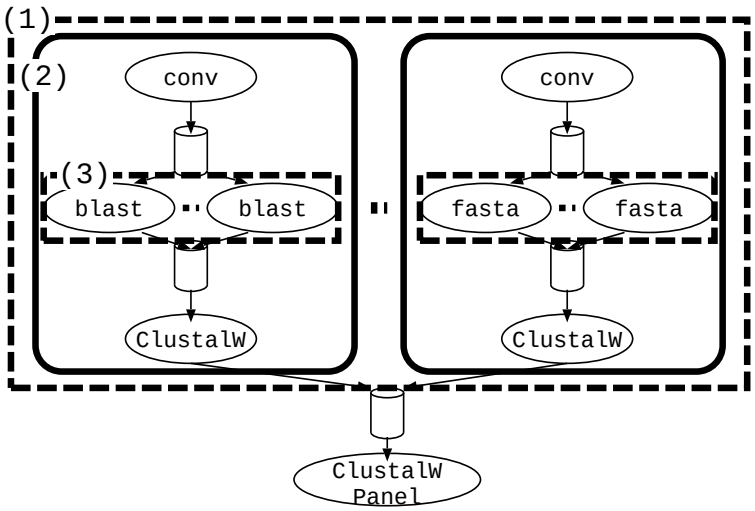

図 3.16: 置換されたワークフロー

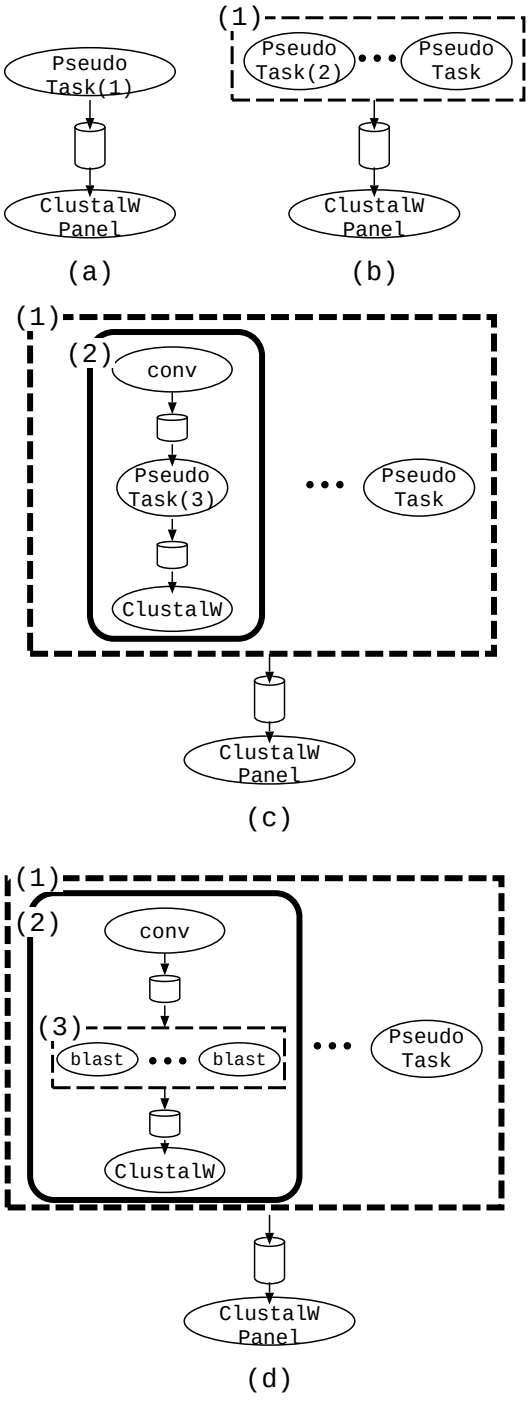
Pseudo
Task(1)
ClustalW
Panel
(a)
(1)
Pseudo
Task(2)
Pseudo
Task
ClustalW
Panel
(b)
(1)
(2)
conv
Pseudo
Task(3)
ClustalW
Pseudo
Task
ClustalW
Panel
(c)
(1)
(2)
conv
(3)
blast
blast
Pseudo
Task
ClustalW
ClustalW
Panel
(d)

図 3.17: RAS の例

3.4.4　シミュレーション評価の方法

本評価ではASを評価した際と同様に，実際にタスクの実行を行わず，イベントドリブン型の抽象シミュレーションを行った．各タスクの計算時間や通信時間はRASの評価と同様である．また，CPUにIntel®Xeon®Processor X3330 (2.66GHz，L2 cache 6MB)，メモリに2GBを搭載した環境で同様にスケジューリングを行った．

RASの効果を確認するためにASと同様に，スケジューリング時間とスケジューリング性能の二つの評価を行った．ここでスケジューリング時間とは，スケジューリング処理の完了までにかかる実行時間を示す．スケジューリング性能とは，適応型スケジューリング手法のスケジューリング長を１として正規化した場合の，RASのスケジューリング長の増加率を示す．

3.4.5　シミュレーション評価の条件

本評価では以下の手順により，ランダムなワークフローを生成した．

1. 初期構造として，2個のタスクを1本のストリームで接続したワークフローを生成する．

2. 以下のいずれかの操作を等確率で適用する．

   連結 ランダムにタスクまたは独立型タスク群T_iを選択する．ただし，ワークフローの先頭のタスクは除外する．新しくタスクT_jとストリームS_kを生成し，T_iを，T_iとT_jをS_kで接続した構造で置き換える．

   コントロール並列 ランダムにタスクまたは独立型タスク群T_iを選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しくタスクT_jを生成し，T_iと同じ入力側・出力側ストリームを持つように接続する．

データ並列 ランダムにタスク T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しく独立型タスク群 T_j を生成し T_i と置き換える．この時，独立型タスク群の大きさは各評価の条件に従う．

サブワークフロー化 ランダムに独立型タスク群 T_i を選択する．独立型タスク群 T_i をサブワークフローの配列に置換する．新しくタスク 2 個をストリームで接続したワークフローを生成し，これをサブワークフローの構造として保存する．

3. タスクの総数が設定値以下であれば，(2) から繰り返す．

RAS ではサブワークフロー内にサブワークフローを含むような，サブワークフローをネストしたワークフローに対してもスケジューリングが可能である．しかし，本実験ではサブワークフローのネストが無いワークフローを用いた．これは，ほとんどの科学技術で用いられるワークフローにはサブワークフローのネストが無いためである．

各タスクの計算量や通信量について，表 3.4 の範囲でランダムに選んだ値を利用した．タスクの通信量については各評価で用いる CCR 値 (Communication to Computation Ratio) を使い，

$$\text{タスクの通信量} = \text{タスクの基本通信量} * \text{CCR 値} \tag{5}$$

とする．これにより CCR 値が大きい場合，全体的にタスクの計算量より通信量が大きいワークフローが，CCR 値が小さい場合はその逆のワークフローが構成される．

実行環境については AS の評価と同様の理由により，表 3.5 の条件に従ってランダムに生成した．

それぞれの評価で示すデータは，同じ条件でランダムに生成した 40 種類のデータセットに対しそれぞれシミュレーションを行った平均値である．

表 3.4: RAS の評価におけるワークフローの生成条件

タスクの計算量	100 - 200
タスクの基本通信量	100 - 200

表 3.5: RAS の評価における非均質環境の生成条件

ホスト間通信性能	100
ホストの計算性能	1.0 - 5.0
ホストの台数	32

3.4.6 シミュレーション評価

RAS のスケーラビリティによる効果を明らかにするために，スケジューリング性能とスケジューリング時間による比較評価を行った．表 3.6 は，100，1,000，10,000 タスクからなるワークフローをスケジューリングした結果である．表中の AS・RAS はそれぞれの手法におけるスケジューリング時間を，増加率はスケジューリング性能を示す．ここで，CCR 値は 1.0 としている．表 3.6 の結果から，タスク数にかかわらず全ての結果で RAS のほうがスケジューリング時間が短い結果となった．またタスク数が増加するほど，スケジューリング時間の低減率は大きくなっている．これは 3.4.3 節で説明したように，RAS ではサブワークフローを一個の疑似タスクとして扱うことで各関数内で扱うタスク数を減らすことができたためと考えられる．またスケジューリング長の増加率に関しても，タスク数が 100 の時は 30%程度増加したが，タスク数が増加するほど増加率を抑えることができ，タスク数が 10,000 の時は 10%程度で済んでいる．これはタスク数が小さいほど，個々のタスクのスケジューリング結果によってスケジューリング長の影響を受け易いためであると考えられる．以上から，ワークフローのタスク数が多いほど RAS は効果があると言える．

表 3.6: RAS におけるスケーラビリティの評価

タスク数	100	1,000	10,000
AS(秒)	0.049	2.481	211.440
RAS(秒)	0.011	0.114	1.782
増加率	1.371	1.255	1.117

表 3.7: RAS における CCR 値の評価

CCR 値	0.01	0.1	1	10
AS(秒)	249.580	269.161	211.440	214.514
RAS(秒)	1.834	1.612	1.782	1.765
増加率	1.063	1.099	1.085	1.117

また，RAS の CCR 値による効果を明らかにするために，スケジューリング性能とスケジューリング時間による比較評価を行った．表 3.7 は，10,000 タスクからなるワークフローをスケジューリングした結果である．表中の AS・RAS はそれぞれの手法におけるスケジューリング時間を，増加率はスケジューリング性能を示したものである．表 3.7 の結果から，CCR 値にかかわらず RAS のスケジューリング時間は 1/150 程度になった．またスケジューリング性能に関しても，スケジューリング長の増加率を 10%程度に抑えることができた．以上から，個々のタスクの計算時間が支配的になるワークフローとタスク間のデータ通信時間が支配的になるワークフローの両方において，RAS は効果があると言える．

最後に実際のワークフローのトポロジーを用いて性能評価を行った．ワークフローには遺伝子解析で用いられる Epigenomics(図 3.18) を用いた．図中の実線枠内はサブワークフローであり，点線枠内はサブワークフローの配列である．表 3.8 に結果を示す．括弧内の数字はワークフロー

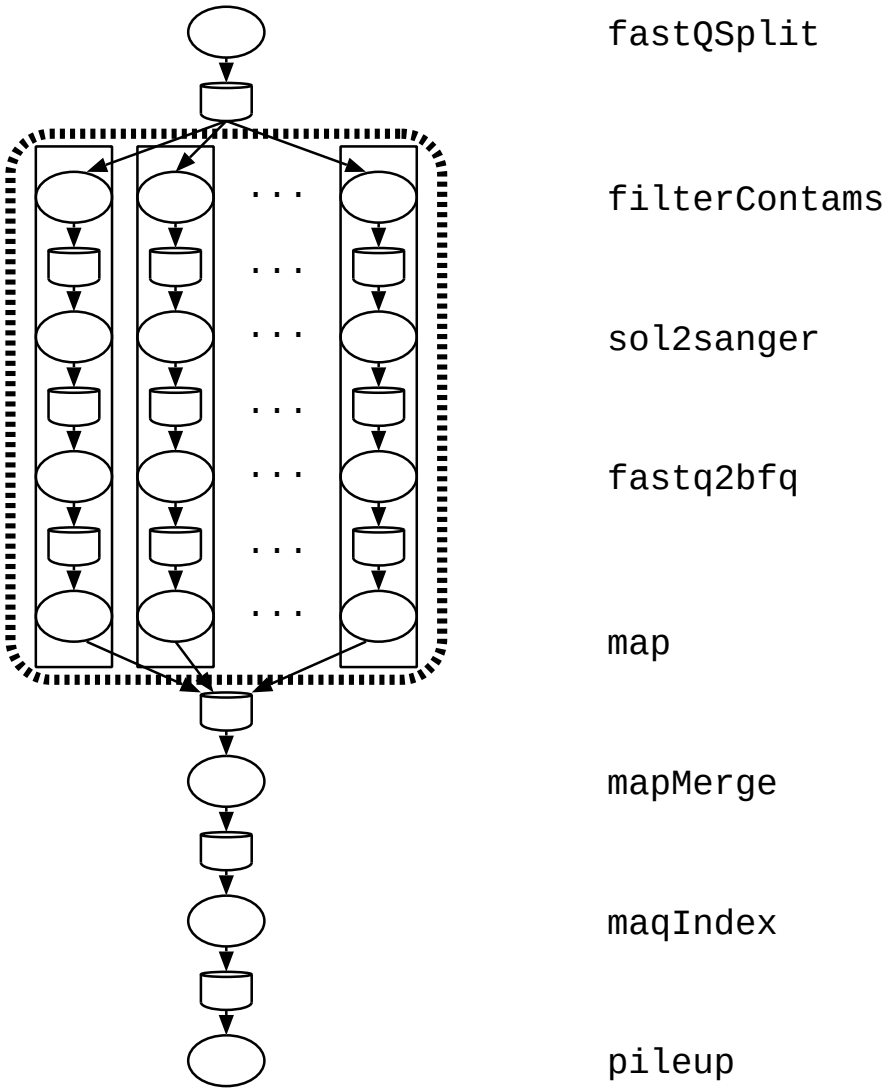

図 3.18: Epigenomics Workflow

中のサブワークフローの個数を示している．タスク数が 10,004 の時，スケジューリング長の増加率は 3%程度であったのに対しスケジューリング時間は 1/100 程度に低減された．一方，タスク数が 104 の時はスケジューリング長が 3 倍程度増加した．従って，ワークフローの規模が大きいほど RAS は効果があるといえる．ワークフローの規模が小さい時は AS のほうが効果があった．

表 3.8: RAS における Epigenomics の評価

タスク数	104 (25)	1,004 (250)	10,004 (2,500)
AS(秒)	0.029	0.762	86.626
RAS(秒)	0.003	0.042	0.569
増加率	3.199	1.277	1.029

3.5 まとめ

大規模なタスク数・ホスト数を想定した並列処理への需要が高まっている．これによりスケジューリングにおけるタスク数が大幅に増大すると考えられ，現実的な時間で従来のタスクスケジューリングを行うことは不可能である．そのため実用的なスケジューリング手法の研究においては，スケジューリング長はある程度維持しつつ，スケジューリング時間を大幅に削減することが必要になる．

そこで，タスクの依存関係に従って処理を切り替える適応型スケジューリング手法を提案した．独立型タスク群に着目して，従来手法である HEFT と Min-Min ヒューリスティックを適応的に切り替えることによって，10,000 タスク/1,000 ホストの条件ではスケジューリング時間を約 1/50〜1/100 に削減することができた．また 1 ホストで実行する平均タスク数が多いほど高い効果を得ており，特に 100 タスク/ホストではスケジューリング時間を約 1/500 に削減することができた．HEFT と適応型スケジューリング手法の単独評価においても約 1/1,000 に低減でき，非常に高い性能を得ることができた．一方抽象シミュレーションによるスケジューリング精度の評価結果により 3〜6%程度の性能低下が発生した．しかし，サブワークフローを含むワークフローでは十分な効果が期待できない可能性がある．

そこで，サブワークフローを多く含むワークフローでも高速なスケジューリングが可能な再帰的適応型スケジューリング手法を提案した．これはサブワークフローや独立型タスク群を一個の疑似タスクに置換することで計算規模を小さくし，スケジューリング時間を大幅に削減する手法である．再帰適応型スケジューリング手法を抽象シミュレーションによる評価を行った結果，10,000 タスクのワークフローではスケジューリング長の増加率を 10%程度に抑えつつ，スケジューリング時間を 1/100 程度に削減することができた．また，ワークフロー内のタスク数が多いほど，スケジューリング長の増加率を抑えつつスケジューリング時間を大幅に低減することができた．

4　動的再スケジューリング手法の高速化

4.1　はじめに

大規模なワークフロー型の並列処理で高いスループットを得るには，実行単位となる各タスクをどの計算機でどの順番で実行するかというタスクスケジューリングが重要になる．そのため様々なタスクスケジューリング手法が提案されている．HEFT[9] や LDBS[10] のような静的スケジューリング手法は，ワークフロー型並列処理全体の実行時間であるスケジューリング長を短くできる点で優れる．しかし，大規模なワークフローを想定した場合，計算量が膨大になり現実的な時間で解くことが難しい．また，静的スケジューリング手法はワークフローを実行する前に一度だけスケジューリングを行うため，他のプロセス等による実行時のホストの計算性能の変化については対応できない．性能の変化に対応している手法としては，デマンドドリブン型のスケジューリング手法が提案されているが，これはワークフローの依存関係を考慮していないため，静的スケジューリング手法と比較して実行効率が低下する恐れがある．一方，静的スケジューリング手法やデマンドドリブン型のスケジューリング手法に対して，AHEFT [19] や Blythe らの手法 [21] のような動的再スケジューリング手法が提案されている．これらの手法は，実行ホストの性能変動が発生した場合，実行ホストの新しい計算性能を用いて再スケジューリングを行う．また再スケジューリングを行う際，ワークフローを考慮する静的スケジューリング手法のアルゴリズムをスケジューリング関数として用いる．従って，動的再スケジューリング手法はワークフローの依存関係を考慮しつつ実行時のホストの性能の変化について対応できる．しかし，動的再スケジューリング手法の計算コストは静的スケジューリング手法の計算コストより高くなるという問題がある．

そこで，動的再スケジューリング手法の計算コストを削減する手法である，動的再帰的適応型スケジューリング手法 (DRAS) を提案する．DRAS では再スケジューリングアルゴリズムに RAS をベースに改良した手法を

用いている．DRAS は以下の三点のスケジューリング高速化手法を適応している．

1. スケジューリングアルゴリズムで，何度実行しても同じ結果を返す処理の省略．
2. ワークフロー実行中に不正確になるスケジューリングの内部変数を適切な値に設定．これによりワークフローを実行する時間が短くなり，それに伴って再スケジューリング回数が減少する．
3. 効果の薄い再スケジューリングを省略．

DRAS を抽象シミュレーションで評価を行った結果，改良していない RAS を用いた動的再スケジューリング手法と比較して，ワークフローの実行時間が 30%ほど短くなった．さらに，スケジューリングの計算時間を 1/6 に低減できた．

```
 1. function DynamicReschedule(T, C)
 2.   Schedule(T, C)
 3.   while (all tasks in T are not finished) do
 4.     sleep until receiving any event
 5.     if event is TaskStartedEvent then
 6.       T = T - the started task T_i
 7.     else if event is PerformanceChangedEvent then
 8.       update C
 9.       Schedule(T, C)
10.     end
11.   end
12. end
```

図 4.19: 一般的な動的再スケジューリング手法のアルゴリズム

4.2 従来の動的再スケジューリング手法

図 4.19 に一般的な動的再スケジューリング手法のアルゴリズムを示す．T はワークフロー中のタスクの集合で，C は各ホストの計算性能などのホスト情報を示す．*Schedule()* 関数はワークフロー中のタスクをスケジューリングする関数である．初めに，一般的な動的再スケジューリング手法は *Schedule()* を呼び出すことで，初期スケジューリングを行う (2 行目)．ワークフローシステムは最初，この結果に従ってタスクを配置，実行する．そして，全てのタスクが実行されるまで，動的性能変動に応じてスケジューリング結果を補正しつつ各タスクを実行する (3–11 行目)．ワークフローシステムから通知されたイベントがタスク T_i の実行を示すものだった場合，スケジューリング対象タスクを格納するリスト T からタスク T_i を除去する (5–7 行目)．ワークフローシステムから通知されたイベントが動的性能変動の発生を通知するものだった場合，C の内容を更新する (7–10 行目)．*Schedule()* は静的スケジューリングアルゴリズムで，更新された C を元に T に含まれるタスクを再スケジューリングする．

4.3 動的再帰的適応型スケジューリング手法

4.3.1 DRAS の概要

RAS は他の静的スケジューリング手法と比較して，スケジューリング時間が大幅に削減されている．しかし，動的性能変動が生じる不安定な環境下では，ワークフローの実行効率が低下する恐れがある．DRAS はそのような動的性能変動に対応した動的再スケジューリング手法の一種である．DRAS は他の静的スケジューリング手法と比較して計算コストが非常に低いことから，再スケジューリングアルゴリズムとして RAS を用いている．しかし，3.4 節で説明したアルゴリズムをそのまま利用した場合，RAS はワークフローの実行中に呼び出されることを想定していないため，十分な性能を発揮できない可能性がある．従って，RAS のアルゴリズムの改良手法について述べる．

DRAS は動的性能変動が発生した時に未実行タスクに対して再スケジューリングを行う．DRAS は本章で説明する改良した RAS を用いて再スケジューリングを行う．これは，より短いスケジューリング結果を得つつスケジューリングの計算時間を削減することを目的としている．

4.3.2 実行モデル

本節では DRAS の実行モデルについて述べる．図 4.20 は 4 台のホストからなる環境の実行モデルである．DRAS の実行モデルでは，各計算ホストでランタイムが起動しており (図 4.20 の `Runtime`)，またランタイムとは別に一個のスケジューラが起動している (図 4.20 の `Scheduler`)．そして，スケジューラは内部に保持しているワークフローの情報 (1) と実行環境の情報 (2) からスケジューリングを行う．各タスクの計算量や通信量，タスク間の依存関係といった情報は，ワークフローの情報として扱う．各計算ホストの計算性能や互いの通信性能については，実行環境の情報として扱う．

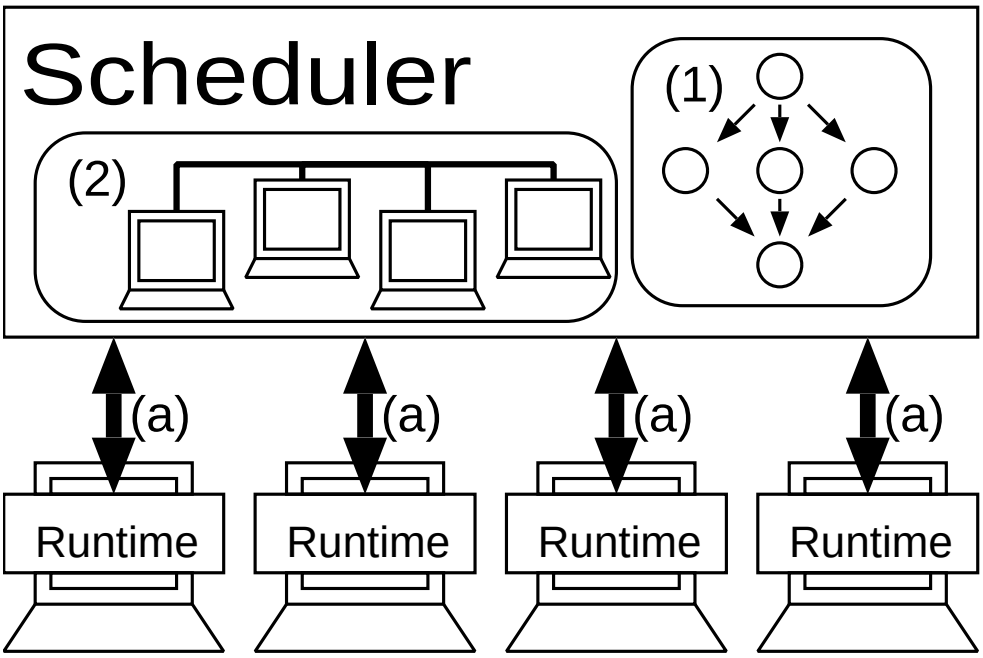

図 4.20: 実行モデルの例

またランタイムとスケジューラは多対一通信を行っており (図 4.20 の (a))，これによってタスクの実行やホストの性能変動に関する情報を通信している．例えば，図 4.21 の (a) ホストでタスク T_k を実行する場合，最初にランタイムはスケジューラからタスク T_k の実行開始命令を受け取る (図 4.21 の (1))．その後，直ちにホスト (a) はタスク T_k の実行プロセスを起動させる (図 4.21 の (2))．またランタイムは計算ホストの性能変動を検知することができ，その情報をスケジューラに通知する．例えば，図 4.21 の (b) のホストの性能が変化した場合，図 4.21 の (3) のようにホストのランタイムからスケジューラに通知する．

4.3.3 タスクソート関数とタスク割り当て関数に基づいた高速化

RAS はリストスケジューリングの一種である．リストスケジューリングの一般的なアルゴリズムを図 4.22 に示す．

リストスケジューラは最初に，2 行目の *SortByPriority()* 関数を呼び出

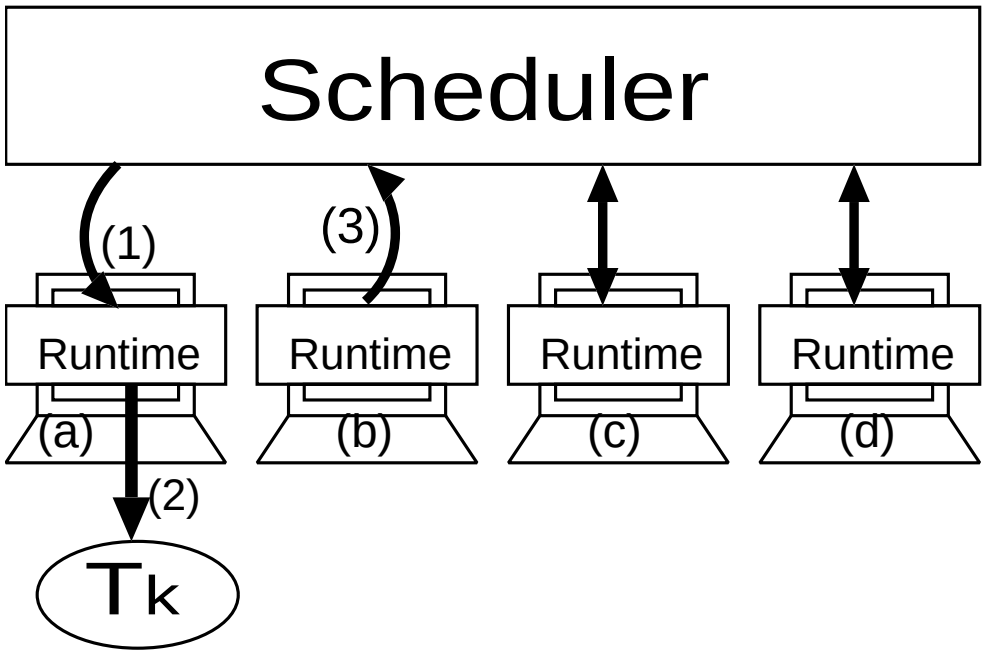

図 4.21: タスクの起動通知や性能変動通知

---

1. **function** *ListSchedule(workflow)*
2. *list* = *SortByPriority*(*all tasks*)
3. **while** (*list* is not empty) **do**
4. *targetTask* = *list.first*
5. *MapToHost*(*targetTask*)
6. Delete *targetTask* from *list*
7. **end**
8. **end**

図 4.22: 一般的なリストスケジューリングアルゴリズム

し，ホストへの割り当てを行うタスクの順番を求める (RAS アルゴリズムはタスクの計算コストと通信コストに基づいて各タスクの優先度を算出し，その算出した優先度に基づいてソートを行っている)．次に，ソートされた順番に従って各タスクをホストへと割り当てていく（3–7 行目）．実際にホストへの割り当ては *MapToHost*(*targetTask*) 関数を用いて行う．

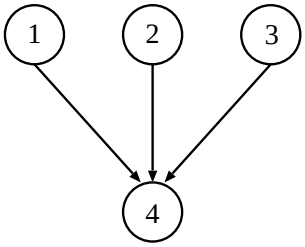

図 4.23: ワークフローの例

このようにリストスケジューリングでは二つのステップ（2 行目と 3–7 行目）を実行することでスケジューリングを行う．

DRAS でタスクを再スケジューリングする際に従来の RAS を用いた場合，動的性能変動の発生するたびに *SortByPriority()* 関数が呼び出される．しかし，*SortByPriority()* の算出において各計算ホストの計算性能は考慮されないため，その結果は常に同じになる．従って，動的性能変動の発生するたびに *SortByPriority()* 関数を呼び出すことはオーバーヘッドとなる．そこで改良型 RAS では，*SortByPriority()* 関数を初期スケジューリングの時だけ呼び出し，その結果を変数 *list* に保存する．そして，動的性能変動が発生し再スケジューリングを行う際は，*SortByPriority()* 関数を呼び出す代わりに *list* の値を参照する．従って，*SortByPriority()* 関数は一回しか実行されないのでスケジューリング時間の高速化が望める．

4.3.4 ホスト割り当て関数による高速化

従来用いられていた *MapToHost()* 関数は，ワークフローの実行中に呼び出されることを想定していないため，ワークフローの実行効率が悪化する可能性がある．

ここで，例を用いて説明する．図 4.24 は図 4.23 のワークフローをスケジューリングした結果を示している．図 4.24(a) は初期スケジューリングの結果を示している．ここで，時刻 t_1 で性能変動が生じたとする．この時，タスク 1 とタスク 2 は既に実行されているが，タスク 3 とタスク

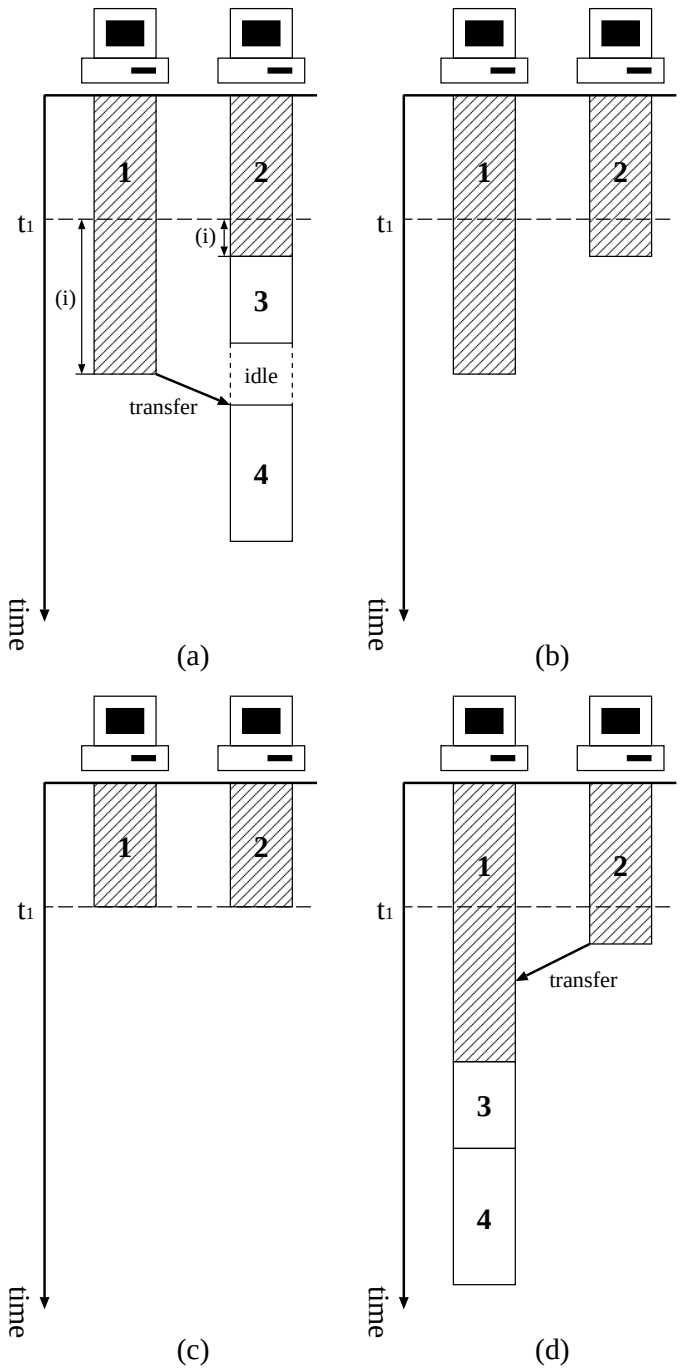

図 4.24: ホスト割り当て関数の問題

4 はまだ実行されていない．従って，タスク 3 とタスク 4 に対して再スケジューリングを行う必要がある．この場合，タスク 1 とタスク 2 はまだ図 4.24 (a) の (i) の分だけタスクの実行が残っている．従って，再スケジューリングする際の各ホストの状態としては，図 4.24 (b) のようになることが正しい．しかし，RAS アルゴリズムはワークフローの実行中に呼び出されることを想定しておらず，各ホスト上に実行中のタスクは無いものとしている．従って，時刻 t_1 の時点で RAS を呼び出した場合，各ホストの状態は図 4.24 (c) のようになってしまう．これを前提にスケジューリングを行うと，図 4.24 (d) のように不適切なスケジューリングが行われる可能性がある．

この問題を解決するために，改良型 RAS では前回のスケジューリング結果を保持しておく．そして，再スケジューリングする際の各ホストの状態は保存された結果から図 4.24 (b) のように復元する．こうすることで，正確な各ホストの状態を算出するために余分な計算をしなくても効果的なスケジューリングを行うことが可能となる．

4.3.5 再スケジューリング条件

従来の動的再スケジューリング手法では，性能変動が生じるたびに再スケジューリング処理を行っている．しかし，全ての再スケジューリング処理がワークフローの実行効率の向上に大きな影響を与えるとは限らず，一部の再スケジューリング処理はワークフローの実行効率に大きな影響を与えない可能性がある．例えば，あるタスク終了イベントから次のタスク終了イベントの間に複数回の性能変動が生じた場合について説明する．この場合，複数回の再スケジューリング処理におけるスケジューリング対象となるタスクは同一となり，それ以降で実行するタスクは一番直近で実行された再スケジューリング処理の結果が用いられる．従って，それ以外のスケジューリング処理は利用されず，これがオーバーヘッドとなる．また，独立型タスク群を実行中に再スケジューリングを行った場合もワークフローの実行効率に影響しない可能性がある．独立型タス

ク群に依存するタスクは，その独立型タスク群に含まれる全てのタスクの実行が完了するまで実行することができない．従って，その独立型タスク群に依存するタスクは，直近に実行された再スケジューリング処理の結果のみを利用し，それ以外の再スケジューリング処理は利用しない．そのため，利用されない再スケジューリング処理はオーバーヘッドとなる可能性が高い．2.2.1 節で説明したように対象とするワークフローには独立型タスク群が多く含まれており，このようなケースが多発すると考えられる．

これらのオーバーヘッドを削減するために，DRAS ではタスク実行終了のタイミングで再スケジューリングしている．また，前回のタスク実行終了から今回のタスク実行終了の期間に性能変動が生じなかった場合，再スケジューリングを実行しても前回の結果と変わらないため効果は無くオーバーヘッドとなるだけである．従って，この場合は前回のスケジューリング結果をそのまま用いてタスクを実行させる．また，独立型タスク群を実行中は再スケジューリング処理を行わないことにする．これらをまとめると，以下のいずれかの条件を満たした場合にのみ再スケジューリング処理を行う．

1. ある独立型タスク群に含まれる全てのタスクの実行が終了した場合
2. いずれの独立型タスク群にも含まれないタスク（以降，通常タスクと呼ぶ）の実行が終了した場合

ここで，実際にタスク終了を条件にした場合において，再スケジューリングの実行回数がどの程度低減されるか例を用いて説明する．図 4.26 は図 4.25 の独立型タスク群 (i) の実行完了前後の様子である．図 4.26 の (a) でホストの性能変動が発生したとする．動的性能変動を条件に再スケジューリングする場合，各 (a) のタイミングで再スケジューリングが行われるので，図 4.26 の時間内で 10 回実行される．次にタスクの実行終了を条件に再スケジューリングする場合について説明する．タスクの実行終了を条件にした場合では (b)，(c)，(d) のタイミングは再スケジューリング処理

実行の候補になるが，二回目の (b) と三回目の (b) の間で性能変動は発生していないため，三回目の (b) のタイミングでは再スケジューリングは実行されない．従って，この時間内では 5 回の再スケジューリングが実行される．最後に関数を用いた場合について説明する．*TriggerITSFinished()* 関数では，(c) のタイミングで `sim` の独立型タスク群に含まれる全てのタスクが実行完了となり，この時に再スケジューリングが実行される．また，タスク `composite` はいずれの独立型タスク群にも含まれていないため，タスク `composite` の実行が完了した (d) のタイミングでも再スケジューリングが実行される．従って，この時間内では 2 回の再スケジューリングが実行される．

また，図 4.25 のワークフローを用いて，全てのタスクの実行終了を条件とした場合と特定のタスクの実行終了を条件とした場合の再スケジューリングの実行回数を比較する．この時，ホストの性能変動は頻繁に発生しているものとし，各タスクが実行完了する間に性能変動が 1 回以上発生しているものとする．全てのタスクの実行終了を条件とした場合，ワークフロー内にタスク数は 2,366 個存在するため，2,366 回のスケジューリング処理が行われる．これに対して特定のタスクの実行終了を条件とした場合，9 回のスケジューリング処理が実行される．従って，スケジューリング回数を約 1/250 に減らせる．

このような仕組みを導入することで，効果の薄い再スケジューリング回数を省き，スケジューリング処理の高速化を行う．

4.3.6 アルゴリズム

図 4.27 に DRAS のアルゴリズムの詳細を示す．*DRAS()* は図 4.19 で示した一般的な動的再スケジューリング手法のアルゴリズムと似ている．しかし，7–12 行目が図 4.19 の 7–10 行目と異なる．DRAS は *CheckTriggers()* を用いて再スケジューリングを行うか否かを調べている．もし再スケジューリング条件を満たした場合，*CheckTriggers()* は true を返す．多くの一般的な動的再スケジューリング手法は DRAS と異なり，動的性能変動が生じる

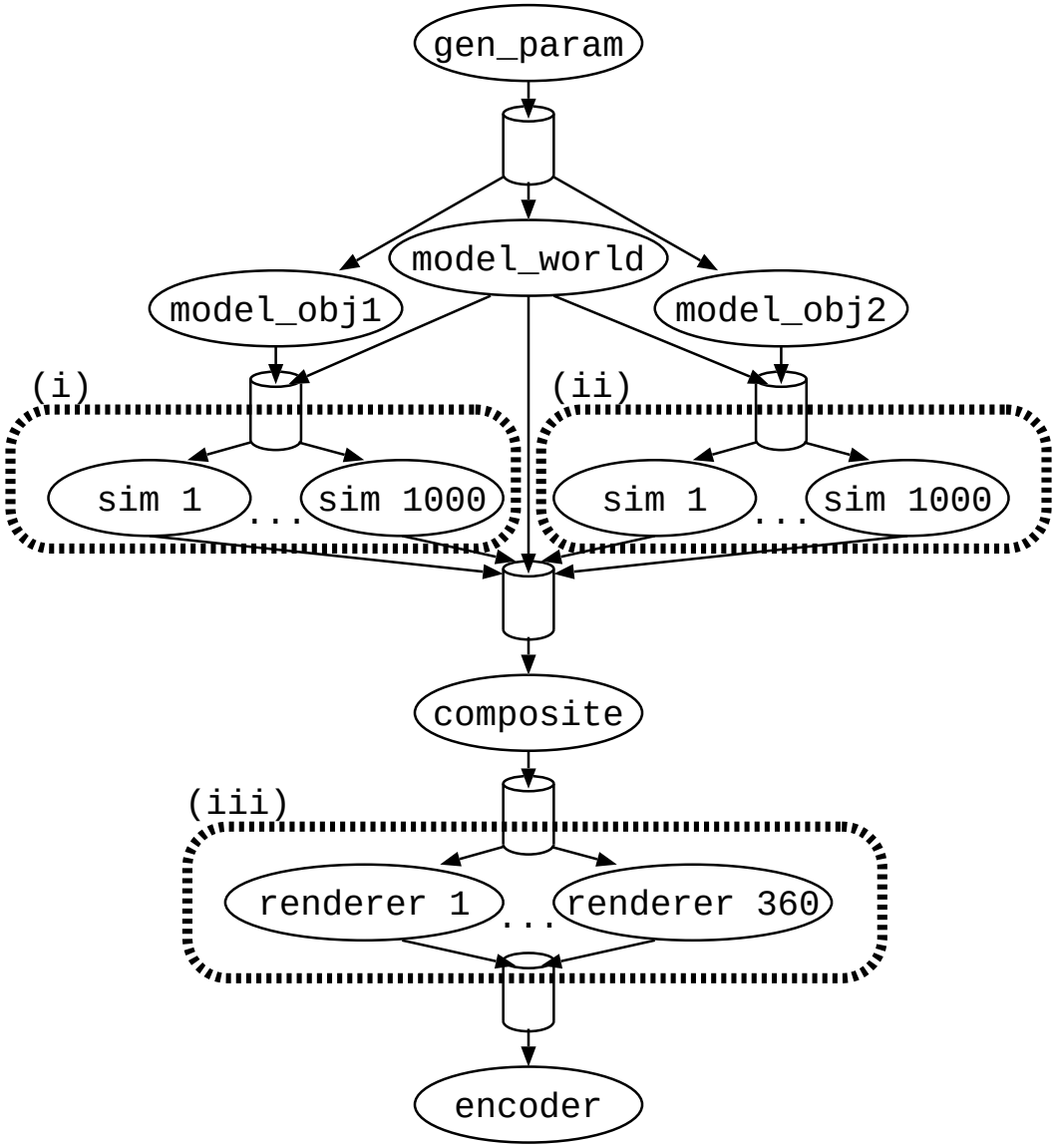

図 4.25: 再スケジューリングの実行例のワークフロー

たびに毎回再スケジューリングを行う．しかし DRAS は *CheckTriggers()* を導入することで，再スケジューリング回数を減らしスケジューリング時間を削減している．

図 4.28 に *CheckTrigger()* の詳細なアルゴリズムを示す．*CheckTrigger()* は最初に，ワークフローシステムが通知したイベントがタスクの実行終

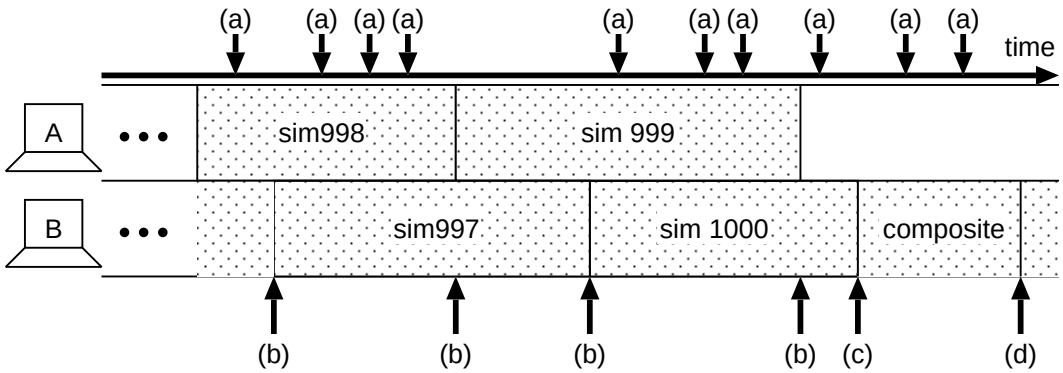

図 4.26: 再スケジューリングの実行例

```
1. function DRAS(T, C)
2.     globalList = SortByPriority(T)
3.   RasSchedule(globalList, C)
4.   while (all tasks in T are not finished) do
5.     sleep until receiving any event
6.     if event is TaskStartedEvent then
7.       list = list - the finished task T_i
8.     else if event is PerformanceChangedEvent then
9.       update C
10.    end
11.    if CheckTriggers(event) = true then
12.      RasSchedule(globalList, C)
13.    end
14.  end
15. end
```

図 4.27: *DRAS()* アルゴリズム

了に関するイベントかどうかをチェックする（2 行目）．もし実行が終了したタスクが通常タスクの場合（独立型タスク群に含まれるタスクでない場合），再スケジューリングを実行する必要があるので *CheckTrigger()* は true を返す（8–9 行目）．もし実行終了したタスクが独立型タスク群 I_i

```
1. function CheckTriggers(event)
2.   if event is FinishedTaskEvent then
3.     task = the finished task.
4.     if task is a member of ITS I_i then
5.       if all tasks in I_i are finished then
6.         return true
7.       end
8.     else
9.       return true
10.    end
11.  end
12.  return false
13. end
```

図 4.28: *CheckTrigger()* アルゴリズム

に含まれるタスクで，かつ独立型タスク群 I_i に含まれる全てのタスクの実行が終了した場合，同様に再スケジューリングを実行する必要があるので *CheckTrigger()* は true を返す．もし独立型タスク群 I_i に含まれるタスクの中で実行が完了していないタスクが存在した場合，これは再スケジューリング処理を省略するために *CheckTrigger()* は false を返す．従って，DRAS は *CheckTrigger()* を呼び出すことで特定のタスクが終了した場合のみ再スケジューリングを行っており，これによってスケジューリング時間の大幅な削減を期待している．

図 4.29 は *RasSchedule()* のアルゴリズムの詳細を示している．*RasSchedule()* は図 4.22 で示した一般的なリストスケジューリング手法のアルゴリズムと似ている．しかし，2 行目が一般的なリストスケジューリング手法のアルゴリズムと異なる．*RasSchedule()* は複数回呼び出されるが，4.3.3 節で述べたように，*SortByPriority()* の実行結果は常に同じである．従って，DRAS は *SortByPriority()* を一回だけ実行し（図 4.27 の 3–5 行目），その結果を変数 *list* に保存する．そして，*RasSchedule()* は *SortByPriority()* を実行する代わりに，*list* に格納された結果を利用して

```
1. function RasSchedule(list, C)
2.    Copy list to tasklist
3.    while (tasklist is not empty) do
4.       targetTask = tasklist.dequeue
5.       MapToHost(targetTask)
6.    end
7. end
```

図 4.29: *RasSchedule()* アルゴリズム

スケジューリングを行う．

*MapToHost()*はHEFTや従来のRASのアルゴリズムと似ている．HEFTや従来のRASアルゴリズムは各ホストに対して$EST(T_i, H_p)$や$EFT(T_i, H_p)$を計算し，そしてタスクT_iを$EFT(T_i, H_p)$の値が最小となるホストH_pへと割り当てる．$EST(T_i, H_p)$や$EFT(T_i, H_p)$は3.3.4節で説明した処理と同様で，それぞれがタスクT_iがホストH_p上で実行可能となる最小時刻と実行終了の最小となる時刻を表している．そしてそれらは

$$EST(T_i, H_p) = max\{eat(H_p), max_{T_j \in pred(T_i)} (EFT(T_j, H_q) + comm(T_j, T_i))\}$$

$$EFT(T_i, H_p) = comp(T_i, H_p) + EST(T_i, H_p)$$

で定義されている．$pred(T_i)$はタスクT_iに直接依存するタスクの集合を返す関数である．$eat(H_p)$はホストH_pでタスクの実行が可能になる最小時刻を返す関数である．$comm(T_j, T_i)$はタスクT_jからタスクT_iへの通信時間を示しており，$comp(T_j, H_p)$はタスクT_jをホストH_p上で実行した時の実行時間を示している．従来のRASアルゴリズムを動的再スケジューリング手法のアルゴリズムとして利用した場合，ワークフローの実行中に再スケジューリングする時は未実行のタスクに対して再スケジューリングを行えばよい．しかし，各タスクのEFTは実行中のタスクについて考慮しておらず，それを元にスケジューリングを行った場合ワークフロー

の実行効率が低下する恐れがある（詳細については 4.3.4 節を参照)．従って，RAS を DRAS で用いるにはこの EFT の計算処理を実行中タスクについても考慮するように改良する必要がある．しかし，実行中タスクについても計算処理を行うと，EFT の値は再スケジューリング処理中で頻繁に参照されるため，スケジューリング時間が大幅に増加してしまう可能性がある．従って，少ない計算コストで EFT を求める必要がある．そこで改良型 RAS は実行中の EFT 値を算出するのではなく，前回のスケジューリング結果を用いる．これによって計算コストをかけずに実行中タスクの EFT の値を得ることが可能になる．そして，次回以降の再スケジューリング処理でも同様に EFT の値を利用するために，今回の再スケジューリング処理で求めた値を保存しておく．

4.3.7　シミュレーション評価の方法

本評価では 3 章の評価と同様に実際にタスクの実行を行わず，以下の条件によりイベントドリブン型の抽象シミュレーションを行った．各タスクの計算時間はそのタスクの計算量に従い，通信時間はそのタスクの通信量に従うものとした．

- 各タスクの通信は，そのタスクの実行終了時に行う．
- 各ホストでは一度に一つのタスクしか実行せず，スケジューリングされた順序は追い越さないものとする．つまり，次に実行するべきタスクが依存関係により実行できない場合，そのホストは当該タスクが実行可能になるまでアイドル状態となる．
- 実行時間は最初のタスクの実行開始から全てのタスクが終了するまでとし，スケジューリング自体に要する時間は考えない．
- タスク間通信が同一ホスト上で行われる場合の通信時間は 0 とする．

4.3.8　シミュレーション評価における条件

本評価では以下の手順により，ランダムなワークフローを生成した．

1. 初期構造として，2 個のタスクを 1 本のストリームで接続したワークフローを生成する．
2. 以下のいずれかの操作を等確率で適用する．

   連結 ランダムにタスクまたは独立型タスク群 T_i を選択する．ただし，ワークフローの先頭のタスクは除外する．新しくタスク T_j とストリーム S_k を生成し，T_i を，T_i と T_j を S_k で接続した構造で置き換える．

表 4.9: DRAS 評価におけるワークフローの生成条件

タスクの計算量	100 - 200
タスクの基本通信量	100 - 200

コントロール並列 ランダムにタスクまたは独立型タスク群 T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しくタスク T_j を生成し，T_i と同じ入力側・出力側ストリームを持つように接続する．

データ並列 ランダムにタスク T_i を選択する．ただし，ワークフローの先頭と末尾のタスクは除外する．新しく独立型タスク群 T_j を生成し T_i と置き換える．

3. タスクの総数が設定値以下であれば，(2) から繰り返す．

各タスクの計算量や通信量について，表 4.9 の範囲でランダムに選んだ値を利用した．タスクの通信量については各評価で用いる CCR 値 (Communication to Computation Ratio) を使い，

$$\text{タスクの通信量} = \text{タスクの基本通信量} * \text{CCR 値} \tag{6}$$

とする．これにより CCR 値が大きい場合，全体的にタスクの計算量より通信量が大きいワークフローとなり，CCR 値が小さい場合はその逆となる．実行環境については表 4.10 の条件に従ってランダムに生成した．

また計算ホストの性能変動をシミュレーションするために，疑似タスクを計算ホストの半数で発生させ，ホストの計算性能は疑似タスクの個数に反比例するようにした．以下は各ホストの計算性能を求める式である．

$$\text{ホストの計算性能} = \frac{\text{ホストの基本計算性能}}{1 + dummy\_task(H_i)} \tag{7}$$

ホストの基本計算性能はスケジューラ等で用いた計算ホストの計算性能と同様で，予め実行環境情報として与えられたホストの計算性能である．

表 4.10: DRAS の評価における実行環境の生成条件

ホスト間通信性能	100
ホストの計算性能	1.0 - 5.0
ホストの台数	16

また，$dummy\_task(H_i)$ 関数は計算ホスト H_i に存在している疑似タスク数を返す関数である．疑似タスクの生成頻度は単位時間に平均 λ 回のポアソン分布に従っている．一方，疑似タスクの消滅は時間間隔が平均 $1/\lambda$ の指数分布に従っている．

それぞれの評価で示すデータは，同じ条件でランダムに生成した 10 種類のデータセットに対しそれぞれシミュレーションを行った平均値である．今後，より広い範囲におけるデータセットで評価を行い，様々な実アプリケーションや実行環境における効果を確認していく必要がある．

4.3.9 シミュレーション評価

DRAS のスケーラビリティについて比較評価を行った．比較対象には，一般的な動的再スケジューリング手法に従来の RAS をスケジューリングアルゴリズムとして実装したもの（*conventional*）を用いた．表 4.11 は各手法のスケジューリング時間とワークフローの実行時間比の平均値である．実行時間比はスケジューリング長の *conventional* に対する DRAS の比である．λ は 4.3.8 節で説明した各ホストへ疑似タスクを生成する割合で，$1/\lambda$ を 100 として評価を行った．

実行時間比は約 50 – 85% になった．従って，DRAS のスケジューリング長は *conventional* と比較して全てのケースにおいて減少した．これは 4.3.4 節で説明した，正確な情報を用いてスケジューリングした結果である．また，スケジューリング時間も *conventional* と比較して，約 70 – 85% ほど減少した．これは，DRAS は 4.3.3 節で述べたオーバーヘッド削減手法や 4.3.5 節で説明したような再スケジューリングの実行回数削減手法を

表 4.11: DRAS に対するスケーラビリティの評価

No. of Tasks	100	500	1,000
DRAS (sec)	0.594	41.150	73.781
conventional (sec)	4.513	131.602	440.606
execution time ratio	0.497	0.833	0.741

表 4.12: Performance with Dynamic Changes Frequency

$1/\lambda$	100	500	1000
DRAS (sec)	73.781	22.800	14.597
conventional (sec)	440.606	58.934	32.117
execution time ratio	0.741	0.697	0.695

適応させているためである．また，スケジューリング長が短くなることでワークフローの実行時間が短くなり，それに伴ってワークフローの実行中に生じる動的性能変動の発生回数が減少する．従って，短いスケジューリング長を得ることはワークフローの実行時間が短縮されるだけでなく，再スケジューリング回数が減ることでスケジューリング処理の高速化も見込める．

また，動的性能変動の発生頻度による DRAS の性能評価を行った．表 4.12 は 1,000 タスクのワークフローに対して評価を行った結果である．実行時間比は先ほどと同様である．$1/\lambda$ を 100，500，1,000 に設定して評価を行った．$1/\lambda$ が小さいほど，動的性能変動の発生頻度は低くなる．

DRAS は全てのケースにおいて *conventional* よりスケジューリング時間が短くなった．特に，DRAS は性能変動が頻繁に生じる環境においてスケジューリング時間が大幅に減少した．これは，*conventional* は一般的な動的再スケジューリング手法であり再スケジューリング回数は性能変動が生じた発生回数に依存する．これに対し，DRAS はワークフロー中のタスク数に依存するため，性能変動が生じた発生回数に関わらず常に

同程度の再スケジューリング実行回数になる．これによってスケジューリング時間は *conventional* と比較して十分に高速化されたといえる．また，DRAS はワークフローの実行効率においても全てのケースで *conventional* と比較して短いスケジューリング長を得た．これはスケーラビリティの評価と同様に，4.3.4 節で説明した，正確な情報を用いてスケジューリングした結果である．従って，DRAS は頻繁に動的性能変動が生じる環境で大きな利点がある手法である．

4.4 まとめ

本章では，動的性能変動の生じる環境で高速に動作する動的再帰的適応型スケジューリング手法について提案した．これは，従来の動的再スケジューリング手法に 3 章で説明した RAS を用いる際に，冗長となる処理を省いている．また，動的性能変動が生じるごとに毎回再スケジューリングを行う従来手法において，効果の薄い再スケジューリング処理を省略することによってスケジューリング処理の高速化を図っている．

DRAS を実装し抽象シミュレーションを用いて比較評価を行った．その結果，従来の動的再スケジューリング手法に 3 章で説明した RAS をそのまま利用した手法と比較して，DRAS のスケジューリング長を 30% ほど削減することができた．また，動的性能変動の発生頻度が高い環境で 1,000 タスクのワークフローをスケジューリングした結果，スケジューリング処理を 6 倍ほど高速化することができた．

5　結論

本章では，前章内容を要約するとともに今後の課題について述べ，本論文をまとめる．

5.1　本研究のまとめ

本論文では，ワークフローシステムの実行効率を高めるためのスケジューリング手法の研究に焦点を当て，動的性能変動の生じる実行環境に適した大規模ワークフローの高速スケジューリング手法の開発について検討を行った．1 章では，本研究の背景とタスクスケジューリングの概要，従来のスケジューリング手法の現状について述べるとともに，本研究の目的を述べ，本研究の位置づけを行った．

3 章では本研究で開発した静的スケジューリング手法の高速化について述べた．3 章では，下位層のスケジューラで扱うタスクの依存関係に着目し，適応的にスケジューリング処理を切り替える適応型スケジューリング手法 (AS) について述べた．これは，HEFT と比較してスケジューリング時間を 1/50 ∼ 1/100 程度に削減できた．しかし BLAST を用いた遺伝子解析のワークフローのように類似性の高いサブワークフローが複数含まれている場合，AS のアルゴリズムでは処理の切り替えを効率的に行うことができず，スケジューリングの計算時間の大幅な削減が行えない可能性がある．そこで AS を改良し，スケジューリング処理の適応的な切り替えを再帰的に行う再帰的適応型スケジューリング手法 (RAS) について述べた．RAS を抽象シミュレーションで評価を行った結果，10,000 タスク規模でスケジューリングの計算時間を 1/100 程度に低減できた．

4 章では，3 章で述べる静的スケジューリングを動的性能変動に対応させた動的再帰的適応型スケジューリング手法 (DRAS) について述べた．これは動的再スケジューリング手法の一種だが，高い計算コストを削減させるための幾つかのテクニックを適応させている．抽象シミュレーションで評価を行った結果，1,000 タスク規模でスケジューリングの計算時間

を 1/6 程度に削減できた．これによって，DRAS が性能変動の生じる実行環境に適した大規模ワークフローの高速スケジューリング手法であることを示した．

5.2 今後の課題

2.3 節で述べたように，本研究は階層型スケジューリング手法の局所スケジューラに適応させることを焦点において，スケジューリングアルゴリズムの高速化と，動的性能変動の生じる環境でスケジューリング結果を補正することでワークフローの高効率化を目指している．今後は DRAS を改良し大域スケジューラにも適応させることで，広域分散環境上に存在する複数のクラスタを利用できるようにすることが必要となる．この場合，クラスタ間のデータ通信性能やクラスタの性能だけではなく，実行環境のトポロジーなども考慮してスケジューリングを行う必要がある．また複数のクラスタを利用することで動的性能変動の発生頻度はより高くなると考えられる．このような中で，動的性能変動に対するスケジューリング結果の補正をマスターワーカー型のようなスタイルで行うだけでなく，ランダムスチール法のようなデマンドドリブン型のようなスタイルでもスケジューリング結果を補正するスタイルも考慮されるべきである．

また，ホスト間の通信性能においてバンド幅を考慮したり，実際の実行環境を調査しより現実的な性能変動モデルを利用し，より現実的なシミュレーション評価を行っていく必要がある．また，本論文では現実のワークフローとして Epgenopics ワークフローのみで評価を行ったが，それ以外の実ワークフローについても評価を行っていきたい．そして，シミュレーション評価だけでなく，実ワークフローに対して InTrigger 等の実際に存在するマルチクラスタ環境を使い，実評価を行っていく必要がある．

参考文献

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] D.J.Lipman and W.R.Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[3] J.D.Thompson, D.G.Higgins, and T.J.Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Reseach*, 22:4673–4680, 1994.

[4] http://ambermd.org/. The amber molecular dynamics package.

[5] http://www.r project.org/. R.

[6] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, Jun 1990.

[7] G. C. Sih. and E. A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *Proceedings of IEEE Symposium on Parallel and Distributed Processing*, 1990.

[8] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par'96 Parallel Processing*, 1996.

[9] H.Topcuoglu, S.Hariri, and M.-Y. Wu. A high-performance mapping algorithm for heterogeneous computing system. *IPPS/SPDP Workshop on Heterogeneous Computing*, pages 3–14, 1999.

[10] A.Dogan and R.Ozguner. Ldbs:a duplication based scheduling algorithm for heterogeneous computing systems. *Proc. Int'l Conf. Par. Proc.*, pages 352–359, 2002.

[11] C. Boeres, E. Rios, and L. S. Ochi. Hybrid evolutionary static scheduling for heterogeneous systems. In *The 2005 IEEE Congress on Evolutionary Computation*, 2005.

[12] G. H. Kim and C. S. G. Lee. Genetic reinforcement learning for scheduling heterogeneous machines. In *1996 IEEE International Conference on Robotics and Automation*, 1996.

[13] Min you Wu and Daniel D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transaction of Parallel and Distributed Systems*, 1(3):330–343, 1990.

[14] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transaction of Parallel and Distributed Systems*, 7(5):506–521, 1996.

[15] R.Armstrong, D.Hensgen, and T.Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. *7th IEEE Heterogenous Computing Workshop*, pages 87–97, 1998.

[16] Min-You Wu, Wei Shu, and H. Zhang. Segmented min-min: a static mapping algorithm for meta-tasks onheterogeneous computing systems. In *Heterogeneous Computing Workshop, 2000*, pages 375–385, 2000.

[17] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in

run-time predictions. In *7th IEEE Heterogeneous Computing Workshop(HCW '98)*, 1998.

[18] R. D. Blumofe and C.E.Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS'94)*, pages 34–43, 1994.

[19] Zhifeng Yu and Weisong Shi. An adaptive rescheduling strategy for grid workflow applications. *Parallel and Distributed Processing Symposium,International*, 0:1–8, 2007.

[20] M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *In Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing*, 2007.

[21] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. *Fifth IEEE International Symposium on Cluster Computing and the Grid*, 2:759–767, 2005.

[22] 松本真樹, 片野聡, 佐々木敬泰, 大野和彦, 近藤利夫, and 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**, 2(1):1–17, jan 2008.

[23] Masaki Matsumoto, Kazuhiko Ohno, Takahiro Sasaki, and Toshio Kondo. A recursive adaptive scheduling scheme for large-scale workflows. In *Proceedings of the 23th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 138–145, 2011.

[24] まつもとゆきひろ and 石塚圭樹. **オブジェクト指向スクリプト言語** *Ruby*. ASCII, 1999.

[25] http://setiathome.berkeley.edu/. Seti@home.

[26] 片野 聡, 森 英一郎, 大野 和彦, 佐々木 敬泰, 近藤 利夫, and 中島 浩. 不均質環境におけるタスクネットワークの静的スケジューリング手法. In **情処研報** *2007-HPC-111*, pages 37–42, 2007.

[27] T.D. Braun et. al. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. *IPPS/SPDP Workshop on Heterogeneous Computing*, pages 15–29, 1999.

[28] M.Srinivas and L.M.Patnaik. Mapping and scheduling heterogeneous task graphs using genetic algorithms. *5th IEEE Heterogenous Computing Workshop*, pages 86–97, 1996.

[29] 斎藤秀雄, 鴨志田良和, 澤井省吾, 弘中健, 高橋慧, and 関谷岳史. 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. In **情処学研報** *HPC-111(SWoPP 2007)*, pages 237–242, 2007.

[30] Kazuhiko Ohno, Akihiro Mita, Masaki Matsumoto, Takahiro Sasaki, Toshio Kondo, and Hiroshi Nakashima. Efficient implementation of large-scale workflows based on array contraction. In *Proceedings of the 22th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 153–162, nov 2010.

謝辞

本研究を行うにあたり，御指導，御助言頂きました大野和彦講師，並びに多くの助言を頂きました近藤利夫教授，佐々木敬泰助教に深く感謝致します．また，様々な局面にてお世話になりました研究室の皆様にも心より感謝いたします．